Test Driven Developement-menetelmän tehokkuus ja laatu

Petri Pihlajaniemi

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 2. 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	2. 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. *Stoica et al* 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 merkistystä 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. 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]. En usko TDD:n käytön vaikuttavan asiakkaan näkemään tuotteen arvoon, vaikka teoriassa menetelmän käyttö saattaisi sitä hieman lisätä.

Vanha versio, onkohan tämä uusi nyt parempi... Côté, Suryn, ja Georgiadoun artikkelin *In search for a widely applicable and accepted software quality model for software quality engineering* mukaan tietojenkäsittelytieteessä ei olla päästy yksimielisyyteen laadun tarkasta merkityksestä, vaan sillä on tarkoitettu useita eri asioita. Standardointijärjestö ISO:n mukaan laatu viittaa siihen, kuinka hyvin tuotteen ominaisuudet vastaavat annettuihin vaatimuksiin. Jim Highsmith laskee laaduksi asiakkaalle tuotetun lisäarvon ja virheiden määrän. Nämä kuvaukset eivät kuitenkaan ole saavuttaneet konsensusta, ehkä koska ne eivät tunnista Kitchenhamin ja Pfleegerin esittämiä perspektiivejä laatuun; metafyysinen, käyttäjän, teollisuuden, tuotteen ja lopullisen perspektiivin. Metafyysinen tai transdentti perspektiivi

viittaa pyrkimykseen täydelliseen laatuun jota ei ehkä ikinä saavuteta. Käyttäjän perspektiivi kysyy onko tuote sopiva käyttötarkoitukseen, johon sitä tarvitaan. Teollinen perspektiivi esittää laadun annettujen vaatimuksien toteuttamisena. Tuotteen perspektiivi mittaa laatua tutkimalla tuotetta, sen ominaisuuksia tutkitaan ja niistä rakennetaan arvio lopullisesta laadusta. Lopullinen perspektiivi perustuu arvoihin, erilaiset osapuolet saattavat nähdä laadun perspektiivit eri tärkeysjärjestyksessä. [CSG07]

Laadun käsitettä on ehkä käsitelty liikaa teollisuuden näkökulmasta, kun standardien noudattamisesta on tehty markkinointityökalu ja sopimuspykälä. 1960-luvulla IBM:n and Yhdysvaltain puolustusministeriön näkemyksen mukaan tiukka ohjelmistotuotantoprosessin noudattaminen johtaa laadukkaaseen tuotteeseen, tämä näkemys on kuitenkin useiden tutkijoiden mukaan väärä. Geoff Dromeyn mukaan liika keskittyminen prosessiin tapahtuu muiden laatumallien kustannuksella ja Kitchenhamin pa Pfleegerin mukaan prosessi varmistaa ainoastaan lopputuloksen samankaltaisuuden. Myös Agile-projektit ovat näyttäneet korkean laadun olevan mahdollinen ilman tiukkaa prosessia. Toistaalta eräät tutkimukset ovat havainneet hyvän prosessin paljastavan virheet koodissa aiemmin. Teolliseen perspektiiviin perustuvat menettelytavat eivät myöskään sovellu pienille projekteille tai pienille kehittäjäryhmille. Tavoite onkin löytää laatumalli joka ottaisi kaikki perspektiivit huomioon.

Cote et al listaavat artikkelissaan vaatimukset laatumallille: sen pitää ottaa huomioon Kitchenhamin ja Pfleegerin perspektit, sille pitää pystyä antamaan vaatimuksia (top to bottom) ja sitä pitää pystyä mittamaan (bottom to top) IEEE:n standarin mukaisesti. Artikkelissa on tarkasteltu neljää erilaista laatumallia. [CSG07]

McCallin, Richardsin ja Waltersin laatumalli on esitellyistä laatumalleista vanhin ja Pfleegerin mukaan yksi ensimmäisistä julkaistuista. Malli koostuu laatuun vaikuttavia tekijöistä, joita ei voi suoraan mitata. Mallin mittarit ovat mitattavia asioita, joiden avulla laatutekijöitä voidaan arvioida. Ongelmana on kuitenkin se, että mitattavat asiat ovat usein subjektiivisia eivätkä siten sovellu tarkkojen laatuvaatimusten asettamiseen. [CSG07]

Boehmin malli on McCallinin mallista kehitelty, mutta laadun tärkeimmäksi tekijäksi on nostettu "yleinen hyödyllisyys"(general utility), jonka mukaan ohjelman on oltava hyödyllinen ollakseen laadukas. Hyödyllisyys koostuu ohjelman käyttön helppoudesta ja tehokkuudesta, ylläpidosta ja siirrettävyydestä. Boehm on myös järjestänyt laatutekijät sen mukaan, mitkä kiinnostavat enemmän teknisiä osapuolia ja mitkä loppukäyttäjää. Cote et al mukaan mallin tekijät ovat kuitenkin yhä liian teknisiä ja mittarit yleisen hyödyllisyyden mittarit liian geneerisiä. [CSG07]

Dromeyn malli keskittyy laatuun tuotteen perspektiivistä. Ohjelman komponentteja tutkitaan konkreettisten laatuominaisuuksien perusteella. Esimerkiksi muuttujat, funktiot ja ehtolausekkeet ovat mallissa komponentteja. Dromeyn komponenttien ominaisuudet voidaan jakaa neljään ryhmään: kor-

rekti (correctness), sisäinen, kontekstinen ja kuvaava. Korrektius mittaa, onko perusperiaatteita rikottu. Sisäinen liittyy siihen, onko komponenttia käytetty sopivalla tavalla sen tarkoitusta varten. Kontekstinen käsittelee ulkoisia vaikutuksia komponenttiin. Kuvaava viittaa esimerkiksi komponentin nimeen ja muihin samankaltaisiin koodin lukemista helpottaviin tekijöihin Dromeyn hypoteesi on, että korkean tason laatua kuvaavat tekijät tulevat esille jos ohjelman matalan tason muuttujat jne. ovat laadukkaita. Artikkelin kirjoittat pitävät hypoteesia vääränä, Dromey keskittyy liikaa konkreettisiin pieniin yksityiskohtiin: loppukäyttäjää ei kiinnosta muuttujien nimeämisen kaltaiset tekniset yksityiskohdat. [CSG07]

Lopuksi artikkeli esittelee ISO/IEC 9126 laatumallin. Vuonna 1991 julkaistu malli saavutti nopeasti mainetta parhaana tapana mitata laatua, mutta Pfleegerin mukaan siitä löytyi muutamia merkittäviä ongelmia. ISO/IEC julkaisikin mallista uusia versioita, jotka pyrkivät vastaamaan löydettyihin ongelmiin. Malli määrittelee kolme laadun osa-aluetta. Käyttölaatu (quality in use) mittaa käyttäjän tyytyväisyyttä ohjelmaan, kun sitä käytetään tietyssä järjestelmässä tiettyyn tarkoitukseen. Itse ohjelmaa ei mitata, vaan tavoitteita jotka sen käyttäjä saavuttaa. Ulkoinen laatu on ohjelman toiminta ajettaessa, käytännössä testaamisen avulla löydettyjen virheiden määrä. Sisäinen laatu käsittelee ohjelman sisäistä rakennetta, ja sitä voidaan parantaa katselmoinnilla ja testaamisella. Mallin on saanut alalla huomiota, ja monet tahot ovat siirtyneet käyttämään sitä. Cote et al toteavat ISO/IEC9126 mallin olevan ainoa heidän esittelemistää malleista, jotka toteuttavat niille annetut vaatimukset. [CSG07]

Artikkelin perusteella olemme siis löytäneet mallin mitata laatua. Voidaksemme käsitellä TDD:n vaikutusta laatuun, pitää laatuun vaikuttavia mittareita tarkastella tarkemmin.

2.1 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 et al 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, Gyimothy et al 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 Gyimothy et al tutkimista mittareista, paljon koodirivejä sisältävissä luokissa oli enemmän virheitä. [GFS05]

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

2.2 Tehokkuuden mittareita

babab

3 Test Driven Development

babab

3.1 Mitä se on? Miten eroaa muista? Miksi käyttää sitä?

babab

3.2 Tutkimuksia

babab

3.3 Tuloksia

babab

3.3.1 Laatu

babab

3.3.2 Tehokkuus

babab

3.4 Meta-analyysit

babab

4 Käyttö yritysmaailmassa

babab

Lähteet

[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.

[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.

[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.