Metriikat käytänteiden tukena ohjelmiston laadun arvioimisessa

Kasper Hirvikoski

Kandidaatintutkielma - Luonnosversio
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 16. maaliskuuta 2013

Sisältö

1 Johdanto

Ohjelmistot kehittyvät elinkaarensa aikana muun muassa uusien vaatimusten, optimisaatioiden, tietoturvaparannusten ja virhekorjausten johdosta. Kehitysvaiheessa olevan ohjelmiston laadun varmistaminen on hankalaa [BBM96, NB05, NB07, ZN08, MNDT09]. Ohjelmiston testaamisen ja käytännössä havaittujen virheiden välillä on usein suuri kuilu. Virheiden määrää ei yleensä pystytä laskemaan luotettavasti ennen kuin tuote on valmis ja julkaistu asiakkaalle. Tässä piilee kuitenkin ongelman ydin: virheiden korjaaminen ohjelmiston julkaisun jälkeen on erittäin kallista.

Ohjelmiston kehittäminen on haastavaa. Vielä haastavampaa on kehittää laadukkaasti suunniteltu ja toteutettu ohjelmisto. Käyttäjät havaitsevat laadun oikein toimivana tuotteena, mutta ennen kaikkea laadukkaasti toteutetun ohjelmiston lähdekoodi helpottaa kehitysprosessia. Ongelmien korjaamisen sijaan kehittäjät voivat keskittyä olennaiseen eli uusien toiminnallisuuksien toteuttamiseen. Virheitä on kuitenkin mahdotonta välttää täysin.

Laadun varmistamista rajaa ohjelmistokehityksessä henkilöt, aika ja raha [BBM96, ZN08]. Kehittäjät kohtaavat usein tiukkoja määräaikoja ja rajallisia henkilöresursseja laadun takaamiseen. Johtajat käyttävät käytännössä pelkästään omakohtaisia kokemuksiaan resurssien tehokkaaseen jakamiseen. Heillä ei kuitenkaan ole läheskään aina tarvittavaa kokemusta tai tietoa, joiden pohjalta he voisivat tehdä järkeviä päätöksiä ohjelmiston laadun kannalta. Tästä johtuen päätökset tehdään usein johtajien odotusten mukaan ja tällöin he itse joutuvat arvioimaan laatua puutteellisin tiedoin. Kriittiseksi osaksi muodostuu näin ollen kehittäjien taitojen lisäksi johtajien taidot.

Laadun varmistaminen ja mahdollisten ongelmakohtien havaitseminen mahdollisimman aikaisessa vaiheessa hyödyttää kehitystyötä [BBM96, NB05]. Ohjelmiston koodin tuottajana on ihminen, joten ohjelmiston laatuun vaikuttuvat inhimilliset tekijät. Kehittäjän käytänteillä on suuri laadullinen merkitys ohjelmiston kaikissa kehitysvaiheissa. Tämän lisäksi ohjelmiston laatua voidaan arvioida mekaanisilla metriikoilla, jotka pyrkivät arvioimaan ohjelmiston komponenttien laatua ja havaitsemaan kriittiset osat tuotteesta.

2 Laadullinen arviointi

ISO 9000 -standardi määrittelee ohjelmiston laadun kokonaisuutena, joka kattaa tuotteen tai palvelun piirteet, jotka täyttävät ohjelmistolle asetetut toiveet ja tarpeet [ISO05]. IEEE taas määrittelee laadun arviona, kuinka hyvin ohjelmisto, järjestelmä, komponentti tai prosessi täyttää sille etukäteen määritellyt vaatimukset ja asiakkaan sekä käyttäjän asettamat tarpeet ja odotukset [IEE06]. Molemmat määritelmät painottavat vahvasti asiakkaan tarpeiden täyttämistä.

Laadulliset kriteerit on jaettu neljään osa-alueeseen: laatumalliin, ulkoi-

siin, sisäisiin ja käyttölaadullisiin metriikoihin (quality in use metrics) [ISO11]. Laatumalli luokittelee laadun jäsenneltynä joukkona piirteitä ja vaatimuksia, jonka kehyksiin organisaatio määrittelee ohjelmistoa varten laadulliset kriteerit. Ulkoiset metriikat vastaavat ohjelmiston toimintaa, kun taas sisäiset metriikat pohjautuvat ohjelmiston sisäisiin rakenteellisiin mittareihin.

Ulkoisia metriikoita voidaan mitata muun muassa julkaisun jälkeisten virheiden määrällä, eli kuinka paljon virheitä tuotteessa on. Sisäisillä metriikoilla ohjelmiston laatua arvioidaan taas koodimetriikoilla, kuten koodin monimutkaisuudella, riippuvuuksilla ja muilla vastaavilla tekijöillä. Sisäinen laatu tutkii olennaisesti koodin laatua. Käyttölaadullisuus voidaan arvioida vasta kun ohjelmisto on julkaistu käyttötarkoitustaan varten.

Metriikat tarjoavat yhden tehokkaan keinon ohjelmistojen laadun arviointiin. Staattisten ja dynaamisten virheenpaikannustekniikoiden johdosta virheiden laatu on muuttunut [ZN08]. Virheenpaikannustekniikoilla pyritään analysoimaan ohjelmiston lähdekoodia ja havaitsemaan siitä silmäänpistävimmät "kielioppivirheet". Tämän ansiosta suurin osa virheraportointijärjestelmiin tallennetuista raporteista, joilla kuvataan ohjelmistossa havaitut virheet, johtuvat pohjimmiltaan semanttisista eli loogisista ongelmista ohjelmiston koodissa [ZN08]. Nykyään metriikoiden tulee ottaa tämä huomioon.

2.1 Ohjelmiston muutoksien laadullinen vaikutus

Nagappan ja Ball esittävät suhteellisen koodikirnu-tekniikan järjestelmän virhetiheyden ennakoimiseen [NB05]. Koodikirnu (code churn) mittaa ja ilmaisee määrällisesti ohjelmiston komponentteihin kohdistuvia muutoksia tietyn ajanjakson aikana. Nagappan ja Ball tuovat esille joukon suhteellisia mittayksiköitä, jotka he rinnastavat muuttujiin kuten komponenttien kokoon tai muokkauksen ajalliseen pituuteen. Käyttäen apuna tilastollisia regressiomalleja, he osoittavat, että suhteelliset koodikirnu-mitat havaitsevat järjestelmän virhetiheyden paremmin kuin ehdottomat mitat. Tutkimuksessaan he suorittivat tapaustutkimuksen, jonka kohteena oli Windows Server 2003. Samalla he osoittavat, että suhteellinen koodikirnu pystyy paikallistamaan virheherkät komponentit 89 % tarkkuudella.

Monimutkaisuusmetriikoilla mitataan tyypillisesti ohjelmiston virhealttiutta [ZN08]. Metriikat ovat muodostettu esimerkiksi komponentin koodirivien, muuttujien ja metodien lukumäärästä. Niiden perimmäinen tarkoitus on arvioida kuinka monimutkainen jokin ohjelmiston komponentti on. Taustalla on yksinkertainen olettamus, että monimutkaisuus lisää ohjelmiston virheherkkyyttä [CK94, BBM96, NB05, NB07, ZN08, MNDT09]. Monimutkaisuusmetriikat keskittyvät kuitenkin harvoin komponenttien välisiin vuorovaikutussuhteisiin.

2.2 Ohjelmiston riippuvuuksien laadullinen vaikutus

Ohjelmiston komponentit riippuvat lähes aina toisista komponenteista, jolloin ne käyttävät niiden tarjoamia palveluita tuottaakseen oman toiminnallisuutensa. Järjestelmän riippuvuudet voidaan esittää verkkoina, joissa komponenttien keskinäiset suhteet paljastuvat [ZN08]. Niistä ilmenee mitä osia komponentit tarvitsevat toiminnalleen sekä mitkä osat tarvitsevat komponentin palveluita.

Zimmermann ja Nagappan esittävät verkkoanalyysin suorittamista komponenttien riippuvuusverkoille [ZN08]. Verkkoanalyysillä voidaan paikallistaa ohjelmiston kriittiset komponentit, jotka ovat muita virheherkempiä. Tämä tapahtuu tutkimalla verkkoa sekä kokonaisuutena että osina (aliverkot) erilaisten verkkometriikoiden pohjalta. Ohjelmistojen kohdalla komponentit muodostavat verkon toimijat ja komponenttien väliset riippuvuudet sekä niiden väliset vuorovaikutukset.

Zimmermann ja Nagappan vertasivat verkkoanalyysia ja monimutkaisuusmetriikoita Windows Server 2003:n laadun arvioimisessa. Verkkoanalyysillä saavutetaan heidän tapaustutkimuksensa tilastollisten analyysien mukaan 10 % parempi hyötyaste kuin pelkillä komponenttien monimutkaisuutta mittaavilla metriikoilla. Zimmermann ja Nagappan havaitsivat lisäksi, että verkkometriikat pystyvät identifioimaan 60 % komponenteista, joita ohjelmiston kehittäjät pitivät kriittisinä ohjelmiston kannalta.

2.3 Ohjelmiston testauksen merkitys

Ohjelmiston kehityksessä koodin testaaminen on kriittinen osa laadun takaamista. Testaamisessa ohjelmiston lähdekoodi alistetaan testitapauksille, joiden tarkoitus on kattaa ja varmistaa mahdollisimman hyvin kaikki loogiset tilanteet, jotka ohjelmisto käy läpi. Parhaassa tapauksessa testit löytävät virheet ohjelmistosta ja kehittäjät pystyvät korjaamaan ne ennen kuin tuote julkaistaan asiakkaalle. Näin ohjelmiston jatkokehitys helpottuu ja tuotteen käyttäjät säästyvät turhautumisilta.

Mockus, Nagappan ja Dinh-Trong tutkivat testien laadullista arviointia keinona havaita virheherkkiä komponentteja ohjelmistosta [MNDT09]. Testien analysoimisessa tulisi keskittyä nimenomaan niiden kykyyn havaita mahdollisia virheitä ohjelmistosta. Taitavat kehittäjät todennäköisesti tuottavat laadukkaampia testejä, mutta testien tehokkuuden ja laadun arvioiminen on järkevämpää toteuttaa automaattisesti. Yleisin testien tehokkuutta arvioiva mittari on testikattavuus.

Testikattavuuden lajeja on useita. Yksinkertaisista luokka-, funktio-, metodi- ja käskykattavuuksista kehittyneisiin haara- ja polkukattavuuksiin. Nimensä mukaan kukin laji testaa lähdekoodin eri osa-alueita. Funktio- ja metodikattavuudella kartoitetaan testien kattavuutta yhden toiminnallisuuden osalta, luokkakattavuudella taas näistä muodostuvan kokonaisuuden testien

kattavuutta. Taustalla on olettamus, että jos jokin yksittäinen looginen ehto tai polku ei ole katettu vähintään yhdellä testillä, ei sen mahdollisesti sisältämiä virheitä pystytä havaitsemaan [MNDT09].

Voidaankin olettaa, että suurempi testikattavuus löytää todennäköisesti enemmän virheitä ja takaa näin ollen paremman laadun. Kattavuus kuvaa yksinkertaisesti osuutta siitä kuinka monta riviä ohjelmakoodia on katettu sitä testaavalla testikoodilla. Kattavuudella ei kuitenkaan pystytä arvioimaan kuinka todennäköisesti nämä rivit aiheuttavat virheen, siksi suuri testikattavuus ei yksinään takaa laatua. Testikattavuus saattaa vääristyä helposti testeillä, jotka eivät todellisuudessa tarkista ohjelmiston koodin varsinaista toimintaa. Testikattavuus auttaa kuitenkin merkittävästi laadun takaamisessa. Muita metriikoita tulisi käyttää kohdentamaan testejä ohjelmiston kriittisiin osa-alueisiin [NB07, MNDT09, YH11].

Testien laadun arvioimiseen on ehdotettu mutaatiotestausta [YH11]. Mutaatiotestaus arvioi testien sopivuutta niiden kattamaan lähdekoodiin simuloimalla yleisempiä virheitä joita kehittäjät tekevät. Alkuperäisen ohjelmiston lähdekoodin syntaksia muuttamalla se tutkii testien laatua ja tehokkuutta havaita mahdollisia virheitä ohjelmiston lähdekoodissa. Yksinkertaisten syntaksimuutosten avulla mutaatiotestaus pystyy muodostamaan virheellisiä mutanttiversioita ohjelmistosta, joiden ei pitäisi mennä testeistä läpi.

2.4 Ohjelmiston koodin kehittäjänä ihminen

Laadun takeeksi ei voida luetella pelkästään mekaanisia laatua arvioivia metriikoita. Kehittäjän käytänteillä on suuri laadullinen merkitys ohjelman kaikissa kehitysvaiheissa, joten ohjelmistotuotantomenetelmät nousevat suureen rooliin. Niiden tulisi ohjata laadukasta kehitystä.

Vanhojen raskaaseen ennakkosuunniteluun pohjautuvien tuotantomenetelmien, kuten vesiputousmallin, rinnalle on noussut uusia ketterän kehityksen malleja. Ne painottavat yksilöitä ja yksilöiden vuorovaikutusta, toimivan ohjelmiston merkitystä, asiakkaan merkitystä kehitysprosessin kriittisenä osana ja muutoksiin sopeutuvaa kehitystä [BBvB+01]. Ketterässä kehityksessä ohjelmisto tuotetaan iteratiivisesti, pala kerrallaan, sopeutuen uusiin tavoitteisiin. Vesiputousmallissa, jossa ohjelmisto suunnitellaan tiukasti ennen toteutusta, lopputuotokset eivät yleensä vastaa haluttua tulosta, varsinkaan asiakkaan kannalta. Useat empiiriset tutkimukset tukevat ketterien kehitysmallien hyötyä merkittävänä laadullisina vaikuttajana [SS10].

3 Metriikat

Metriikat tarjoavat yhden tehokkaan keinon ohjelmistojen laadun arviointiin. Metriikoita on lukuisia, näistä muutamia pinnalla olevia ovat koodikirnu, verkkoanalyysi, testikattavuus ja mutaatiotestaus. Koodikirnulla arvioidaan

ohjelmiston muutoksien vaikutusta ohjelmiston virheherkkyyteen, verkkoanalyysillä tutkitaan ohjelmiston komponenttien riippuvuuksien vaikutusta ohjelmiston virhealttiuteen ja testikattavuudella sekä mutaatiotestauksella analysoidaan ohjelmiston lähdekoodin testien tehokkuutta ja laadukkuutta.

3.1 Perinteiset metriikat

Metriikoiden käyttäminen ohjelmiston virheherkkyyden ja laadun arvioimisessa ei ole mikään uusi käytäntö. Metriikoita on ehdotettu ja tutkittu vuosikymmenien ajan. Perinteisimmät mittarit pohjautuvat yksinkertaisesti ohjelmiston lähdekoodin koodirivien määrään [BBM96]. Olio-ohjelmoinnin noustessa vahvempaan suosioon 90-luvulla, ryhtyivät tutkijat pohtimaan mittoja jotka soveltuisivat kyseiseen ajatusmalliin.

Vuonna 1994 Chidamber ja Kemerer ehdottivat kuusi olio-ohjelmointimittaria ohjelmiston laadun arvioimiseen [CK94]. Nämä CK-metriikat keskittyvät metodien, yliluokkien, ja lapsien lukumääriin sekä tutkivat komponenttien riippuvuuksia, vastuita ja yhtenäisyyttä. Metodien määrällä kuvataan muun muassa komponentin monimutkaisuutta. Valta osa kyseisistä mitoista on havaittu tehokkaiksi käytännöllisiksi arvioiksi ohjelmiston komponenttien virhealttiuden mittaamisessa [BBM96].

Ajatuksena on, että mitä enemmän luokalla on metodeja sen monimutkaisempi se on [BBM96]. Luokalla, jolla on taas suuri määrä yliluokkia lisää riskiä, että jokin näistä yliluokista aiheuttaa ongelmia kyseiselle luokalle. Vastaavasti luokka, jolla on paljon lapsia, vaikeuttaa luokalle tehtäviä muutoksia niin, että aliluokkien toiminnallisuus säilyy ehjänä. Sen lisäksi tiukat kytkökset luokkien välillä ja luokan suurempi vastuu lisäävät komponenttien virhealttiutta.

CK-metriikat muodostavat pohjan lukuisille metriikoilla. Luokkien riippuvuudet ovat keskeisessä asemassa muun muassa koodikirnussa ja verkkoanalyysissä [NB05, NB07, ZN08]. Muutokset ohjelmiston lähdekoodiin laajenevat yleensä komponenttien riippuvuuksia pitkin [NB05, NB07].

3.2 Koodikirnu

Nagappan ja Ball esittävät ohjelmistojen virhetiheyden arvioimiseen ratkaisuksi koodikirnua [NB05]. Se mittaa ohjelmiston komponenttien ohjelmakoodiin kohdistuvien muutosten määrää tietyn ajanjakson aikana. Muutosten määrä on helposti saatavilla ohjelmiston versionhallintajärjestelmien muutoshistoriasta. Useimmat versionhallintajärjestelmät vertailevat lähdekooditiedostojen historiaa ja laskevat automaattisesti koodiin kohdistuvia muutoksia. Nämä muutokset ilmentävät kuinka monta riviä tiedostoon on ohjelmoijan toimesta lisätty, poistettu tai muutettu sitten viimeiseen versiohistoriaan tallennetun version. Nämä muutokset muodostavat koodikirnun pohjan.

Nagappan ja Ball esittävät joukon suhteellisia koodikirnu-mittoja virhetiheyden havaitsemiseen. Mitat ovat normalisoituja arvoja koodikirnun aikana saaduista tuloksista. Normalisoinnilla niistä on pyritty poistamaan mahdolliset häiriöt. Näitä mittoja on muun muassa yhteenlaskettujen koodirivien määrä, tiedostojen muutokset ja tiedostojen määrä. Tutkimukset ovat osoittaneet, että ehdottomat mittayksiköt, kuten pelkkä koodirivien summa, ovat huonoja ohjelmiston laadullisia ennusteita. Yleisesti ottaen ohjelmiston kehitysprosessia mittaavien yksiköiden on havaittu olevan parempia osoittimia vikojen määrästä kuin pelkkää koodia arvioivat kriteerit.

Ohjelmistoa kehitettäessä sen komponenttien monimutkaisuus muuttuu. Monimutkaisuuden kasvun suhde on hyvä mittari virheherkkyyden kasvulle. Koodikirnu-mittojen on havaittu korreloivan näkyvästi ohjelmistoista tehtyjen vikailmoitusten kanssa. Mittojen välillä on havaittavissa lisäksi keskinäisiä suhteita, joita voidaan mallintaa verkkoina. Yksinään kyseiset mittarit eivät välttämättä tuota toivottua tulosta. Näin ollen mittoja verrataan keskenään mahdollisten ristiriitaisuuksien havaitsemiseksi. Johtopäätöksiin päätyminen on hankalaa empiirissä tutkimuksissa, koska prosessien taustalla on usein laajoja kontekstisidonnaisia tekijöitä.

Ohjelmiston virheherkkyyteen vaikuttavat mitat

Nagappan ja Ball listaavat seuraavat ehdottomat mitat koodikirnun pohjaksi. Nämä muodostavat suhteellisille mitoille vertailukohdat ohjelmiston virheherkkyyden analysoimisessa. Ehdottomat mitat eivät yksinään tuota luotettavaa tulosta.

Yhteenlaskettu koodirivien määrä, ohjelman uuden version ei-kommentoitujen koodirivien summa kaikkien lähdekooditiedostojen kesken.

Käsiteltyjen koodirivien määrä, ohjelman lähdekoodiin lisättyjen ja muuttuneiden koodirivien summa edelliseen versioon nähden.

Poistettujen koodirivien määrä, ohjelman lähdekoodista poistettujen koodirivien määrä edelliseen versioon nähden.

Tiedostojen määrä, yhden ohjelman kääntämiseen tarvittavien lähdekooditiedostojen määrä.

Muutosten ajanjakso, yhteen tiedostoon kohdistuneiden muutosten ajanjakson pituus.

Muutosten määrä, ohjelman tiedostoihin kohdistuneiden muutosten määrä edelliseen versioon nähden.

Käsiteltyjen tiedostojen määrä, ohjelman käsiteltyjen tiedostojen yhteenlaskettu määrä.

Näiden pohjalta he muodostivat kahdeksan suhteellista koodikirnu-mittaa ja osoittivat, että nämä mitat korreloivat kohonneeseen virhemäärään koodirivejä kohden. He käyttivät Spearmanin järjestyskorrelaatiokerrointa, joka kuvaa kahden asian keskinäistä vastaavuutta. Analyysissään he havaitsivat suhteellisten mittojen ylivertaisuuden ehdottomiin verrattuna. Empiiristen tutkimusten avulla he havaitsivat seuraavien mittojen soveltuvuuden todellisen virhetiheyden ennakoimiseen.

1. **Käsiteltyjen koodirivien määrä / Yhteenlaskettu koodirivien määrä**

   Suurempi osa käsiteltyjä koodirivejä suhteessa yhteenlaskettuun koodirivien määrän vaikuttaa yksittäisen ohjelman virhetiheyteen.

2. **Poistettujen koodirivien määrä / Yhteenlaskettu koodirivien määrä**

   Suurempi osa poistettuja koodirivejä suhteessa yhteenlaskettuun koodirivien määrään vaikuttaa yksittäisen ohjelman virhetiheyteen. Nagappan ja Ball havaitsivat korkean korrelaation mittojen 1. ja 2. välillä.

3. **Käsiteltyjen tiedostojen määrä / Tiedostojen määrä**

   Suurempi osa käsiteltyjä tiedostoja suhteessa ohjelman rakentavien tiedostojen lukumäärään lisää todennäköisyyttä, että nämä käsitellyt tiedostot aiheuttavat uusia vikoja. Esimerkiksi meillä on kaksi ohjelmaa A ja B, jotka molemmat koostuvat 20 lähdekooditiedostosta. A sisältää viisi käsiteltyä tiedostoa ja B kaksi. Todennäköisyys sille, että muutokset ohjelmaan A saattavat aiheuttaa uusia vikoja on suurempi.

4. **Muutosten määrä / Käsiteltyjen tiedostojen määrä**

   Mitä suurempi määrä yksittäisiin tiedostoihin on kohdistunut muutoksia, sitä suurempi on todennäköisyys sille, että tämä vaikuttaa kyseisistä lähdekooditiedostoista muodostuvan ohjelman virhetiheyteen. Esimerkiksi jos ohjelman A viittä lähdekooditiedostoa on muutettu 20 kertaa ja ohjelman B viittä tiedostoja on muutettu kymmenen kertaa, todennäköisyys sille, että ohjelma A sisältää uusia vikoja on suurempi.

5. **Muutosten ajanjakso / Tiedostojen määrä**

   Mitä pitempi aika on kulutettu muutoksiin, jotka kohdistuvat pieneen joukkoon tiedostoja, sitä suurempi on todennäköisyys sillä, että nämä tiedostot sisältävät monimutkaisia rakenteita. Monimutkaisuus vaikuttaa koodin helppoon ylläpidettävyyteen ja näin ollen lisää näiden tiedostojen aiheuttamaa virhetiheyttä.

6. **Käsiteltyjen ja poistettujen koodirivien määrä / Muutosten ajanjakso**

   Käsiteltyjen ja poistettujen koodirivien määrä suhteessa muutosten ajanjaksoon mittaa muutoksen määrää, jota pelkkä muutosten ajanjakso ei yksinään ilmaise. Tätä mittaa tulee verrata mittaan 5. Oletuksena on, että mitä suurempi määrä käsiteltyjä ja poistettuja koodirivejä on, sitä pitempi muutosten ajanjakson tulisi olla. Tämä taas vaikuttaa ohjelman virhetiheyteen.

7. **Käsiteltyjen koodirivien määrä / Poistettujen koodirivien määrä**

   Ohjelmiston kehitys ei koostu pelkästään vikojen korjaamisesta vaan jatkuvasta uuden kehittämisestä. Uusien ominaisuuksien kehittämisessä käsiteltyjen koodirivien määrä on suhteessa suurempi kuin poistettujen koodirivien määrä. Suuri arvo tälle mitalle ilmaisee uutta kehitystä. Saatua arvoa verrataan mittoihin 1. ja 2., jotka yksinään eivät ennakoi uutta kehitystä.

8. **Käsiteltyjen ja poistettujen koodirivien määrä / Muutosten määrä**

   Mitä suurempi muutoksen laajuus on suhteessa muutosten määrään, sitä suurempi virhetiheys on. Mitta 8. toimii verrokkina mitoille 3. – 6. Suhteessa mittoihin 3. ja 4., mitta 8. ilmaisee todellisen muutoksen määrää. Se kompensoi sitä tietoa, että yksittäisiä tiedostoja ei käsitellä toistuvasti pienten korjausten takia. Suhteessa mittoihin 5. ja 6.,mitä suurempi käsiteltyjen ja poistettujen koodirivien määrä on käsittelyä kohden, sitä pitempi muutosten ajanjakso tarvitaan ja sitä enemmän muutoksia kohdistuu esimerkiksi jokaista viikkoa kohden. Muussa tapauksessa suuri määrä muutoksia on saattanut kohdistua hyvin lyhyeen ajanjaksoon, joka ennakoi suurempaa virhetiheyttä.

Johtopäätökset koodikirnusta

Nagappan ja Ball havaitsivat, että koodi joka muuttuu useasti ennen julkaisua on virheherkempää kuin koodi, joka muuttuu vähemmän saman ajanjakson aikana. He tutkivat kahden ohjelmistojulkaisun Windows Server 2003 ja Windows Server 2003 Service Pack 1 pohjalta saatuja tuloksia. Julkaisuista analysoitiin 44,97 miljoonaa riviä koodia, joka muodostui 96 189 lähdekooditiedostosta. Niistä käännettiin 2 465 yksittäistä ohjelmaa.

He päätyivät tutkimuksessaan neljään johtopäätökseen:

1. Suhteellisten koodikirnu-mittojen nousua seuraa ohjelmiston virheherkkyyden kasvu.

2. Suhteelliset mitat ovat parempia laadullisia arvioijia kuin ehdottomat mitat.

3. Suhteellinen koodikirnu on tehokas tapa arvioida ohjelmiston virheherkkyyttä.

4. Suhteellinen koodikirnu pystyy havaitsemaan virheherkän ja toimivan komponentin toisistaan.

Koodikirnun pätevyyteen vaikuttavia tekijöitä

Nagappan ja Ball toteavat, että mittausvirheet vaikuttavat arvion luomiseen. Ongelma ei ole kuitenkaan suuri, sillä versionhallintajärjestelmät hoitavat automaattisesti analyysiin vaadittavat lähtöarvot. Koodikirnu vaatii kuitenkin ohjelmiston kehittäjältä hyviä käytäntöjä. Jos kehittäjä on tehnyt useita muutoksia rekisteröimättä niitä versionhallintajärjestelmän historiaan, osa muutoksista jää näkemättä. Kehittäjän toimista riippuen muutosten ajanjakson pituus voi merkittävästi pidentyä, jos muutoksia ei hyväksytä tarpeeksi aikaisin versionhallintajärjestelmään. Mittojen vertaaminen keskenään lieventää tästä johtuvia poikkeamia.

He toteavat lisäksi, että tapaustutkimuksen pätevyyteen voidaan nähdä vaikuttavan se, että tutkimuksessa analysoitiin vain yhtä ohjelmistojärjestelmää. Tämä ohjelmistojärjestelmä kuitenkin koostuu lukuisista komponenteista ja suuresta määrästä koodia. Analyysi on itsessään erittäin kattava.

3.3 Verkkoanalyysi

Ohjelmiston komponentit riippuvat lähes aina toisista komponenteista. Järjestelmän riippuvuudet voidaankin esittää matalan tason verkkoina, jossa komponenttien keskinäiset suhteet paljastuvat. Zimmermann ja Nagappan esittävät verkkoanalyysin suorittamista riippuvuusverkoille ohjelmiston virheherkkyyden arvioimiseksi [ZN08]. Verkkoanalyysillä voidaan paikallistaa ohjelmiston kriittiset komponentit, jotka ovat oletettavasti muita virheherkempiä.

Zimmermann ja Nagappan tutkivat julkaisun jälkeisten virheilmoitusten ja riippuvuusverkkojen suhdetta ja havaitsivat kaksi olennaista vaikuttajaa. Keskeisessä roolissa olevat komponentit sekä yksittäiset komponentit, joilla on suuri määrä keskinäisiä riippuvuuksia, ovat yleisesti herkempiä virheille.

Riippuvuus ohjelmistoissa on suunnattu yhteys kahden koodiosan kuten lausekkeen tai metodin välillä. Riippuvuudet voidaan erotella toisistaan: tietoriippuvuus on määrittelyiden ja arvojen välinen yhteys ja kutsuriippuvuus on funktio- ja metodimäärittelyiden ja niitä kutsuvien paikkojen välinen yhteys.

Zimmermann ja Nagappan löysivät useita piirteitä verkoista, joilla keskeisessä roolissa olevat komponentit voidaan havaita samankaltaisista aliver-

koista. Yksi on niin kutsuttu tähtipiirre (star pattern), joka on komponenteilla joilla on useita satelliittikomponentteja. Satelliitit ympäröivät tähteään ja riippuvat yksinään siitä. Suurimmassa osassa tämän piirteen omaavista aliverkoista tähtikomponentti oli virheherkkä kun taas satelliitit eivät. Verkkoanalyysin mukaan tähtikomponentti on keskeinen komponentti, jos se ohjaa satelliittejaan. Tämänkaltaista tähtikomponenttia kutsutaan usein välittäjäksi.

Mitä suurempi joukko verkossa olevia komponentteja riippuu keskenään toisistaan (clique), sitä suurempi on näiden komponenttien todennäköinen virheherkkyys. Riippuvuuden suunnalla ei tässä tapauksessa ole väliä. X komponentti voi riippua Y:stä, Y komponentti X:stä tai molemmat yhtäaikaisesti toisistaan. Joukkoa kutsutaan maksimaaliseksi, jos yhtään komponenttia ei voida lisätä tähän aliverkkoon siten, että maksimaalisuus ei säilyisi.

Zimmermann ja Nagappan kävivät läpi maksimaaliset joukot Windows Server 2003:n riippuvuusverkosta ja järjestivät nämä joukon koon mukaan. Virheilmoituksia vertaamalla he havaitsivat, että mitä enemmän riippuvuuksia joukossa on, sitä enemmän virheilmoituksia korreloi näihin komponentteihin. He toteavat yhdeksi syyksi sen, että nämä joukot ovat tämän tiedon valossa muita monimutkaisempia riippuvuuksiensa johdosta.

Aikaisempien tutkimusten mukaan on havaittu, että koodikirnu ja riippuvuusverkot ovat yhdessä hyviä metriikoita arvioida ohjelman virheherkkyyttä [NB07]. Jos komponentti B muuttuu paljon eri versioiden välillä, voidaan olettaa, että komponentin A täytyy muuttua, jotta muutokset B:hen ovat mahdollisia. Muutos yleensä leviää riippuvuuksien välillä.

Riippuvuusverkossa näkyvät yhteydet ilmentävät kuinka paljon työtä tarvitaan yhteyksien ylläpitämiseen. Muutosten määrän lisäksi yhteydet voivat kertoa virheherkkyydestä muutakin tärkeää tietoa. Lähdekoodi ei muodosta pelkästään yksittäisistä komponenteista vaan arkkitehtuurista josta koko ohjelmisto koostuu. Näitä asioita arvioimalla pystytään paikallistamaan muita virheherkempiä komponentteja ja kohdistamaan fokus testejä ja koodikatselmusta varten.

Ohjelmiston virheherkkyyteen vaikuttavat verkkomitat

Zimmermann ja Nagappan suorittivat verkkoanalyysin Windows Server 2003:n riippuvuusverkolle. He keräsivät samalla lukuisia monimutkaisuusmetriikoita ja vertasivat niitä verkkoanalyysiin. Tapaustutkimuksessaan he keskittyivät pelkästään riippuvuuksien esiintymiin. Kahden komponentin keskinäisten riippuvuuksien määrä jätettiin huomioimatta, toisin sanoen tutkimuksessa ei huomioitu jos esimerkiksi A komponentti riippuu kolme kertaa B:stä. Spearmanin ja Pearsonin järjestyskorrelaatiokertoimien avulla Zimmermann ja Nagappan havaitsivat seuraavien verkkomittojen korreloivan merkittävästi tai positiivisesti julkaisun jälkeen ilmoitettuihin virheisiin.

1. **Egoverkot**

   Jokaisella solmulla, eli komponentilla, on verkossa sitä vastaava egoverkko, joka kuvaa miten kyseinen solmu on kytketty naapurisolmuihinsa. Komponentteja, jotka riippuvat solmusta kutsutaan sisäsolmuiksi, ja komponentteja joista solmu itse riippuu kutsutaan ulkosolmuiksi. Egoverkko on sisä- ja ulkosolmujen muodostama aliverkko. Se mahdollistaa komponentin paikallisen tärkeyden mittaamista suhteessa naapureihinsa.

   Egoverkolle voidaan suorittaa useita mittauksia. Tehokkaimpia metriikoita olivat tutkimuksen mukaan mittarit, jotka kohdistuvat muun muassa egoverkon kokoon, komponenttien siteisiin, pareihin ja tiheyteen sekä verkon, eli riippuvuuksien, polkuihin. Zimmermann ja Nagappan havaitsivat tutkimuksessaan, että ulkosolmut ovat sisäsolmuja virheherkempiä. Toisin sanoen komponentit, joista muut komponentit riippuvat, ovat virheherkempiä.

2. **Globaaliverkot**

   Globaaliverkko muodostuu koko ohjelmiston riippuvuusverkosta ja sen muodostavien komponenttien välisistä riippuvuuksista. Globaaliverkosta voidaan tutkia yksittäisen solmun tärkeyttä koko ohjelmiston kannalta ja näin havaita koko ohjelmiston kriittisimmät komponentit. Keskeisessä roolissa olevat komponentit ovat tutkimuksen mukaan muita virheherkempiä.

3. **Rakenteelliset puutteet**

   Ideaalisesti solmujen väliset riippuvuudet ovat toisiinsa nähden tasapainossa. Mikäli meillä on solmut A, B ja C ja kaikilla on keskinäinen suhde toisiinsa, vallitsee solmujen välillä tasapaino. Jos kuitenkin esimerkiksi solmujen B ja C välillä ei ole keskinäistä riippuvuutta, vaan ne riippuvat toisistaan A:n kautta, solmulla A on selvä etulyöntiasema. A toimii välittäjänä B:n ja C:n välillä ja näin ollen solmut eivät ole tasapainossa. Välittäjäsolmut ovat Zimmermannin ja Nagappanin tutkimuksen mukaan muita virheherkempiä.

4. **Keskeisyys**

   Yleisin verkkomitoista on solmun keskeisyys. Sillä pyritään havaitsemaan komponentit jotka ovat suotuisassa asemassa, eli useat muut komponentit riippuvat kyseisestä komponentista. Keskeisyyttä voidaan mitata riippuvuuksien määrällä, komponenttien riippuvuuksien etäisyyksillä toisistaan ja komponentista johtavien riippuvuuspolkujen piirteillä. Mikäli komponentti riippuu hyvin suuresta määrästä toisia komponentteja, on se todennäköisesti muita komponentteja virheherkempi.

Komponentit, joiden riippuvuuksien välillä on hyvin lyhyet polut, voidaan todeta olevan muita virhealttiimpia. Muutokset näihin komponentteihin yleensä leviävät komponenttien riippuvuuksiin.

Keskeisyydellä on tutkimuksen mukaan vastakkainen vaikutus. Keskeisyys saattaa tehdä komponenteista vähemmän virheherkkiä. Oletettavasti näihin komponentteihin on kehitystyössä luultavasti panostettu enemmän, joka parantaa niiden laatua.

Johtopäätökset verkkoanalyysistä

Windows Server 2003:n kehittäjätiimit pitivät listaa ohjelmistojärjestelmän kriittisistä komponenteista. Kehittäjät valitsevat käsin nämä komponentit niihin liittyvän historian perusteella. Arvioon vaikuttuvat muun muassa komponentteihin kohdistuneet muutokset ja virheiden määrä. Kriittisiä komponentteja tulee varjella muita tarkemmin. Jos esimerkiksi Windowsin ytimeen tehdään muutoksia, täytyy muutokset varmentaa kattavien testausten ja tarkastelujen avulla.

Zimmermann ja Nagappan vertaisivat verkkoanalyysin havaitsemia komponentteja kehittäjien laatimaan kriittisten komponenttien listaan. Monimutkaisuusmetriikat löysivät kriittistä komponenteista vain 30 %, kun verkkoanalyysillä saavutettiin kaksi kertaa parempi tulos.

Yksittäisissä tapauksissa monimutkaisuusmetriikat olivat toisaalta hieman parempia kuin verkkometriikat. Olio-ohjelmointiin liittyvät monimutkaisuusmetriikat eivät kuitenkaan sovellu epäyhtenäisten ohjelmistojen arvioimiseen. Ohjelmiston täytyy noudattaa täysin olio-paradigmaa, jotta metriikoilla olisi merkitystä.

Zimmermann ja Nagappan pitävät verkkoanalyysin suorittamista ohjelmiston komponenttien riippuvuusverkoille tehokkaana tapana arvioida ohjelman virheherkkyyttä.

He päätyivät tutkimuksessaan kolmeen johtopäätökseen:

1. Verkkometriikat riippuvuusverkoissa pystyvät löytämään kriittisiä komponentteja, joita pelkät monimutkaisuusmetriikat eivät havaitse.
2. Verkkometriikoiden antamat arvot riippuvuusverkoissa korreloivat julkaisun jälkeen ilmoitettujen virheiden kanssa. Suurempaa arvoa johtaa todennäköisesti suurempi virhetiheys.
3. Verkkometriikat riippuverkoissa pystyvät arvioimaan julkaisun jälkeisten virheiden määrää.

Verkkoanalyysin pätevyyteen vaikuttavia tekijöitä

Zimmermann ja Nagappan olettivat tutkimuksessaan, että virheet ja niiden korjaukset sijoittuvat lähdekoodissa samaan paikkaan. He kuitenkin

toteavat, että näin ei aina ole, mutta tämä olettamus on yleisesti käytössä tutkimuksissa.

Yleisesti pätevien päätelmien tekeminen empiirisistä tutkimuksista on vaikeaa niiden kontekstisidonnaisen luonteen takia. Virheherkkyyden arvioimiseen ei ole löytynyt yksittäistä ”parasta” ratkaisua, joten verkkoanalyysin tuottamia tuloksia ei pystytty vertailemaan sellaisen tehokkuuteen. Tulokset antavat kuitenkin lupaavia viitteitä.

Zimmermann ja Nagappan toteavat, että tapaustutkimukseen voi vaikuttaa se, että tutkimuksessa analysoitiin vain yhtä ohjelmistojärjestelmää. Tämä järjestelmä on kuitenkin kooltaan suurempi kuin useat muut kaupalliset ohjelmistot. Tästä johtuen verkkoanalyysin pitäisi soveltua virheherkkyyden arvioimiseen muissa ohjelmistoissa.

Verkkoanalyysi ei todennäköisesti sovellu yksinään virheherkkyyden arvioimiseen. Sen voidaan kuitenkin nähdä olevan osa palapeliä. Tutkimuksessa esille tulleet monimutkaisuusmetriikat ja koodikirnu ovat Zimmermannin ja Nagappanin mukaan varteenotettavia lisiä verkkoanalyysin tehokkuudelle. Mikään kyseisistä metriikoista ei kuitenkaan ota huomioon virheiden inhimillistä tekijää. Kehittäjät loppujen lopuksi aiheuttavat virheet itse kehitystyön tuloksena.

3.4 Testikattavuus

Mockus, Nagappan ja Dinh-Trong tutkivat testien laadullista arviointia keinona havaita virheherkkiä komponentteja [MNDT09]. Testien arvioimisessa tulisi nimenomaan keskittyä testien kykyyn havaita mahdollisia virheitä ohjelmistosta. Parempien testien tulisi löytää enemmän mahdollisia ongelmakohtia ohjelmiston lähdekoodista ja näin johtaa ohjelmiston parempaa laatuun.

Mockus ym. tekivät tapaustutkimuksen kahden eri organisaation hyvin erilaisista ohjelmistosta: Microsoft Windows Vista:sta ja Avaya:sta. Ensimmäinen ohjelmisto on toteutettu C ja C++ kielillä, toinen Javalla. Windows Vistan testaamiseen käytettiin Microsoftin sisällä kehitettyä työkalua ja Avayan testaamiseen JUnit-testiympäristöä. Testikattavuus keskittyi eri ohjelmointikielistä johtuen näiden kielten eri piirteisiin.

Mockus ym. havaitsivat tutkimuksessaan, että molempien ohjelmistojen osalta suurempaa testikattavuutta seurasi pienempi määrä julkaisun jälkeisiä virheilmoituksia. He huomasivat, että suuremman testikattavuuden saavuttaminen kasvaa eksponentiaalisesti, mitä suurempiin testikattavuuksiin tähdätään. Samalla virheherkkyys vähenee vain lineaarisesti.

Optimaalinen testikattavuus ei tunnu olevan lähelläkään 100 %, eikä sen saavuttaminen ole ohjelmiston laadun kannalta välttämätöntä, saati tehokasta. He käyttivät tutkimuksessaan Spearmanin järjestyskorrelaatiokerrointa testikattavuuden ja julkaisun jälkeisten virheiden korrelaation selvittämiseen.

Testikattavuuden kannalta olisi mielenkiintoista tietää kuinka suuri osa

itse testeistä havaitsi virheet. Suurin osa tästä työstä tapahtuu kuitenkin kehittäjän toimesta kehitysvaiheessa yksikkötestauksen tasolla. Viitteet tästä eivät tallennu versionhallintaan eikä virheitä ilmoiteta virheraportointijärjestelmiin. Tämän korrelaation tutkiminen on valitettavan haasteellista.

Mockus ym. havaitsivat tutkimuksessaan, että lähdekoodin monimutkaisuus, ohjelmiston käyttökohde, kehittäjien kokemus ja etätyöskentely vaikuttavat ohjelmiston virheherkkyyteen sekä testien kattavuuteen. Vähemmän kokeneilla kehittäjillä ja etätyöskentelyssä testikattavuuden merkitys kasvaa merkittävästi.

Testikattavuuden vaikutus ohjelmiston virheherkkyyteen

Mockus ym. tutkimus analysoi versionhallinta- ja virheraportointijärjestelmistä sekä testikattavuudesta saatavia tietoja arvioidakseen kunkin osa-alueen vaikutusta ohjelmiston virheherkkyyteen. Samalla ohjelmistoista laskettiin monimutkaisuusmetriikoita ja tutkittiin lähdekoodin muutosten määrää. Aikaisempien tutkimusten mukaan muutosten määrä on varteenotettava mittari virheherkkyyden arvioimiseen, siksi testikattavuus on syytä suhteuttaa siihen. Tilastollisten analyysien lisäksi he haastattelivat kehittäjiä tulosten vahvistamiseksi.

Avaya -tapauksessa Mockus ym. havaitsivat näkyvän korrelaation testikattavuuden kasvulla ja julkaisun jälkeisten virheilmoitusten vähenemisellä. Lähdekooditiedostoja joita testit eivät kattaneet, löytyi eniten virheilmoituksia. Vastaavasti tiedostoille, joiden testikattavuus oli vähintään 50 %, virheiden määrä oli pienempi. Testikattavuuden teho kuitenkin ryhtyy vähenemään jo 60 % kattavuuden kohdalla.

Versionhallintajärjestelmistä pystytään analysoimaan testikattavuuden saavuttamiseen käytetty aika, eli kuinka kauan kehittäjän on täytynyt käyttää yksittäisen komponentin automaattiseen testaamiseen. Kattavimpiin testeihin kuluu eksponentiaalisesti pidempi aika mitä korkeampia testikattavuuksia tavoitellaan. Tämä indikoi, että täyden testikattavuuden tavoittaminen ei kaikissa tapauksista ole välttämättä hyödyllistä. Tämä voi johtua siitä, että tiedostot jotka muuttuvat eniten ovat testattu kattavammin, sillä muutokset helposti tuovat uusia virheitä. 50 % testikattavuuden saavuttaminen näyttää tulosten pohjalta olevan suhteellisen helppoa. Mockus ym. vertasivat Avayasta saatuja tuloksia ja havaitsivat, että samat johtopäätökset voitiin tehdä Windows Vistan tapauksessa.

Tilastollisen analysoinnin tulosten tueksi Mockus ym. suorittivat haastattelun tuotteiden kehittäjätiimeissä. Kehittäjät, jotka kirjoittavat lähdekoodin alusta asti, testaavat yleensä koodin kattavammin. Kehittäjät, jotka vain ylläpitävät toisten kirjoittamaa koodia, harvoin kasvattavat koodin testikattavuutta. Loogisesti monimutkaiset ja helpot lähdekooditiedostot testataan usein muita kattavammin. Komponentit, jotka tarjoavat palveluita useille muille komponenteille testataan haastattelun pohjalta kattavammin.

Keskeisessä roolissa olevista komponenteista saatetaan toisaalta löytää enemmän virheitä puhtaasti sen takia, että niitä käytetään useammin. Nämä komponentit joutuvat tiukempiin käytännöntilanteisiin liittyviin testauksiin. Testikattavuuden saavuttaminen on heikompaa käyttöliittymään ja tietokantaan liittyvissä koodiosuuksissa, koska näiden testaamista pidetään kehittäjien keskuudessa muita haastavampana. Haastattelussa havaittiin, että etätyöskentely tuntuu vähentävän testauksen määrää.

Johtopäätökset testikattavuudesta

Testikattavuuden lähtökohtana on se, että lähdekoodin virheitä ei pystytä havaitsemaan ellei kyseisiä rivejä testata vähintään yhdellä testillä. Testikattavuuden ja laadun välistä yhteyttä on tutkittu yllättävän vähän, varsinkaan laadullisista näkökulmista. Testikattavuutta tulisi ohjata komponenttien tärkeysjärjestyksen pohjalta. Olennaisesti kriittisempiä osia tulisi painottaa testeissä, unohtamatta siltikään pienemmissä osissa olevia osia.

Ongelmana on kuitenkin se, että vakavimmat virheet voidaan havaita jo hyvin pienellä testikattavuudella. Vaikka jokin yksittäinen rivi lähdekoodista olisi katettu testeillä, ei se takaa sitä, että tämä testi pystyisi havaitsemaan kyseisen rivin mahdollisesti aiheuttamia virheitä. On kuitenkin kohtuullista odottaa, että suurempi testikattavuus lisää todennäköisyyttä, että nämäkin tilanteet tulevat katettua.

Eri ohjelmistot kehitetään lähtökohtaisesti eri tarkoituksiin, eri kehittäjien ja testaajien toimesta. Testikattavuus on tärkeä suhteellistaa näihin olosuhteisiin. Kontekstit ovat harvemmin samoja. Kokeneet kehittäjät kirjoittavat yleensä laadullisesti parempia testejä, koska kokemuksen karttuminen kasvattaa testikattavuutta ja näin ollen vähentää ohjelmiston virhetiheyttä. Inhimillisillä tekijöillä on valtava merkitys ohjelmiston laadullisissa tekijöissä.

Mockus ym. toteavat testikattavuuden olevan käytännöllinen ja järkevä keino mitata ja varmistaa ohjelmiston laatua. Valitettavasti täydellisen testikattavuuden saavuttaminen ei ole todennäköisesti järkevää, sillä sen lopullinen hyödyllisyys näyttää olevan negatiivinen ja samalla suurempien testikattavuuksien saavuttaminen haasteellista.

Mockus ym. päätyvät seuraaviin neljään johtopäätökseen testikattavuudesta:

1. Ohjelmiston hyväksymäkriteereihin tulisi kuulua 70 % testikattavuus.
2. Yli 70 % testikattavuus ei yleensä ole tehokkuuden kannalta järkevää. Suuremman testikattavuuden saavuttaminen kasvaa eksponentiaalisesti mitä suurempia testikattavuuksia tähdätään. Samalla virheiden määrä näyttää laskevan vain lineaarisesti.
3. Yli 70 % testikattavuus vaatii haasteellista poikkeustenkäsittelyä lähdekoodissa.

4. Lopulta 70 % testikattavuus on vain suuntaa antava luku, tehokkuuden kannalta optimaalisin testikattavuus vaihtelee suuntaan tai toiseen ohjelmistosta riippuen.

Testikattavuuden pätevyyteen vaikuttavia tekijöitä

Testikattavuuden pätevyyteen vaikuttaa samat piirteet, jotka vaikuttivat koodikirnun ja verkkoanalyysin pätevyyteen. Empiiristä tutkimuksista on vaikea tehdä yleisiä päätelmiä niiden kontekstisidonnaisen luonteen takia. Tutkimuksen pätevyyttä tukevoittaa kuitenkin se, että tapaustutkimuksessa tutkittiin kahta täysin erilaista ohjelmistoa. Ohjelmistojen takana oli eri organisaatio, sovellusala, ohjelmointikieli ja koko. Samalla kehittäjätiimien ja käyttäjäkuntien koko oli eri. Voidaankin olettaa, että testikattavuus soveltuu hyvin muihin ohjelmistoihin.

3.5 Mutaatiotestaus

Testikattavuuden ongelmaksi muodostuu se, että se ei arvioi testien laatua. Se tarkastelee vain mitä osia ohjelmiston lähdekoodista on katettu testitapauksilla. Useinkaan pelkkä testikattavuus ei arvioi ohjelmiston laatua halutulla tasolla.

Testien laadun arvioimiseen on ehdotettu mutaatiotestausta [YH11]. Mutaatiotestaus arvioi testien sopivuutta niiden kattamaan lähdekoodiin simuloimalla yleisempiä virheitä joita kehittäjät tekevät. Ohjelmiston lähdekoodia kuten sen sisältämiä ehtoja harkitusti muuttamalla pystyy mutaatiotestaus jäljittelemään kaikki mahdolliset testitapaukset. Mutaatiotestaus ei pelkästään arvioi testien laadukkuutta vaan sen on havaittu parantavan suoraan testien laatua. Sillä voidaan priorisoida testien kohteita sekä minimoimaan niitä testaavaa koodia säilyttäen kuitenkin niiden alkuperäisen kattavuuden.

Yksinkertaisten syntaksimuutosten avulla mutaatiotestaus pystyy muodostamaan virheellisiä mutanttiversioita ohjelmistosta, joiden ei pitäisi mennä testeistä läpi. Jokainen mutanttiversio on syntaksiltaan toista hieman erilainen. Valitut mutantit ajetaan ohjelmiston testitapausten läpi ja samalla tutkitaan havaitsevatko testit mutanttien väärän toiminnallisuuden.

Ideana on, että mitä suuremman määrän mutanttien aiheuttamia virheitä testitapaukset löytävät, sitä parempia ne ovat laadullisesti. Mutaatiotestaus on hyvin monikäyttöinen metriikka ohjelmiston testitapausten arvioimiseen. Sitä voidaan käyttää yksikkö-, integraatio- ja määritelmätason testeihin. Mutaatiotestaus soveltuu hyvin eri käyttötapauksiin perusohjelmistojen lisäksi kuten tietokoneympäristöjen, web-sovellusten, verkkojen ja turvallisuuden arvioimiseen.

Mutaatiotestauksen ongelmaksi muodostuu kuitenkin mahdollisten mutanttien valtava määrä. Jokaiselle syntaktiselle muutokselle ei ole järkevää saati mahdollista muodostaa mutanttia. Olio-ohjelmointi lisää haastavuutta

sillä se tuo paljon korkean tason ominaisuuksia kieleen ja syntaksiin. Mutaatiotestaukseen on tärkeää valita vain oleellisimmat tapaukset, mutta tämä on kuitenkin itsessään erittäin haastava ongelma. Syntaksisesti erilaiset mutantit voivat olla toiminnaltaan täysin samanlaisia. Tätä ei voida kuitenkaan automaattisesti päätellä sillä ohjelmien yhtäläisyys on ratkeamaton ongelma. Mutaatiotestauksen tuleekin valita järkevästi vain osa mutanteista ja suorittaa nämä tehokkaasti. Ongelmaan on kuitenkin ehdotettu useita eri ratkaisuja.

Ohjelmoijat ovat usein täysin päteviä tuottamaan koodia joka on vähintään hyvin lähellä tarkoitettua toiminnallisuutta. Virheet lähdekoodissa ovat usein hyvin pieniä. On syytä olettaa, että mutaatiotestauksen tarvitsee keskittyä vain lähimpiin syntaksivirheisiin. Yleisimmät mutaatio-operaattorit muuttavat, lisäävät ja poistavat muuttujia ja ehtolauseita. Näitä ovat esimerkiksi JA -operaattoreiden muuttaminen TAI -operaatioiksi.

Vaikka mutaatiotestausta on tutkittu erittäin kattavasti, on sen käytäntöön soveltamisessa vielä haasteita. Mutaatiotestausta voitaisiin käyttää hyödyksi kattavammin testitapausten ja testikehysten parantamisessa. Ennen kaikkea mutaatiotestaus voisi tarjota käytännöllisen keinon ohjelmiston laadun parantamisessa.

4 Kehittäjien käytänteet

Ohjelmiston koodin tuottajana on aina ihminen: kehittäjien käytänteillä ja ohjelmiston kehitysmalleilla on suuri laadullinen merkitys. Avainkysymykseksi nousee paikallistaa ne käytänteet, joilla on ratkaiseva yhteys ohjelmiston laatuun.

Vanhojen raskaaseen ennakkosuunniteluun pohjautuvien menetelmien, kuten vesiputousmallin, rinnalla on noussut uusia ketterän kehityksen malleja. Sfetsos ja Stamelos suorittivat katselmuksen ketterän kehityksen empiirisistä tutkimuksista [SS10]. Katselmuksessa käytiin läpi 46 tutkimusta kahdeksasta eri tutkimustietokannasta. He havaitsivat katselmuksessaan, että ketterien kehitysmallien hyöty on merkittävä laadullinen tekijä.

Ketterän kehityksen manifesti (agile manifesto) on muodostunut ketterän kehityksen tavoitteiden ympärille [BBvB+01]. Se painottaa yksilöitä ja yksilöiden vuorovaikutusta, toimivan ohjelmiston merkitystä, asiakkaan merkitystä kehitysprosessin kriittisenä osana ja muutoksiin sopeutuvaa kehitystä. Näiden periaatteiden takaamiseksi, ketterän kehityksen malleille on muodostunut useita käytänteitä. Näillä kehittäjät pystyvät hallinnoimaan ja varmistamaan kehitystyötä, varsinkin laadullisesta näkökulmasta. Ketterän kehitys huomioi nimenomaan asiakkaan kehitysprosessin tärkeänä osana.

Asiakkaan tarpeet tulisi kartoittaa ja taata koko kehitysjakson aikana. Lopulta tuotettavan tuotteen tulisi tuoda jotain arvoa asiakkaalle. Ohjelmiston vaatimuksia on vaikea määrittää kattavasti heti alusta lähtien, siksi

ketterässä mallissa painotetaan muutoksien hyväksymistä. Ohjelmiston kehitys saattaa olla hyvin pitkäaikainen prosessi, tilanteet ja käyttötarkoitukset muuttuvat prosessin aikana. Asiakas saattaa havaita ohjelmiston kannalta tärkeitä asioita hyvinkin myöhään kehityksessä.

4.1 Ketterä kehitys

Ketterän kehityksen idea on mahdollistaa muutokset kehitystyössä. Kehitystyötä tehdään iteratiivisesti pienissä palasissa ja samanaikaisesti painotetaan menetelmiä, jotka kasvattavat ohjelmiston hallintaa ja laatua. Laadun hallinta ja valvonta tulee jakaa koko kehityksen ajalle. Tarkoituksena on vähentää muutoksista johtuvia kustannuksia kehitystyössä [HC01]. Jokaisen iteraation eli palasen jälkeen tulisi asiakkaalle toimittavaa osa ohjelmiston toiminnallisuudesta. Tällöin asiakas voi havaita jo aikaisessa vaiheessa mahdolliset ongelmat tavoitteidensa ja toivomusten osalta ja pyrkiä selventämään niitä kehittäjille. Ketterän kehityksen on todettu vähentävän kehitykseen kuluvaa aikaa pitkällä tähtäimellä [HC01].

Ketterässä kehityksessä laadun valvonta ei ole pelkästään yhden henkilön tehtävä. Jokainen kehitystyöhön osallistuva henkilö tekee sitä jatkuvasti omalta osaltaan. Henkilöille pitää luoda ilmapiiri ja ympäristö, jossa tämä on mahdollista. Kehitystiimin ja asiakkaan välinen luottamus on ensisijaisen tärkeää. Jatkuva tiedonvälitys ja keskustelu on olennainen osa tämän saavuttamista. Asiakas on jatkuvasti kehityksessä mukana arvioiden sovelluksen soveltavuutta tarkoituksiinsa. Samanaikaisesti hänen tulee varmistaa tuotteen toimivuus käyttäjien kannalta. Varsinkin ohjelmiston kehittäjät kokevat tämän erittäin hyödyllisenä kehitysprosessissa [DD08].

Hyvien käytänteiden ja laadullisesti järkevien ratkaisujen seuraaminen parantaa ohjelmiston suunnittelua ja laatua. Mahdollisia ongelmia tulee katselmoida mahdollisimman usein ja tiimin toimintoja kehittää näiden ongelmien osalta. Ketterän kehityksen mallit eivät kuitenkaan aina kerro yksiselitteisesti miten kehitys pitäisi toteuttaa. Ne luovat kehyksen, jonka pohjalta kukin ohjelmistokehittäjä ja tiimi rakentaa omaan tarkoitukseen toimivan kokonaisuuden [Kni07].

Nykyään eniten käytössä olevat ketterän kehityksen mallit ovat Scrum ja XP [SS10]. Suurin osa tutkimuksista on suoritettu XP:stä [DD08]. Scrum tarjoaa ketterälle kehitykselle toimivaksi osoitetun kehyksen ja XP lukuisia ketterään kehitykseen soveltuvia ohjelmistokehityksen käytäntöjä. Scrum keskittyy lähinnä kehityksen hallinnolliseen puoleen: miten ohjelmistokehitys tulisi suunnitella, hallinnoida ja ajoittaa. Tästä syystä Scrum ja XP tukevat hyvin toisiaan [Kni07].

Scrumissa yksittäisiä iteraatioita kutsutaan pyrähdyksiksi, eli Sprinteiksi. Jokainen Sprintti muodostuu yhdessä asiakkaan kanssa valituista ja priorisoiduista ominaisuuksista tai parannuksista, joita kehittäjien tulisi sen aikana pyrkiä toteuttamaan. Itse toteutusta tukee useita XP:n esittämiä käytäntöjä,

muun muassa testilähtöinen kehitys, pariohjelmointi ja suunnittelupeli. Testilähtöisessä kehityksessä ohjelmiston kehityksessä testit kirjoitetaan ennen niiden toteuttavaa toiminnallisuutta. Pariohjelmoinnissa harjaannutetaan kehittäjien taitoja ratkomalla ongelmia yhdessä työskentelyparin kanssa ja suunnittelupelissä yritetään arvioida kehitykseen kuluvaa aikaa ja samalla havaita ja pohtia mahdollisia ongelmia. XP:n käytännöt on käytännössä nykyään sulautunut osaksi Scrumia [Kni07].

Ketterää kehitystä on kritisoitu [DD08]. Ketterä kehitys voi viedä huomion ohjelmiston kokonaissuunnittelusta. Näin ollen ohjelmiston suunnitteluratkaisut saattavat jäädä pirstaleisiksi. Ketterä kehitys sisältää paljon ideologiaa ja tutkimukset ovat suurelta osin vain empiirisiä. Ketterän kehityksen tuoman hyödyn mittaaminen on siksi hyvin haasteellista. Yhdeksi kysymykseksi on nostettu ketterän kehitysmallien soveltaminen isoissa yrityksissä, koska käytänteiden soveltaminen on usein helpompaa pienemmissä kehitystiimeissä. Kuitenkin valtaosa kehittäjästä jotka ovat kokeilleet ketterää kehitystä haluavat jatkaa sen käyttämistä.

Useissa tutkimuksissa on pohdittu ristiriitoja kokeellisten ja empiiristen tutkimusten välillä [DD08, SS10]. Kokeelliset tutkimukset ajoittuvat yleensä hyvin lyhyelle aikajaksolle kun taas empiiriset tutkimukset arvioivat ajallisesti pidempää kehitysprosessia. Siksi kokeelliset tutkimukset päätyvät yleensä maltillisempiin tuloksiin ketterän kehityksen hyödyistä. Empiiriset tutkimukset näkevät ketterien mallien vahvan hyödyn, mutta ongelmaksi muodostuu tulosten yleistäminen. Hyöty voidaan yleensä nähdä vain parannuksena ulkoiseen laatuun [SS10].

4.2 Testilähtöinen kehitys

Testilähtöinen kehitys (test-driven development, TDD) koostuu lähdekoodin kirjoittamisesta testilähtöisesti ja koodin jatkuvasta parantamisesta eli refaktoroinnista. Refaktoroinnissa tehdään pieniä muutoksia ohjelmiston koodiin muuttamatta sen ulkopuolista toiminnallisuutta. Testilähtöisessä kehityksessä testit kirjoitetaan ennen itse toiminnallisuuden ohjelmoimista. Tämän on tarkoitus saada kehittäjä suunnittelemaan ja miettimään uusia toiminnallisuuksia ja niiden ongelmia ennen itse logiikan toteuttamista. Jatkuvalla refaktoroinnilla pyritään rakentamaan toiminnallisuudet paremmiksi, kehittämällä jatkuvasti ohjelmiston lähdekoodista parempaa.

Hyväksymätestien lisäksi, joita käytetään ohjelmiston vaatimusten määrittelemiseen, testilähtöinen kehitys ja refaktorointi parantavat yleisesti ohjelmiston laatua. Suurin osa kokeista ja tapaustutkimusta osoittivat Sfetsoksen ja Stameloksen katselmuksen mukaan laajoja laadullisia parannuksia ohjelmistojen ulkoiseen [SS10]. Julkaisun jälkeiset virheet vähenivät 5 % – 45 %, joissain tapauksissa 50 % – 90 %. Tapaustutkimukset osoittivat suurempaa parannusta kuin kokeet. Kokeiden osalta heikompi parannus voidaan selittää valvotun ympäristön ja ajallisten rajoitusten seurauksena. Pančur ja

Ciglarič mainitsevat heikon parannuksen osasyyksi testilähtöisen kehityksen vaikeuden [PC11]. Testien kirjoittaminen ennen itse toiminnallisuutta vaatii harjaantumista, jota jälkeen kirjoitettujen testien osalta ei vaadita. Siksi testilähtöisen kehityksen hyödyn vaikutukset todennäköisesti ilmentyvät vasta myöhemmin. Vain muutama koe ei havainnut testilähtöisellä kehityksellä olevan merkittävää vaikutusta ohjelmiston ulkoiseen laatuun.

Sisäinen laatu kasvoi testilähtöisen kehityksen ansiosta merkittävissä määrin. Lähdekoodin uudelleenkäytettävyys ja vaivannäkö testaamista varten parani. Osa tutkimuksista osoitti lopullisen ohjelmiston kehitysajan kasvavan, mutta samalla osa tutkimuksista huomasi kehityksen kokonaiskustannusten vähenevän. Tuottavuuden kannalta Sfetsoksen ja Stameloksen katselmoivat tutkimukset osoittivat ristiriitaisia tuloksia: osassa tuottavuus kasvoi, osassa tuottavuudelle ei tapahtunut merkittäviä muutoksia, osassa tuottavuus laski.

4.3 Pariohjelmointi

Pariohjelmointi on erittäin sosiaalinen ja yhteistyöhön perustuva toimintamalli. Se keskittyy kehittäjien yksilöllisiin taitoihin, kokemukseen, ominaispiirteisiin ja persoonallisuuteen. Pariohjelmoinnin tarkoitus on jatkuva suunnittelu ja koodikatselmus kahden kehittäjän kesken. Kehittäjät kirjoittavat yhdessä ohjelmiston toiminnallisuutta. Tämä vähentää virheiden määrää ja parantaa ohjelmiston suunnittelua ja laatua [SS10]. Pariohjelmoinnin havaittiin lisäävän hyvien ohjelmointitapojen käyttöä [DD08].

Sfetsos ja Stamelos havaitsivat pariohjelmoinnin olevan yksi merkittävimmistä laadullista tekijöistä käytännön näkökulmasta. Koodin suunnittelu ja laatu kasvoi 15 % – 65 %. Monimutkaisiin ja vaativiin ongelmiin pariohjelmointi tuotti olennaisesti parempaa koodia kuin ohjelmointi yksin. Pariohjelmoinnin todettiin parantavan tiimityöskentelyn laatua, tiedon ja taitojen parempaa siirtymistä yksilöltä toiselle [DD08, SS10], tehokkaampia ja paremmin suunniteltuja algoritmeja, moraalin kasvua ja luottavaisempia kehittäjiä [SS10]. Pariohjelmointi täten parantaa tuotetun lähdekoodin laatua [DD08].

Pariohjelmoinnin todettiin kuitenkin vaativan enemmän vaivannäköä kehittäjiltä ja näin ollen pariohjelmointi kasvatti kehitystyön kustannuksia. Tehokkuus väheni lievästi ja kehitystyön aikataulutus vaikeutui ja samalla kehitystiimeissä havaittiin persoonallisuus kitkoja. Tutkimukset havaitsivat tiettyjen taito, tieto ja kokemuspiirteiden sopivan paremmin pariohjelmointiin. Erityisesti persoonallisuuspiirteillä havaittiin suuri merkitys: avomieliset ja vastuulliset yksilöt sekä monipuoliset persoonallisuudet ja temperamentit soveltuvat pariohjelmointiin paremmin. Osa kehittäjistä pitävät pariohjelmointia turhauttavana [DD08].

5 Metriikat käytänteiden tukena

Laadun varmistamista rajaa ohjelmistokehityksessä henkilöt, aika ja raha [BBM96, ZN08]. Kehittäjät kohtaavat usein tiukkoja määräaikoja ja rajallisia henkilöresursseja laadun takaamiseen. Johtajat käyttävät käytännössä pelkästään omakohtaisia kokemuksiaan resurssien tehokkaaseen jakamiseen. Yleisenä totuutena pidetään, että monimutkaisiin komponentteihin on syytä varata enemmän aikaa että rahaa [BBM96, ZN08]. Tällä turvataan se, että komponenttien testaus ja tarkastus ohjataan haastavimpiin osa-alueisiin. Johtajilla ei kuitenkaan ole läheskään aina tarvittavaa kokemusta tai tietoa, joiden pohjalta he voisivat tehdä päätöksiä järkevästi. Siitä johtuen päätökset tehdään usein johtajien odotusten mukaan ja tällöin he itse joutuvat arvioimaan laatua puutteellisin tiedoin. Kriittiseksi osaksi muodostuu näin ollen johtajien taito. On hyvin todennäköistä, että laadullisen arvioinnin tehokkuus ja vaatimustaso kärsivät tästä.

5.1 Kehittäjien tuki ja vastuu

Beck ym. puolustavat ketterän kehityksen manifestissa ketterän kehityksen itse organisoituvaa luonnetta [BBvB+01]. Kun kehittäjille annetaan tarpeeksi tukea, ottavat he itse vastuun ohjelmiston laadullisista puolista. Jatkuva hyvien käytänteiden ja suunnitteluperiaatteiden seuraaminen johtaa lopulta ohjelmiston laadun kasvuun [SS10]. Tiimien tulee arvioida näiden onnistumista tarpeeksi usein, jotta mahdollisiin ongelmiin voidaan puuttua ja tiimin käytänteitä hienosäätää. Vastuun siirtäminen johtajilta kehittäjille nopeuttaa virheiden löytämistä ja niihin puuttumista [DD08]. Ohjelmiston lähdekoodin yhteisvastuu kasvatti tiimin jäsenten moraalia.

Tiimien ei tule luistaa ohjelmiston sisäisen laadun varmistamisesta [Kni07]. Se on perustavanlaatuisesti kehittäjien vastuu. Kehitys täytyy tehdä laadukkaasti vaikka se veisi enemmän aikaa kun alun perin suunniteltiin. Muuten kehitettyjä ominaisuuksia ei voida hyväksyä osaksi ohjelmistoa. Tiimien täytyy löytää itselleen optimaaliset työskentelytavat ja sovittaa työvauhti sopivaksi kehityksen kannalta. On järkevämpää toteuttaa vähemmän kerralla, mutta toteuttaa se laadukkaasti. Suunnitelmien tiukka noudattaminen ei edistä ohjelmistojen tarkoitusta asiakkaan tarpeiden ja toiveiden toteuttamisessa [HC01].

Suunnittelupeli ja Sprintin suunnittelu kattavat toiminnan, jota voitaisiin verrata suoraan laadullisten määritelmien kehykseksi. Ne muodostavat raamit laadukkaalle kehitykselle. Testilähtöinen kehitys, pariohjelmointi ja jatkuva integraatio taas paneutuvat laadullisen toteutuksen puoliin [SS10]. Ne tukevat kehittäjää toteuttamaan laadukasta ohjelmistoa. Toimiva koodi on ketterän kehityksen tärkeimpiä tavoitteita [HC01]. Suora kommunikaatio kehittäjien keskuudessa ja ennen kaikkea asiakkaan kanssa tuottaa todennäköisesti paremman tuloksen kuin yksilökeskeinen ympäristö.

Moni XP:n käytännöistä yhdessä, kuten suunnittelupeli, pariohjelmointi ja testilähtöinen kehitys paransivat ohjelmiston laatua Sfetsoksen ja Stameloksen katselmuksen mukaan. Suunnittelupelin havaittiin parantavan kehityksen työmäärän ajallista estimointia. Käytänteiden seuraamisesta havaittiin refaktoroinnin ja tuottavuuden kasvu. XP -käytänteiden havaittiin toimivan paremmin nimenomaan pienissä kehittäjätiimeissä.

5.2 Hyvät ohjelmointimallit ja tavat

Lopulta laadukkaan ohjelmiston tekee laadukkaat kehittäjät. Ulkoinen ja sisäinen laatu lähtee siitä, että ohjelmisto suunnitellaan, toteutetaan ja testataan käyttäen hyväksi todettuja ohjelmointitapoja ja malleja. Nämä vaihtelevat myötäilen jokaisen ohjelmointikielen ajatusmalleja. Jokaisella kehittäjällä on oma mielipide asiasta.

Hyvän koodin laatuattribuuteiksi voidaan luotella muun muassa lähdekoodin kapselointi (algoritmien yksityiskohtien piilottaminen), koheesio (komponenttien yksi vastuu), riippuvuuksien vähäisyys, toistettavuus, testattavuus ja selkeys [Bai08]. Jokainen näistä laatuattribuuteista pureutuu syvälle ohjelmoinnin juuriin. Hyvillä käytänteillä minimoidaan sortuminen virheisiin ja helpotetaan ohjelmiston ylläpidettävyyttä.

Vaikka ohjelmisto toteutettaisiin pala kerraltaan, hyvät suunnittelu- ja toteutusmallit nousevat lopulta tärkeään rooliin kokonaisuuden kannalta. Jo vuonna 1976 päädyttiin siihen, että rakenteellisesti järkevät ohjelmistot helpottavat ohjelmiston ylläpidettävyyttä ja tuottavat laadullisesti paremman ohjelmiston [LK76].

Huonot ratkaisut johtavat ohjelmiston kannalta tekniseen velkaan [FFS12]. Tämän voidaan nähdä vaikuttavan ohjelmiston laatuun. Fowler määrittelee lukuisia koodihajuja, joita voidaan pitää ohjelmiston huonon sisäisen laadun ja ylläpidettävyyden merkkinä. Koodihajuja ovat muun muassa suuret komponentit ja toistuva koodi. Suuret komponentit ovat muita virheherkempiä ja huomattavasti vaikeampia ylläpitää. Varsinkin toistuvaa koodia pidetään huonona ratkaisuna jota pitäisi välttää [FFS12]. Toisaalta koodihajujen ja heikomman ylläpidettävyyden yhteys on jokseenkin kyseenalainen [SYA+12]. Koodihajut eivät välttämättä lisää kehityksen työmäärää.

5.3 Kehityksen varmistaminen

Tutkimukset koodikirnusta, verkkoanalyysista ja testikattavuudesta ohjelmiston laadullisina metriikoina toivat esille inhimillisen tekijän laadun takaamisessa [NB05, ZN08, MNDT09]. Kehittäjän tulee aktiivisesti itse vaikuttaa ohjelmiston laatuun. Sfetsoksen ja Stameloksen katselmuksen pohjalta voidaan olettaa, että ketterä kehitys on oivallinen käytäntö ohjelmiston laadun varmistamisessa.

Nagappan ja Ball toivat esille koodikirnun ja ohjelmiston riippuvuuksien

käyttämisen laadullisina metriikoina [NB05, NB07]. Testien kirjoittaminen tulisi kanavoida näiden metriikoiden ilmaisemiin virheherkkiin komponentteihin [MNDT09]. Voidaan nähdä, että testilähtöinen kehitys tukisi näiden metriikoiden tavoitteita osuvasti. Nagappan ja Ball painottivat erikseen hyviä versionhallinnan käytäntöjä [NB05]. Ohjelmistoon tehdyt muutokset tulisi rekisteröidä pienissä palasissa versionhallintaan mahdollisimman aikaisin ja usein. Tällä säästetään kallisarvoista kehitystyön historiaa ja mahdollistetaan ongelmatapauksissa paluu vanhoihin toimiviin koodiversioihin.

Zimmermannin ja Nagappanin esittämällä verkkoanalyysillä [ZN08] voitaisiin taas ohjata sekä testauksen, että suunnittelun varoja sinne missä ne ovat tärkeimpiä [NB07, MNDT09]. Mutaatiotestauksella pystytään arvioimaan testien laadukkuutta ja priorisoimaan niitä lähdekoodin kohteita joita tulisi testata kattavammin [YH11]. Samalla vähennettäisiin inhimillisten, johtajien tai muiden kehitystiimin jäsenten tietotaitoon liittyviä riskejä ja parannettaisiin näin ohjelmiston laatua.

Vaikka metriikoita on tutkittu jo pitkään, niiden täydellistä potentiaalia ei ole kuitenkaan vieläkään valjastettu käyttöön [YH11]. Metriikat ovat perimmiltään vielä hyvin tieteellisiä eikä niitä arvioivia kehittyneitä työkaluja ole vielä helposti saatavilla. Suurin hyöty niistä saadaan vasta kun ne saadaan jokaisen kehittäjän käsiin.

6 Yhteenveto

Kehitysvaiheessa olevan ohjelmiston laadun varmistaminen on hankalaa [BBM96, NB05, NB07, ZN08, MNDT09]. Automaattisesti analysoitavat metriikat tarjoavat yhden keinon kohdentaa resursseja laadun takaamiseksi. CK-metriikat keskittyvät olio-ohjelmoinnin piirteistä johtuvien vikaherkkyyksien havaitsemiseen [CK94, BBM96]. Nagappan ja Ball esittävät suhteellisen koodikirnu-tekniikan järjestelmän virhetiheyden ennakoimiseen [NB05]. Koodikirnu mittaa ja ilmaisee määrällisesti ohjelmiston komponentteihin kohdistuvia muutoksia tietyn ajanjakson aikana. Komponentit jotka muuttuvat paljon ovat tutkimuksen mukaan muita herkempiä virheille.

Zimmermann ja Nagappan esittävät verkkoanalyysin suorittamista komponenttien riippuvuusverkoille [ZN08]. Verkkoanalyysillä voidaan paikallistaa ohjelmiston kriittiset komponentit. Keskeisessä roolissa olevat komponentit sekä yksittäiset komponentit, joilla on suuri määrä keskinäisiä riippuvuuksia, ovat yleisesti herkempiä virheille.

Mockus, Nagappan ja Dinh-Trong tutkivat testien laadullista arviointia keinona havaita virheherkkiä komponentteja ohjelmistosta [MNDT09]. Testien analysoimisessa tulisi keskittyä nimenomaan niiden kykyyn havaita mahdollisia virheitä ohjelmistosta. Taustalla on olettamus, että jos jokin yksittäinen looginen ehto tai polku ei ole katettu vähintään yhdellä testillä, ei sen mahdollisesti sisältämiä virheitä pystytä havaitsemaan. Voidaankin olet-

taa, että suurempi testikattavuus löytää todennäköisesti enemmän virheitä ja takaa näin ollen paremman laadun. Mutaatiotestauksella voidaan arvioida ohjelmiston testien laatua [YH11]. Alkuperäisen ohjelmiston lähdekoodin syntaksia muuttamalla pystytään arvioimaan testien tehokkuutta havaita mahdollisia virheitä ohjelmiston lähdekoodissa.

Ohjelmiston koodin kehittäjänä on lopulta aina ihminen. Laadun takeeksi ei voida luetella pelkästään mekaanisia laatua arvioivia metriikoita. Kehittäjän käytänteillä on suuri laadullinen merkitys ohjelman kaikissa kehitysvaiheissa, joten ohjelmistotuotantomenetelmät nousevat suureen rooliin. Niiden tulisi ohjata laadukasta kehitystä. Lopulta laadukkaan ohjelmiston tekee laadukkaat kehittäjät. Ulkoinen ja sisäinen laatu lähtee siitä, että ohjelmisto suunnitellaan, toteutetaan ja testataan käyttäen hyväksi todettuja ohjelmointitapoja ja malleja.

Vanhojen raskaaseen ennakkosuunniteluun pohjautuvien tuotantomenetelmien, kuten vesiputousmallin, rinnalla on noussut uusia ketterän kehityksen malleja. Ne painottavat yksilöitä ja yksilöiden vuorovaikutusta, toimivan ohjelmiston merkitystä, asiakkaan merkitystä kehitysprosessin kriittisenä osana ja muutoksiin sopeutuvaa kehitystä [BBvB$^+$01]. Useat empiiriset tutkimukset tukevat ketterien kehitysmallien hyötyä merkittävänä laadullisina vaikuttajana [SS10].

Lähteet

[Bai08] Bain, Scott L.: *Emergent Design: The Evolutionary Nature of Professional Software Development*. Addison-Wesley Professional, 2008, ISBN 0321509366.

[BBM96] Basili, V.R., Briand, L.C. ja Melo, W.L.: *A validation of object-oriented design metrics as quality indicators*. Software Engineering, IEEE Transactions on, 22(10):751 – 761, Oct. 1996, ISSN 0098-5589. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=544352`.

[BBvB+01] Beck, K., Beedle, M., Bennekum, A. van, Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J. ja Thomas, D.: *Manifesto for Agile Software Development*, 2001. `http://agilemanifesto.org/`.

[CK94] Chidamber, S.R. ja Kemerer, C.F.: *A metrics suite for object oriented design*. Software Engineering, IEEE Transactions on, 20(6):476 – 493, June 1994, ISSN 0098-5589. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=295895`.

[DD08] Dybå, T. ja Dingsør, T.: *Empirical studies of agile software development: A systematic review*. Information and Software Technology, 50(9 – 10):833 – 859, 2008, ISSN 0950-5849. `http://www.sciencedirect.com/science/article/pii/S0950584908000256`.

[FFS12] Fontana, F.A., Ferme, V. ja Spinelli, S.: *Investigating the impact of code smells debt on quality code evaluation*. Teoksessa *Managing Technical Debt (MTD), 2012 Third International Workshop on*, sivut 15 – 22, June 2012. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6225993`.

[HC01] Highsmith, J. ja Cockburn, A.: *Agile software development: the business of innovation*. Computer, 34(9):120 – 127, Sept. 2001, ISSN 0018-9162. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=947100`.

[IEE06] IEEE: *IEEE Standard for Developing a Software Project Life Cycle Process*. IEEE Std 1074-2006 (Revision of IEEE Std 1074-1997), 2006. `http://ieeexplore.ieee.org/servlet/opac?punumber=11045`.

[ISO05] ISO: *Quality management systems – Fundamentals and vocabulary.* 2005. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=42180.

[ISO11] ISO/IEC: *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.* 2011. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=35733.

[Kni07] Kniberg, H.: *Scrum and XP from the Trenches.* C4Media/InfoQ.com, 2007, ISBN 9781430322641. http://www.infoq.com/minibooks/scrum-xp-from-the-trenches.

[LK76] Lucas, H. C. ja Kaplan, R. B.: *A Structured Programming Experiment.* The Computer Journal, 19(2):136 – 138, 1976. http://comjnl.oxfordjournals.org/content/19/2/136.full.pdf+html.

[MNDT09] Mockus, A., Nagappan, N. ja Dinh-Trong, T.T.: *Test coverage and post-verification defects: A multiple case study.* Teoksessa *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, sivut 291 – 301, Oct. 2009. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5315981.

[NB05] Nagappan, N. ja Ball, T.: *Use of relative code churn measures to predict system defect density.* Teoksessa *Proceedings of the 27th international conference on Software engineering*, ICSE '05, sivut 284 – 292, New York, NY, USA, 2005. ACM, ISBN 1-58113-963-2. http://doi.acm.org/10.1145/1062455.1062514.

[NB07] Nagappan, N. ja Ball, T.: *Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study.* Teoksessa *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, sivut 364 – 373, Sept. 2007. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4343764.

[PC11] Pančur, M. ja Ciglarič, M.: *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.

[SS10] Sfetsos, P. ja Stamelos, I.: *Empirical Studies on Quality in Agile Practices: A Systematic Literature Review*. Teoksessa *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, sivut 44 – 53, Oct. 2010. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5654783.

[SYA+12] Sjøberg, D., Yamashita, A., Anda, B., Mockus, A. ja Dyb a, T.: *Quantifying the Effect of Code Smells on Maintenance Effort*. Software Engineering, IEEE Transactions on, PP(99), 2012, ISSN 0098-5589. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6392174.

[YH11] Yue, J. ja Harman, M.: *An Analysis and Survey of the Development of Mutation Testing*. Software Engineering, IEEE Transactions on, 37(5):649 – 678, Sept. - Oct. 2011, ISSN 0098-5589. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5487526.

[ZN08] Zimmermann, T. ja Nagappan, N.: *Predicting defects using network analysis on dependency graphs*. Teoksessa *Proceedings of the 30th international conference on Software engineering*, ICSE '08, sivut 531 – 540, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-079-1. http://doi.acm.org/10.1145/1368088.1368161.