

hyväksymispäivä arvosana

arvostelija

Testauslähtöinen ohjelmistokehitys ja testaustehokkuus

Nikita Zhuk

Helsinki 5.5.2006

Tutkielma

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tiedekunta/Osasto — Fakultet/Sektion — Faculty		Laitos — Institution — Department	
Matemaattis-luonnontieteellinen		Tietojenkäsittelytieteen laitos	
Tekijä — Författare — Author			
Nikita Zhuk			
Työn nimi — Arbetets titel — Title			
Testauslähtöinen ohjelmistokehitys ja testaustehokkuus			
Oppiaine — Läroämne — Subject			
Työn laji — Arbetets art — Level		Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Tutkielma		5.5.2006	29 sivua
Tiivistelmä — Referat — Abstract			
<p>Vesiputousmalli sekä lineaarinen, etukäteen tapahtuvaan suunnitteluun ja perusteelliseen dokumentointiin perustuva ohjelmistokehitys ovat saaneet viimeaikaisissa tutkimuksissa ja artikkeleissa huomattavia määriä kritiikkiä. Paljon julki-suutta saaneet kevyemmät ketterät prosessimallit ja niiden taustalla olevat periaatteet ovat synnyttäneet aktiivista keskustelua niin tieteellisissä piireissä kuin myös yritysmaailmassa. Varsinkin ketterien mallien taustalla oleva testauslähtöisyys kääntää perinteiset olettamukset pääläelleen - prosessin ohjaavana voimana ei olekaan dokumentaatio vaan testit.</p> <p>Puhtaassa testauslähtöisyydessä testit kirjoitetaan ennen tuotantokoodia. Näiden testien tehokkuus voi olla kuitenkin puutteellista, sillä niillä pyritään ilmaisemaan sitä ohjelmiston toiminnallisuutta, joka tulisi toteuttaa. Testit eivät siis pyri mahdollisimman kattavaan testaukseen, ja siksi pelkkä testien olemassaolo ilman niiden tehokkuuden valvontaa voi olla ongelmallista.</p> <p>Tämä tutkielma esittelee lyhyesti prosessimallien eri tyyppejä sekä tarkastelee mallien syntyä ja niiden käyttöä erilaisissa projekteissa käyttäen kronologista lähestymistapaa. Tämän jälkeen otetaan tarkempaan tarkasteluun testauslähtöisyys, joka kuuluu oleellisena osana joihinkin iteratiivisiin ja ketteriin malleihin. Sen jälkeen käsitellään testaustehokkuutta, joka tässä tutkielmassa käsittää testien kykyä löytää ohjelmistosta virheitä suhteessa testien suunnitteluun, toteutukseen ja suoritukseen vaadittuihin resursseihin. Testaustehokkuutta pyritään peilaamaan myös testauslähtöiseen ohjelmistokehitykseen. Testauksen haasteista esitellään myös yksi konkreettinen esimerkki Helsingin Yliopiston portaalihankkeen näkökulmasta. Lopuksi koostetaan tarkastelun tulokset sekä kerrotaan aiheeseen liittyvän jatkotutkimuksen mahdollisista suunnista.</p> <p>ACM Computing Classification System (CCS): D.2.9 [Management] B.2.3 [Reliability, Testing, and Fault-Tolerance]</p>			
Avainsanat — Nyckelord — Keywords			
prosessimallit, testauslähtöinen, ohjelmistokehitys, testaus, tehokkuus, alma			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — övriga uppgifter — Additional information			

Sisältö

1 Johdanto	1
2 Prosessimallien esittely	1
2.1 Vesiputousmalli	1
2.2 Iteratiiviset ja ketterät mallit	2
2.3 Prosessimallien kehitys	3
2.3.1 Kehitys ennen 1980-lukua	3
2.3.2 1980-luku	4
2.3.3 1990-luku	6
2.3.4 2000-luku	6
3 Testauslähtöisyyden pääperiaatteet	6
3.1 Testit ennen koodia	7
3.2 Testit ohjaavat suunnittelua	8
3.3 Refaktorointi	8
3.4 Regressiotestaus	9
3.5 Testit validointivälineenä	9
4 Testaustehokkuus ja kustannukset	10
4.1 Mustalaatikkotestaus	11
4.2 Lasilaatikkotestaus	12
4.3 Kattavuusmittareiden käyttö testien tehokkuuden tarkkailussa . .	13
4.4 Testejä vai esimerkkejä?	14
4.5 Kulttuurin ja tekniikan haasteet	15
4.6 Automatisointi	17
5 Tapaus Alma	18
5.1 Prosessimallin valinta	19
5.2 Testaus ja automatisointi	20

	iii
6 Yhteenveto	21
Lähteet	23

1 Johdanto

Perinteisissä prosessimalleissa ohjelmistoa testataan vasta sen jälkeen, kun se on saatu joko kokonaan tai osittain valmiiksi [HVZ04]. Koska järjestelmän julkaisuajankohta on projektin lopussa yleensä jo kiinnitetty, testauksessa esille tulleita virheitä ei useinkaan ehditä korjaamaan ja näin virheellisesti toimivia ohjelmistoja päätyy asiakkaiden käyttöön [Hen04]. Systemaattinen testaaminen on tullut myös yhä hankalammaksi ja aikaavievämmäksi [Whi00].

Näiden ongelmien ratkaisuksi on esitetty ns. *testauslähtöisiä prosessimalleja* (test-driven process models), jotka perustuvat ohjelmakoodin testaamiseen pienemmissä osissa eli yksiköissä [Jef99]. Näiden prosessimallien pääperiaatteena on, että testit kirjoitetaan aina ennen yksikköjen toteutusta, jolloin ohjelmisto tulee testattua paremmin ja virheet löydettyä ja korjattua mahdollisimman aikaisessa vaiheessa [Amb03]. Yksiköiden testaamisen lisäksi testeillä pyritään myös varmistamaan, että ohjelmisto kokonaisuudessaan täyttää sille alun perin asetetut vaatimukset [Jef99].

Testauslähtöisistä prosessimalleista puhuttaessa keskitytään yleensä testauksen työvaiheisiin ja korostetaan testien olemassaoloa ennen testattavan koodin toteutusta. Testauksella on siis suuri vastuu projektin onnistumisesta. Kuitenkin pitää aina ottaa huomioon myös se, kuinka hyvin testit löytävät ohjelmakoodin virheitä ja kuinka testauksesta saa mahdollisimman kustannustehokasta. Perinteisessä IT-alan organisaatiokulttuurissa testausta ei arvosta riittävän korkealle ja se on ensimmäisenä kärsimässä projektin aikataulupaineista ja resurssipulasta [HMe02, Mur94].

2 Prosessimallien esittely

Tärkeimpinä luokitteluperusteina prosessimalleille voidaan käyttää dokumentoinnin ja suunnittelun asemaa kussakin mallissa sekä mallin mukaisen prosessin rakennetta. Tässä luvussa esitellään lyhyesti prosessimallien päätyypit käyttäen edellä mainittuja seikkoja luokitteluperusteina.

2.1 Vesiputousmalli

Perinteinen vesiputousmalli on dokumenttilähtöinen, toisin sanoen sen tavoitteena on dokumentoida etukäteen järjestelmän vaatimuksia, toimintaa sekä rakennetta mahdollisimman tarkasti [HVZ04]. Edellisen vaiheen dokumentaatio ohjaa seuraavaa vaihetta. Eri vaiheista syntyvien tietojen dokumentointi mahdollistaa myös näiden tietojen validoinnin käyttäen ulkoisia tietolähteitä (esimerkiksi kunkin vaiheen asiantuntijoita) [Boe02]. Tällä tavoin pyritään pienentämään projektiyhtymän tietotason mahdollisesta yliarvioinnista johtuvaa riskiä. Mahdolliset ohjelmistoon tulevat muutokset aiheuttavat samalla kuitenkin jatkuvan dokumentaation päivitystarpeen, mikä lisää joko dokumentaation vanhentumisen riskiä tai dokumentaation päivitysten vaatimia kuluja.

Vesiputousmallin lähtökohtana on, että virheelliset ratkaisut eivät ole vaihtoehtoja ja ratkaisut pitää tehdä kerralla oikein [Sot01]. Vesiputousmalli oli ensimmäinen laajalle levinnyt prosessimalli, joten sitä kutsutaan myös *perinteiseksi malliksi* (traditional model), vaikkakin jäljempänä tehty historiallinen tarkastelu osoittaa, että vesiputousmalli ei suinkaan kehittynyt ennen muita malleja.

2.2 Iteratiiviset ja ketterät mallit

Iteratiivisissa malleissa koko ohjelmistoa ei kehitetä kerralla, vaan sen kehitys on jaettu osiin [LaB03]. Toisin kuin vesiputousmallissa, iteratiivisissa malleissa ei oleteta, että ratkaisuja pystyttäisiin tekemään kerralla oikein ja tavoitteena onkin löytää nämä väistämättömät virheet mahdollisimman nopeasti [Sot01]. Iteratiivisissa malleissa ohjelmistoa kehitetään toistuvissa iteraatioissa, jotka yleensä käsittävät yhä laajenevaa vaatimusjoukkoa [LaB03]. Näiden mallien lisäksi käytetään myös ns. evolutionäärisiä malleja, jotka välttävät iteratiivisten mallien tavoin raskasta dokumentointia ja painottavat edellisistä iteraatioista sekä asiakkaalta saadun palautteen merkitystä muutoksiin sopeutumisessa [JSa05].

Ketterien prosessimallien pääperiaatteena on niinkään vähentää projekteissa syntyvän ja ylläpitoa vaativan dokumentaation määrää ja *etukäteissuunnittelua* (front-up design), lyhentää ohjelmistojen integrointi- ja julkaisusyklejä sekä painottaa asiakkaalta saadun palautteen merkitystä [HVZ04]. Ketterien mallien iteraatioiden kestot ovat muita iteratiivisia malleja huomattavasti lyhyempiä. Ketteryys ei kuitenkaan tarkoita suunnittelun ohittamista kokonaan, vaan sen seikan hyväksymistä, että kaikkia ohjelmiston vaatimuksiin kohdistuvia muutoksia ei pystytä

huomioimaan etukäteen, jolloin laaja etukäteissuunnittelu ja raskas dokumentointi ei ole järkevää [HVZ04].

Ketterissä prosessimalleissa muutokseen varautuminen tulee ilmi jo prosessin rakenteesta: suunnittelua tehdään vain sen verran, että seuraavaan lyhyeen iteraatioon kuuluvat ominaisuudet saadaan toteutettua [HCo01, Boe02]. Etukäteissuunnittelun puutteesta huolimatta laatua pyritään kuitenkin pitämään korkealla tasolla mm. prototyyppien käytöllä, useilla tapaamisilla, ohjelman rakenteen jatkuvalla parantamisella ja yksinkertaistamisella (eli *refaktoroinnilla*, refactoring) sekä jatkuvalla testauksella.

2.3 Prosessimallien kehitys

Viime vuosien aikana paljon julkisuutta saaneet iteratiiviset ja ketterät prosessimallit voivat tuntua nykyaikaisilta korvikkeilta perinteiselle lineaariselle vesiputousmallille [LaB03]. Kuitenkin viitteitä ei-lineaaristen, iteratiivisten mallien käytöstä on löydettävissä jo 60-luvulta lähtien. Historian valossa näyttääkin siltä, että iteratiiviset ja lineaariset prosessimallit ovat kehittyneet rinnakkain, ja vasta 90-luvulla lineaariset prosessimallit ovat alkaneet jäämään vähemmälle käytölle iteratiivisten mallien yleistyessä.

Prosessimallien historiallista kehitystä on luontevinta tutkia aikajanaan, joka alkaa 60-luvun testauslähtöisyyttä soveltaneesta NASAn projektista ja päättyy nykypäivänä hyvin tunnettuihin ketteriin prosessimalleihin. Kaikkia olemassaolevia prosessimalleja ei ole tämän tutkielman puitteissa mahdollista esitellä, mutta seuraava katsaus pyrkii poimimaan merkittävimpiä pisteitä prosessimallien historian aikajanalta.

2.3.1 Kehitys ennen 1980-lukua

Vuonna 1958 NASA aloitti *Project Mercury* -projektin, joka tavoitteli ensimmäisenä USAn avaruusprojektina miehitettyjen avaruuslentojen aloittamista [Ken00]. Projektin puitteissa järjestettiin kuusi miehitettyä lentoa. Project Mercuryn ohjelmistokehityksessä käytettiin erittäin lyhyitä, noin puolen päivän pituisia iteraatioita, ja projekti noudatti testauslähtöisyyden tärkeintä periaatetta - testit kirjoitettiin ennen vastaavaa toiminnallisuutta [LaB03].

70-luvulla Winston Royce [Roy70] esitteli artikkelissaan "Managing the Develop-

ment of Large Software Systems" prosessimallin, joka sai myöhemmin nimityksen vesiputousmalli. Royce esitteli lineaarisen mallin, jossa vaatimusmäärittely, suunnittelu, toteutus ja ylläpitovaiheet olivat erillisiä ja seurasivat toinen toisiaan. Samassa yhteydessä hän kuitenkin suositteli koko lineaarisen prosessin suorittamista kahteen kertaan, mikäli ohjelmistoa kehitettiin ensimmäistä kertaa. Royce myös ehdotti, että jos johonkin erityisesti laajaan projektiin liittyy paljon uusia tai tuntemattomia vaatimuksia, niin voisi olla perustelua järjestää ensin lyhyempi pilottiprojekti [LaB03]. Pilottiprojektista saatu palaute siis ohjaisi varsinaisen projektin kehittämistä. Myöhemmin Winston Roycen poika Walker Royce on sanonut, että lineaarinen vesiputousmalli oli tarkoitettu vain kaikkein suoraviivaisimpiin projekteihin ja että Winston Roycen tarkoituksena oli itseasiassa suositella iteratiivista lähestymistapaa monimutkaisempiin järjestelmiin.

Vuonna 1972 IBM Federal Systems Division kehitti US Trident-sukellusveneen komento- ja ohjausjärjestelmän käyttäen iteratiivista prosessimallia [One83, LaB03]. Projektin aikataulussa pysyminen oli erittäin tärkeää, sillä muussa tapauksessa IBM olisi joutunut maksamaan myöhästymisestä suuria päivittäisiä sakkoja. Projekti päätettiin jakaa neljään noin kuuden kuukauden mittaiseen iteraatioon, jotta projektin riskejä ja monimutkaisuutta pystyttäisiin hallitsemaan paremmin. IBM kehitti 70-luvulla myös helikoptereissa ja laivoissa käytetyn laajan hajaute- tun LAMPS-järjestelmän, jonka kehitys vaati 200 henkilötyövuotta [Mil99]. Sekä projektin jokainen 45:sta iteraatiosta että projekti kokonaisuudessaan onnistuttiin toteuttamaan ajallaan ja budjetin puitteissa.

2.3.2 1980-luku

Vuonna 1982 Daniel McCracken ja Michael Jackson [McJ82] arvostelivat vesiputousmallia ja syyttivät sitä ohjelmistotuotannon ongelmista jotka johtuivat kommunikaation epäonnistumisesta järjestelmien kehittäjien ja loppukäyttäjien välillä. Heidän näkemyksensä mukaan prosessimallin tulisi joko tarjota asiakkaalle sellaisia työkaluja, joista hän voisi itse rakentaa tarvittavan järjestelmän tai, jos tämä ei ollut mahdollista, asiakas tulisi ottaa mukaan ohjelmistotuotantoprosessin kaikkiin vaiheisiin. He korostivat myös sitä, että koko ohjelmiston kaikkia vaatimuksia ei voida edes periaatteessa määritellä etukäteen.

Vuonna 1987 Frederick P. Brooks, Jr. kirjoitti kuuluisaksi tulleessa artikkelissaan "No silver bullet: Essence and accidents of software engineering" [Bro87]:

Monet tämän päivän ohjelmistojen hankinnat perustuvat sille olettamukselle, että on mahdollista määritellä etukäteen vaatimukset täyttävä järjestelmä, kilpailuttaa sen toteuttajat, toteuttaa se ja asentaa se. Mielestäni olettaus on pohjimmiltaan väärä ja monet ohjelmistojen hankintaan liittyvät ongelmat johtuvat tästä virheolettamuksesta.

Brooksin mukaan ohjelmistot olivat tulleet jo tuolloin niin monimutkaisiksi, että niiden täydellinen ja tarkka määrittelemine etukäteen oli mahdotonta. Hän suositteli sen sijaan vaiheittaista, iteratiivista mallia monimutkaisten ohjelmistojen kehitykseen.

Vuonna 1986 David Parnas ja Paul Clements [PaC86] listasivat joukon perustavanlaatuisia ongelmia, jotka olivat syynä siihen, että ohjelmistoprojektit eivät koskaan olleet eivätkä voi olla täysin "rationaalisia" eli ennalta-arvattavia. Vaatimusten kartutusvaiheessa ohjelmiston tilaajalla ei useimmissa tapauksissa ole selkeää kuvaa siitä, mitä ominaisuuksia ohjelmistolta halutaan. Toisaalta vaatimusten täydellinen listaaminen ja ymmärtäminen ei riitä ohjelmiston täydelliseen suunnitteluun, sillä usein uusia vaatimuksia tai suunnittelurajoitteita tulee esille vasta ohjelmiston toteutusvaiheessa. Ohjelmiston vaatimukset voivat muuttua kesken projektin myös ulkoisista syistä. Jos vaatimukset muuttuvat, ohjelmiston osien suunnittelua pitää muuttaa. Tällainen evoluution kautta syntyvä suunnittelu eroaa "rationaalisesta" suunnittelusta, koska se ei ole ennalta määriteltävissä. Muun muuassa näiden syiden takia on epärealistista olettaa, että ohjelmistosuunnittelija pystyisi suunnittelemaan virheettömän, rationaalisesti toimivan ohjelmiston käyttäen pelkästään etukäteen määriteltyä vaatimusten joukkoa.

80-luvun loppupuolella USAn puolustusministeriöllä (US Department of Defense) oli suuria ongelmia ohjelmistohankintojen kanssa - esimerkiksi vuonna 1999 tehdyssä laskelmassa todettiin, että 75% projekteista epäonnistuivat tai niiden tuottamia ohjelmistoja ei koskaan käytetty ja vain 2% projektien tuottamista ohjelmistoista oli käytössä ilman suurempia muutoksia [Jar99]. Näin ollen puolustusministeriö muutti käytäntöjään ja salli myös iteratiivisten menetelmien avulla kehitettyjen ohjelmistojen hankinnan [LaB03]. Päätös perustui Frederick P. Brooksin johtaman työryhmän tekemään raporttiin, jonka mukaan paras lopputulos sekä teknisesti että taloudellisesti saadaan, kun kehitettävän ohjelmiston kokoa kasvatetaan astettain sekä sijaan, että koko ohjelmisto pyrittäisiin määrittelemään, spesifioimaan ja toteuttamaan kerralla.

2.3.3 1990-luku

90-luvun alussa syntyi Dynamic Systems Development Method Consortium (DSDM), joka pyrki kehittämään yhdenmukaisen ja itsenäisen ohjelmisto- ja prosessikehityksen, joka tukisi RAD-mallin (Rapid Application Development) mukaista ohjelmistokehitystä [Dyn06]. RAD-mallissa järjestelmä jaetaan muutamiiin itsenäisiin komponentteihin, joita kehitetään toisistaan riippumattomasti.

Vuosikymmenen puolessa välissä kehitettiin *Rational Unified Process*-metodi (RUP), joka perustui iteratiiviseen ohjelmistojen kehitykseen [Rat01]. RUPin oleellisena osana ovat myös vaatimusten hallinta, komponenttipohjaisten arkkitehtuurien käyttö, ohjelmiston visuaalinen mallintaminen mm. UML-kuvauskieltä käyttäen [OMG04], laadun varmistus sekä ohjelmiston muutosten hallinta. RUP-metodissa kukin ohjelmistoprojekti jaetaan neljään vaiheeseen, jotka koostuvat yhdestä tai useammasta iteraatiosta [Rat01]. Vuonna 1996 Kent Beck kehitti Chrysler C3-projektin yhteydessä myöhemmin hyvin tunnetuksi tulleen Extreme Programming-mallin [Bec99] peruseriaatteet, jotka painottivat testauslähtöisyyttä ja kommunikointia [LaB03, CBo05].

2.3.4 2000-luku

Vuonna 2001 Alan MacCormack raportoi tutkimuksesta, jossa tutkittiin 29:n ohjelmistoprojektin piirteitä ja pyrittiin näin päättämään, mitkä piirteet johtivat parhaaseen lopputulokseen [Mac01]. Tutkimuksesta kävi ilmi, että tärkeimmät projektin onnistumista tukevat asiat olivat ohjelmiston asteittainen, evolutionäärinen kehitys, uuden toiminnallisuuden jatkuva integrointi, kokenut projektiryhmä sekä investointi ohjelmiston arkkitehtuurin suunnitteluun.

Helmikuussa 2001 monien eri iteratiivisten ja evolutionääristen mallien asiantuntijat perustivat Agile Alliance:n, jonka tarkoituksena on levittää tietoa näistä malleista [Agi06]. Samalla näiden mallien yleisnimeksi muodostui termi *ketterät mallit* (agile methods) [LaB03]. Vuonna 2002 ilmestyi ensimmäinen tätä uutta nimeä kantava kirja [Coc02].

3 Testauslähtöisyyden pääperiaatteet

Uutta testauslähtöisessä ohjelmistokehityksessä on se, että testitapaukset eivät rajoitu pelkästään ohjelman oikeellisuuden tarkistamiseen, vaan ne ohjaavat myös ohjelmiston suunnittelua [JSa05]. Ohjelmistoa ei siis suunnitella etukäteen, vaan aina yksikkötestien vaatiman koodin toteutuksen jälkeen parannetaan koodin rakennetta (refaktoroidaan). Refaktorointi ei muuta ohjelmayksiköiden toiminnallisuutta, joten testien muutostarve refaktoroinnin yhteydessä pysyy pienenä. [Sho04].

Ohjelmistokehitys voidaan luokitella testauslähtöiseksi, jos se noudattaa tiettyjä periaatteita [Jef99]. Jo nimessä näkyvä periaate, testien kehitys ennen koodia, on tärkeä muttei suinkaan ainoa periaate, jota testauslähtöisen ohjelmistokehityksen tulisi noudattaa. Testien avulla voidaan myös tarkistaa, että kunkin iteraation tulos täyttää sille asetetut vaatimukset, toisin sanoen testaus toimii ohjelmiston validoinnin välineenä. Testeillä ohjataan myös ohjelmiston suunnittelua sekä mahdollistetaan ohjelmiston ylläpitovaiheessa tulleiden muutosten toteutus ja niiden verifiointi regressiotestauksen avulla, joten testit ohjaavat näin ohjelmiston kehitysprosessin kaikkia osavaiheita [JSa05].

3.1 Testit ennen koodia

*”Testauslähtöisessä ohjelmistokehityksessä automaattisesti suoritettavat yksikkötestit kirjoitetaan ennen ohjelmayksiköiden toteutusta lyhyiden iteraatioiden aikana” [JSa05]. Ohjelmayksikön määritelmä ei ole aina yksiselitteinen. Se voi olla yksittäinen metodi, luokka tai *luokkarypäs* (class cluster). Yleensä pyritään kuitenkin siihen, että yksikkö on helposti testattavissa muista yksiköistä riippumatta ja testattava toiminnallisuus on suhteellisen suppeaa. IEEE:n standardin [IEE86] mukaan ohjelmiston testausyksikkö (test unit) on *joukko tietokoneohjelman moduleita sekä niihin liittyviä ohjaustietoja, käyttöproseduureja sekä operatiivisia proseduureja joille pätee seuraavat ehdot:**

- Kaikki modulit ovat peräisin samasta ohjelmasta
- Vähintään yksi moduli uusien tai muuttuneiden modulien joukosta ei ole läpäissyt yksikkötestiä

- Modulien joukko yhdessä ohjaustietojen ja proseduurien kanssa muodostaa yhtenäisen testattavan objektin testausprosessissa

Testauslähtöisessä ohjelmistokehityksessä IEEE:n standardia ei voida kuitenkaan käyttää suoraan, sillä sen mukainen yksikkötestaus olettaa, että testattava ohjelmisto on olemassa. Testauslähtöisessä ohjelmistokehityksessä testattavan ohjelman yksikköä ei kuitenkaan ole vielä olemassa siinä vaiheessa, kun testejä suunnitellaan ja kirjoitetaan. IEEE:n määritelmää voitaisiinkin käyttää käänteisesti ja nähdä testausyksikkö suunniteltavien testien lopullisena tavoitteena.

3.2 Testit ohjaavat suunnittelua

Vaikka testauslähtöisyys nähdään usein pelkkänä ohjelmiston testausstrategiana, sen todellinen merkitys ohjelmistoprosessin kannalta on laajempi [JSa05, Ras03]. Ohjelmiston varsinaiseen testaukseen liittyvät periaatteet ohjaavat yksikkötestien suunnittelua ja toteutusta sekä ohjelmistoon tehtyjen muutosten aiheuttamien virheiden havainnointia eli regressiotestausta. Kuitenkin vähintään yhtä tärkeää on testauslähtöisyyden vaikutus ohjelmiston suunnitteluun ja kehitykseen [Amb03]. Yhtenä testauslähtöisyyden pääperiaatteena voidaan pitää sitä, että uutta toiminnallisuutta tarjoavaa koodia tulisi kirjoittaa vain silloin, kun jokin olemassa oleva yksikkötesti ei mene läpi. Toinen pääperiaate on, että koodin duplikointia pyritään ehkäisemään jatkuvasti refaktoroinnin avulla.

Ohjelmiston toiminnallisuutta kasvatetaan siis vain testeillä todetun tarpeen mukaisesti. Mahdollisia tulevia vaatimuksia ei pyritä ennustamaan eikä niihin varauduta. Tällainen todettuun tarpeeseen perustuva ohjelmiston astettainen kehitys voi vaikuttaa positiivisesti myös ohjelmiston suorituskykyyn [Fea02], sillä ohjelmistossa tapahtuva laskenta siirtyy lähelle sitä ohjelmiston osaa, missä laskentaa todella tarvitaan.

3.3 Refaktorointi

Refaktorointi tarkoittaa ohjelman lähdekoodin muuttamista siten, että ohjelman toiminnallisuus ei muutu [DBD04]. Refaktoroinnin tavoitteena on parantaa ohjelmiston ylläpidettävyyttä ja helpottaa tulevia muutostarpeita parantamalla koodin rakennetta, esimerkiksi yhdistelemällä duplikoitua koodia metodeiksi, vähentämällä luokkien välisiä riippuvuuksia, ulkoistamalla luokan julkiset metodit

rajapinnaksi jne.

Refaktorointi voidaan jakaa kolmeen osaan: ensin tutkitaan, milloin ohjelman koodin rakennetta tulisi parantaa, sitten päätetään, mitä eri tapoja on suorittaa refaktorointi ja lopuksi suoritetaan valitut toimenpiteet [KIA02, MTM03]. Refaktoroinnin takana on usein työkalutuki, joka tarjoaa apua refaktorointitoimenpiteiden suorittamiseen, mutta kaksi ensimmäistä vaihetta joudutaan kuitenkin useimmiten suorittamaan manuaalisesti.

Refaktoroinnin asema testauslähtöisessä ohjelmistokehityksessä on merkittävä [Bec99]. Kattavan yksikkötestikokoelman olemassaolo mahdollistaa laajatin refaktoroinnit poistamalla epävarmuustekijän muutoksien vaikutuksista ohjelmiston toiminnallisuuteen. Jos refaktoroinnin seurauksesta toiminnallisuus muuttuu, se huomataan heti yksikkötestauksessa. Esimerkiksi Extreme programming -mallissa painotetaan, että koodin rakenne syntyy nimenomaan testauslähtöisen suunnittelun ja sen jälkeisen refaktoroinnin kautta, eikä etukäteen tapahtuvaa rakenteen suunnittelua tarvita.

3.4 Regressiotestaus

Regressiotestaus (regression testing) on *testausprosessi jota käytetään sen jälkeen, kun ohjelmistoon tehdään muutoksia* [LWh89]. Käytännössä regressiotestaus tarkoittaa olemassa olevan yksikkötestijoukon suorittamista muuttuneen ohjelmiston oikeellisuuden varmistamiseksi. Ohjelmistoon voi kohdistua monenlaisia muutoksia, alkaen pienistä virhekorjauksista päättyen aina osajärjestelmien lisäyksiin tai poistoihin. Regressiotestauksella pyritään varmistamaan, että muuttuneet osat toimivat oikein ja että uusi toiminnallisuus ei aiheuta konflikteja vanhan toiminnallisuuden kanssa [LWh89]. Regressiotestaus tuo myös varmuutta ohjelmiston integrointiin, kun kattava joukko testejä ajetaan aina automaattisesti integrointivaiheessa. Tämä voi kuitenkin aiheuttaa sen, että täydellisen testijoukon ajo kestää pitkään, ja siksi testien suoritusajaksi on kiinnitettävä huomiota [Ras03].

Regressiotestaus on osa testauslähtöisiä ohjelmistoprosesseja [EMT05]. Se helpottaa uuden toiminnallisuuden aiheuttamien konfliktien löytämistä ja lyhentää näin ohjelmistosta saatavaa palautesykliä. Regressiotestaus toimii siis eräänlaisena turvaverkkona ohjelmiston ylläpitovaiheessa, jolloin ohjelmistoon lisättäviä uusia ominaisuuksia tai siihen tehtyjä korjauksia sekä niiden yhteisvaikutuksia olemassa olevan toiminnallisuuden kanssa pystytään testaamaan [JSa05].

3.5 Testit validointivälineenä

Ohjelmiston modulien eli yksiköiden toiminnallisuuden testaaminen yksikkötestien avulla tarjoaa kehittäjille tietoa ohjelmayksiköiden oikeellisuudesta [Jef99]. Se myös tarjoaa puitteet ohjelmiston ylläpitäjille ja jatkokehittäjille ohjelmiston laajentamiseen tai ohjelmiston toiminnallisuuden tai rakenteen muuttamiseen regressiotestauksen tukemana. Testauslähtöisessä ohjelmistokehityksessä voidaan yksikkötestien lisäksi käyttää myös ns. *funktionaalisia*, korkeamman tason testejä (functional tests), jotka testaavat kunkin iteraatiokierroksen tuottamia uusia tai muuttuneita toimintoja. Funktionaaliset testit siis testaavat, että kunkin iteraation vaatimukset ovat toteutettu ja että ne toimivat halutulla tavalla. Usein yksittäisen vaatimuksen funktionaalisten testien suunnittelu paljastaa vaatimuksen mahdollisia puutteita, kuten moniselitteisiä tai epäselviä käsitteitä, joita on käytetty vaatimuksessa [Gra02].

Funktionaaliset testit tehdään pääasiassa vaatimuksen tekijää eli asiakasta sekä projektin johtoa varten [Jef99]. Asiakas pystyy varmistamaan, että hänen vaatimuksensa täyttyvät, ja projektin johto pystyy seuraamaan paremmin projektin etenemistä sekä tunnistamaan mahdolliset ongelmakohdat ajoissa. Koska funktionaaliset testit eivät ole tarkoitettu pelkästään ohjelmoijia varten, niiden tulisi olla paitsi yksiselitteisiä ja automaattisia (kuten yksikkötestienkin) niin myös sen verran selkeitä, että kaikki kiinnostuneet sidosryhmät (kuten asiakas ja projektin johto) pystyisivät ymmärtämään niitä. Sen lisäksi niiden tulisi olla luonnollisesti kaikkien kiinnostuneiden sidosryhmien saatavilla.

4 Testaustehokkuus ja kustannukset

Pelkkä testien olemassaolo ei kerro mitään siitä, kuinka hyvin testit löytävät virheitä ohjelmakoodista [Mur94]. Mikäli tätä testien laatutekijää ei tarkkailla, suuri määrä yksikkötestejä voi luoda virheellisen turvallisuuden tunteen, mikä voi vaikuttaa koko projektin kulkuun negatiivisesti. Testauslähtöisessä ohjelmistokehityksessä testeillä on suuri vastuu niin ohjelmiston rakenteen luomisessa kuin myös ohjelmiston validointivälineenä. Ohjelmiston täydellinen testaaminen ei ole mahdollista eikä siihen pyrkiminen ole näin ollen perusteltua [Bac98].

Testaukseen liittyvät tekniikat, prosessit ja työkalut eivät itsessään takaa tehokasta testausta tai testattavan ohjelmiston korkeaa laatua. Testauksen tehokkuuteen

vaikuttavat teknisten seikkojen lisäksi kulttuurilliset, taloudelliset sekä hallinnolliset tekijät [Mur94, Bac98]. Kuitenkin ohjelmistojen korkeaa laatua ja asiakkaan tyytyväisyyttä ohjelmistotuotteeseen painotetaan yhä enemmän, joten on mielekästä tutkia testaukseen liittyviä haasteita sekä testauksen aiheuttamiin kustannuksiin liittyviä tekijöitä.

Tässä luvussa tutkitaan testien tehokasta käyttöä ohjelmiston validointivälineenä, sekä testaukseen liittyviä haasteita ja niiden mahdollisia ratkaisuvaihtoehtoja. Tarkastelussa pyritään ottamaan huomioon myös taloudellinen näkökulma. Tässä käsittelyssä testien tehokkuudella tarkoitetaan testien kykyä löytää ohjelmistosta virheitä suhteessa testien suunnitteluun, toteutukseen ja suoritukseen vaadittuihin resursseihin.

4.1 Mustalaatikkotestaus

Mustalaatikkotestauksella (black-box testing) tarkoitetaan testien suunnittelutapaa, jossa testattavan ohjelmayksikön rakennetta ei tunneta tai sitä ei huomioida lainkaan [Hua75]. Testattava yksikkö on testaajalle kuin musta laatikko, joka tarjoaa annetuilla syötteillä yksikön vaatimusten mukaisia tulosteita. Kaikkien mahdollisten syötteiden ja niiden yhdistelmien testaaminen ei ole mahdollista, sillä niiden määrä on kaikissa mielenkiintoisissa tapauksissa käytännössä ääretön. Siksi mustalaatikkotestauksessa testisuunnittelussa pyritään pienentämään testattavien syötteiden määrää [Rei97]. Mustalaatikkotestauksessa yleisesti käytetyt testaustekniikat ovat ekvivalenssiositus sekä arvoalueanalyysi. Näiden lisäksi käytetään mm. tilasiirtymätestausta, syy-seurausanalyysiä sekä syötteen syntaksiin perustuvaa syntaksitestausta [BCS01].

Ekvivalenssiosituksessa testattavan yksikön syötteet ja tulosteet pyritään jakamaan arvo- tai arvovälijoukosta koostuviin ekvivalenssiluokkiin siten, että kaikki samaan ekvivalenssiluokkaan kuuluvat syötteet aiheuttavat saman toiminnallisuuden suorituksen testattavassa yksikössä [BCS01]. Näin ollen yksikköä voidaan testata siten, että kustakin syötteiden ekvivalenssiluokasta valitaan yksi syöte testausta varten. Tulosteiden ekvivalenssiluokkia voidaan käyttää virheellisiin luokkiin kuuluvien tulosteiden havainnoinnissa. Ekvivalenssiosituksen kattavuus saadaan mitattua jakamalla testattujen ekvivalenssiluokkien määrä ekvivalenssiluokkien yhteismäärällä.

Arvoalueanalyysissä syötteet ja tulosteet jaetaan yhtenäisiin arvoaluejoukkoihin,

joille voidaan määritellä yksikäsitteiset raja-arvot [BCS01]. Kaikkien samaan arvoalueeseen kuuluvien syötteiden tulisi suorittaa testattavaa yksiköä samalla tavalla. Kunkin arvoalueen raja-arvo tuottaa kolme testattavaa syötettä: raja-arvolla oleva arvo sekä raja-arvon molemmilla puolilla lähellä olevat arvot. Lähekkäisten arvojen etäisyys pyritään pitämään mahdollisimman pienenä, esim. kokonaislukujen tapauksessa se voi olla 1. Arvoalueanalyysin kattavuus saadaan mitattua jakamalla testattujen arvoalueiden raja-arvojen määrä arvoalueiden raja-arvojen yhteismäärällä. Koska arvoalueanalyysi tuottaa vähintään yhden kappaleen kuhunkin arvoalueeseen kuuluvia syötteitä, se täyttää myös ekvivalenssisuosituksen vaatimukset.

4.2 Lasilaatikkotestaus

Lasilaatikkotestaus (white-box testing, glass-box testing) eroaa mustalaatikkotestauksesta siten, että testattavan ohjelmayksikön sisäinen rakenne on tunnettu ja testit suunnitellaan suoraan testattavan koodin rakenteen perusteella [Mis03, CPT00]. Koska lasilaatikkotestaus perustuu ohjelmayksikön varsinaiseen koodiin, sen avulla löydetään monia sellaisia virheitä, joita ei ole mahdollista löytää mustalaatikkotestauksella. Syy siihen on se, että mustalaatikkotestauksessa syötteiden arvovälit tai ekvivalenssiluokat perustuvat testattavan ohjelmayksikön dokumentoituun tai dokumentoimattomaan määrittelyyn. Ohjelmayksikön määrittely ei taas välttämättä kerro sitä, miten kyseinen ohjelmayksikkö käyttäytyy - se kertoo vain sen, miten sen tulisi käyttäytyä.

Yleisimmät lasilaatikkotestauksen tekniikat ovat lausetestaus sekä haaraumatestaus [BCS01]. Näiden lisäksi käytetään myös ehtotestausta, moniehtotestausta, tietovuotestausta, silmukkatestausta ja polkutestausta, jotka jätetään kuitenkin tämän tutkielman ulkopuolelle. Kattava kuvaus näistä tekniikoista löytyy mm. British Computer Societyn tekemästä standardista "Standard for Software Component Testing" [BCS01]. Kaikille tekniikoille on yhteistä se, että ne perustuvat johonkin testattavan ohjelmayksikön lähdekoodista luotuun malliin, jonka avulla tarkkaillaan koodin suoritusta erilaisilla syötteillä.

Lausetestaus on yksinkertaisin lasilaatikkotestaustekniikka, jossa testattavasta lähdekoodista luodaan malli, jossa jokainen lähdekoodin lause joko suoritetaan tai jätetään suorittamatta [BCS01]. Tavoitteena on suunnitella testi, joka tietyllä joukolla syötteitä aiheuttaisi mahdollisimman monen lähdekoodin lauseen suori-

tuksen. Lausekattavuus lasketaan jakamalla suoritettujen lauseiden määrä koodin lauseiden kokonaismäärällä. Vaikka jokaisen lauseen suorittaminen ei takaa sitä, että ohjelma toimii oikein, näinkin yksinkertaisen kattavuuskriteerin ja ohjelmistosta myöhemmin löydettyjen virheiden määrän välillä on selvä korrelaatio [DHK93].

Haaraumatestauksessa testattavasta lähdekoodista luodaan malli, joka kuvaa koodin haaraumia eli suoritusta ohjaavia päätöksiä, kuten ehtolauseita tai silmukoita. Haaraumatestauksella pystytään löytämään sellaisia virheitä, jotka eivät tule esille täydelliselläkään lausekattavuudella, esim. tilanteessa, jossa if-ehdolla ei ole else-haaraa. Tällöin täydellinen lausekattavuus saavutetaan, kun testitapaus aiheuttaa if-ehdon toteutumisen, ja koodin toinen haarauma (if-ehdon toteutumattomuus) jää testaamatta. Testien tavoitteena on suorittaa mahdollisimman paljon koodin haaraumia. Haaraumakattavuus saadaan jakamalla suoritettujen haaraumien määrä koodin haaraumien kokonaismäärällä.

4.3 Kattavuusmittareiden käyttö testien tehokkuuden tarkkailussa

Musta- ja lasilaatikkotestauksen tehokkuutta voidaan arvioida niistä saatavilla kattavuusmittareilla. Kattavuusmittari on prosentuaalinen luku, joka kertoo kuinka suuren osan kaikista kyseisen testaustavan mukaisista ilmentymistä kyseisen testitapaus kattaa [Mar99, Mis03]. Jokainen kattavuusmittari mittaa testien kattavuutta vain yhdestä näkökulmasta, ja siksi kattavuutta tulee aina arvioida suhteessa sen tuottaneeseen mittariin - koko ohjelmiston täydellistä kattavuutta ei ole mahdollista saavuttaa. Edellä esitettyjen kattavuusmittareiden lisäksi on olemassa lukuisia muitakin mittareita, mm. luokkakattavuus, osajärjestelmä-kattavuus, virhetilannekattavuus jne. Cem Kaner [Kan96] listasi artikkelissaan "Software negligence and testing coverage" yhteensä 101 erilaista kattavuusmittaria - jokainen testauksessa oleva ohjelmisto voi kuitenkin vaatia yhä uusia kattavuusmittareita.

Kattavuusmittareita käytetään usein väärin [Mar99]. Mikäli jonkin testitapausten kattavuus on tietyn kattavuusmittarin mukaan puutteellinen, testien kirjoittajan tulisi analysoida nykyisiä testitapauksia ja miettiä, mistä puutteellinen kattavuus johtuu - onko jokin testattavan ohjelman toiminto tai poikkeustilanne jäänyt huomioimatta. Heikon kattavuuden ei tulisi aiheuttaa sitä, että testit suunniteltaisiin

tarjoamaan mahdollisimman korkeita kattavuusmittareita ilman alla olevan ohjelman huomioonottamista. Kattavuusmittarit toimivat myös suhteellisen matalalla tasolla - niiden avulla on vaikeaa mitata mm. monen eri alijärjestelmän kattavaan ominaisuuteen liittyvää testiä. Testien laaduntarkkailussa on otettava huomioon myös testitapausten päällekkäisyydet - samaa toimintoa samoilla syötteillä testaavat tapaukset nostavat testauksen resurssivaatimuksia, mutta eivät paranna testien tehokkuutta [Bac98].

Kattavuusmittareiden mahdollinen väärinkäyttö ei kuitenkaan tarkoita sitä, että niiden käyttöä tulisi välttää tai että niiden käyttö olisi tehotonta [Mar99]. Lukuisissa empiirisissä tutkimuksissa on todettu, että eri kattavuusmittareiden arvoilla ja ohjelmiston luotettavuudella on suora yhteys [DFG95, MNB94, CLW95]. Ohjelmiston luotettavuudella tarkoitetaan todennäköisyyttä, että ohjelmisto ei aiheuta järjestelmässä vikatilannetta, kun ohjelmistoa käytetään tietyinä aikajaksona tietyissä olosuhteissa [IEE88]. Todennäköisyys on syötteiden, järjestelmän käytön sekä järjestelmässä olevien virheiden funktio. Syötteistä riippuen järjestelmässä olevat virheet voidaan havaita.

4.4 Testejä vai esimerkkejä?

Joidenkin käsitysten mukaan testauslähtöisen ohjelmistokehityksen nimitys on epätarkka, sillä ennen ohjelmayksiköiden toteutusta kirjoitetut testit eivät ole mustalaatikkotestejä eivätkä lasilaatikkotestejä [Mar03, Jef06]. Mustalaatikkotestauksessa oletetaan, että ohjelmayksikön sisäisestä rakenteesta ei tiedetä mitään. Kuitenkin testauslähtöisessä ohjelmistokehityksessä kehittäjä itse kirjoittaa sekä testin että ohjelmayksikön toteutuksen, jolloin hänellä on jonkinlainen kuva siitä, millainen yksikön sisäinen rakenne tulee olemaan. Tämä seikka korostuu, kun samaan ohjelmayksikköön ryhdytään lisäämään toiminnallisuutta, jolloin kehittäjä näkee olemassa olevan koodin uuden testin kirjoituksen aikana.

Myöskään lasilaatikkotestauksen ehdot eivät täyty kokonaan, sillä ennen yksikköjen toteutusta kirjoitetuilla testeillä ei pyritä löytämään mahdollisia koodin staattisesta tai dynaamisesta rakenteesta johtuvia virheitä, vaan testien tehtävänä on lähinnä kertoa, miten toteutettavan yksikön tulee toimia [Mar03, Bec99]. Siksi jotkut ohjelmistotuotannon ammattilaiset ovatkin ehdottaneet, että yksikköjen toteutusta ohjaavia testejä kutsuttaisiin esimerkkitapauksiksi ja koko testauslähtöisyys uudelleennimettäisiin esimerkkilähtöisyydeksi [Mar03, Jef06]. Näitä

esimerkkitapauksia voisi sitten täydentää musta- ja lasilaatikkotesteillä, jos riittämätön koodikattavuus huomataan.

Vaikka kyse onkin yhden termin vaihtamisesta toiseen, se vaikuttaa projektiryhmän sisäiseen ja ulkoiseen kommunikointiin [Mar03]. Monessa tilanteessa voi olla helpompaa puhua asiakkaan kanssa esimerkkitapauksista kuin testitapauksista. Jos testitapauksen pääasiallinen tarkoitus on virheiden osoittamisen sijasta puuttuvan toiminnallisuuden osoittaminen, sen nimeäminen esimerkkitapaukseksi voisi olla myös termistöä selkiinnyttävä tekijä [Jef06]. Nimenä esimerkkilähtöisyys ei ole ainakaan vielä saanut laajaa hyväksyntää, ja sekä akateemisessa maailmassa että yritysmaailmassa puhutaan edelleenkin testauslähtöisyydestä.

Keskustelu testauslähtöisen ohjelmistokehityksen uudelleennimeämisestä esimerkiksi lähtöiseksi kehitykseksi perustuu ajatukselle, että hyvä esimerkkitapaus tarjoaa jo itsessään kohtuullisia kattavuusarvoja, joten myöhemmin aidoilla testeillä korjattavat puutteet kattavuudessa jäävät vähäisiksi [Mar99, Mar03]. *Skenaariopohjainen yksikkötestaus* (scenario-based unit testing, myös statistical software testing) lähestyy testausta samankaltaisista lähtökohdista - testattavalle toiminnolle generoidaan sen käyttökuvauksen perusteella joukko testiskenaarioita, jotka kuvaavat toiminnon käyttöä sen todellisessa käyttöympäristössä todellisilla syötteillä [KHG02]. Eräässä empiirisessä tutkimuksessa [KHG02] tällainen lähestymistapa tarjosi 100 prosentin lausekattavuuden sekä vähintään 90 prosentin ehtokattavuuden. Samassa tutkimuksessa myös todettiin, että pelkkien kattavuusmittareiden käyttöön perustuva testaus ei olisi löytänyt kaikkia niitä virheitä, jotka skenaariopohjaiset yksikkötestit löysivät.

4.5 Kulttuurin ja tekniikan haasteet

Testaukseen on perinteisesti suhtauduttu toimenpiteenä, joka suoritetaan projektin viimeisessä vaiheessa [HMe02, Mur94]. Kuten aikaisemmassa tarkastelussa on todettu, testauksen jättäminen projektin loppuun aiheuttaa herkästi sen, että testaus ohitetaan kokonaan, jos aikataulupaineet alkavat kasvaa. Pelkät muutokset ohjelmistotuotantoprosessissa eivät kuitenkaan riitä, vaan testausta on tutkittava aktiviteettina, joka koostuu testitapauksista, työkaluista, testaus suunnitelmasta, tekniikoista, prosesseista, standardeista sekä ihmisistä ja organisaatiosta. Ihmiset ja organisaatiokulttuuri ovat lopulta tärkein tekijä minkä tahansa järjestelmän rakentamisessa. Jotta testauksen asemaa sekä tehokkuutta voitaisiin pa-

rantaa, kehittäjien asenteiden tulee muuttua. Testauksella tulisi pyrkiä parantamaan ohjelmistotuotteen käyttäjän eli asiakkaan tyytyväisyyttä sen sijaan että keskityttäisiin pelkästään ohjelmistovirheiden etsimiseen. Toisin sanoen on siirryttävä asiakkaan tyytymättömyyden minimoinnista tyytyväisyyden maksimointiin.

Useissa organisaatioissa testaukseen suhtaudutaan ylimääräisenä toimintona, jota ei ole välttämättä pakko suorittaa - ohjelmiston saaminen asiakkaalle kunnossa missä hyvänsä voidaan nähdä tärkeämpänä kuin ohjelmiston luotettavuus tai riittävä virheettömyys [Mur94]. Testaus nähdään usein toisarvoisena toimintana, joka voidaan antaa kokemattomamman projektiryhmän jäsenen hoidettavaksi, vaikka käytännössä toiminta vaatii tietoa, taitoa sekä motivaatiota aivan yhtä lailla kuin muutkin ohjelmistotyön vaiheet. Tämä asenne vaikuttaa myös budjettipäätöksiin - testaukselle varattavat resurssit ovat usein liian rajalliset. Testauslähtöisessä ohjelmistokehityksessä yksikkötestit kirjoittaa ohjelmoija itse, joten testaustyön arvostus säilyy helpommin tarvittavalla tasolla, ja delegointi kokemattomille työntekijöille estyy. Varsinkin ketterissä prosessimalleissa suositetaan asiantuntijoita, jotka eivät keskity vain yhteen ohjelmistotyön osa-alueeseen (generalizing specialists) [Amb06].

Ohjelmistosta löydetyt virheet ja puutteet tulee dokumentoida kehittäjien toimesta, jotta niihin voitaisiin kehittää myöhemmin ratkaisuja, ja jotta ne voitaisiin ottaa huomioon seuraavissa projekteissa [Mur94]. Testauslähtöisessä ohjelmistokehityksessä löydetyt virheet ja puutteet on luonnollista dokumentoida testitapauksilla, jotka osoittavat virheen olemassaolon [Bec99]. Organisaatiokulttuurin ilmapäärin tulisi olla sellaista, että virheistä ja puutteista vastuussa olevat kehittäjät eivät pelkäisi joutuvansa syytettyjen asemaan ja jättäisi siksi virheitä ja puutteita dokumentoimatta. Organisaatiokulttuurin tulisi rohkaista kaikkia organisaation jäseniä keskustelemaan löydetyistä virheistä ja puutteista sekä niiden ratkaisu- vaihtoehdoista. Samalla organisaation johdon tulee pitää mielessä myös ohjelmistotestauksen rajoitteet - testaus ei voi olla koskaan täydellistä, ja siksi kehittäjiltä tai muilta organisaation jäseniltä ei tule vaatia asioita, joita on mahdotonta toteuttaa [Bac98].

Ohjelmistotuotteen käyttäjän eli asiakkaan tulisi olla mukana ohjelmiston testauksessa [HCo01, Mur94, Bec99]. Testauslähtöisessä ohjelmistokehityksessä asiakkaan rooli koskee lähinnä funktionaalisia testejä, jotka toimivat ohjelmiston validointivälineenä. Asiakkaan rooli voi vaihdella passiivisemmasta konsultin roo-

lista aktiivisempaan funktionaalisten testien suunnittelijan ja kirjoittajan rooliin. Asiakkaalla on myös tärkeä rooli testausprosessia kehittävän palautteen antajana. Kaikki asiakastyytyväisyyteen liittyvät seikat tulee analysoida ja dokumentoida, ja niiden pohjalta on pyrittävä tuomaan testausprosessiin tarvittavia muutoksia.

Organisaatiokulttuuriin ja prosesseihin liittyvien haasteiden lisäksi testaukseen liittyy myös useita puhtaasti teknisiä haasteita, jotka voivat tehdä testauksesta ajankäytöllisesti ja siten myös taloudellisesti haastavaa [HMe02, Bac98]. Yksi tärkeimmistä ohjelmateknisistä haasteista, joka ei riipu testattavan ohjelmiston tyypistä, on saada ohjelmiston testattavuus riittävän korkealle tasolle [HMe02, Bec99]. Testauslähtöisessä ohjelmistokehityksessä lisättävän koodin testattavuus pysyy riittävän korkeana itsestään, sillä se tuotetaan vastaamaan testien tarpeisiin. Aina ohjelmistoja ei kuitenkaan lähdetä kirjoittamaan puhtaalta pöydältä, ja ns. legacy-koodin testattavuus voi osoittautua heikoksi. Tällöin legacy-koodin testattavuutta pyritään parantamaan refaktoroinnin avulla asteittain. Myös jotkut tekniset ratkaisut saattavat hankaloittaa ohjelmiston testausta, kuten ns. container-pohjaiset ratkaisut. Näissä ratkaisuissa ohjelmisto toimii jonkin muun ohjelmiston eli containerin sisällä ja riippuu containerin tarjoamista palveluista, esimerkiksi J2EE-ohjelmopalvelimen sisällä toimivat ohjelmistot [HMe02, GBC01]. Tällaisten ohjelmistojen eristys containerista tehokkaan testauksen järjestämiseksi on hankalaa lukuisten toiminnallisten riippuvuuksien vuoksi.

4.6 Automatisointi

Kuten aikaisemmin on tullut esille, testausta pidetään aikaa vievänä työnä, jolle varataan usein liian vähän aikaa ja resursseja [Ram04, DTB03, TRa99]. Testauksen eri vaiheiden automatisoinnin avulla on mahdollista tehostaa testausta ja säästää näin resurssien säästöä. Testauksen automatisointi ei kuitenkaan ole aina kannattavaa, sillä jokaiseen automatisointitoimenpiteeseen liittyy potentiaalisia ohjelmistohankintoja, koulutusta ja ylläpitoa. Automatisointiin vaikuttaa myös itse ohjelmiston testattavuuden taso sekä syötteiden generointiin ja tulosteiden validointiin vaaditun logiikan monimutkaisuus. Jos ohjelmiston testattavuus on heikko tai sen syötteiden automaattinen generointi on hyvin vaikeaa, testauksen manuaalinen suoritus voi tulla taloudellisesti kannattavammaksi kuin suorituksen automatisointi [Ram04, Mar98].

Automatisoidulla testauksella voi olla myös rajallinen elinikä: jos jokin automatisoitu testauksen työvaihe ei enää toimi sen jälkeen, kun ohjelmisto muuttuu, niin tällaisen muuttumisen todennäköisyys vaikuttaa suoraan automatisoinnin kannattavuuteen, sillä automatisoidun työvaiheen korjauksen kustannus vastaa saman työvaiheen manuaalisen suorituksen kustannusta [Mar98]. Myös testaukseen liittyvä aikaväli ja toistojen määrä on otettava huomioon - mitä useampia kertoja tiettyä testauksen työvaihetta suoritetaan, sitä kannattavamaksi sen automatisointi muuttuu. Jotkut testauksen osa-alueet, kuten rasiustestaus, ovatkin lähes aina kannattavampia automatisoituina kuin manuaalisesti suoritettuina.

Automatisoitavissa olevia testauksen työvaiheita ovat mm. testitapausten suoritus [Jun06, Ocu06], testitapausten generointi koodista [DTB03, Mun88], testitapausten generointi koodista muodostetusta erillisestä mallista [TRa99] tai koodiin syötetyistä väitelauseista eli assertioista [KoA96], testitapausten suorittaman koodin kulun seuranta (kattavuusmittaus) [Cob06, Qui03, Gco01] sekä testitapausten, -datan, -tulosten ja -raporttien hallinta [BTP01]. Työkalutuki on tyypillisesti ohjelmointikielisisidonnaista, ja niin avoimen lähdekoodin kuin myös kaupallisten työkalujen saatavuus vaihtelee suuresti kielestä toiseen.

Testitapausten suorituksen automatisoinnin kustannusta voidaan arvioida mm. käyttäen Brian Marickin [Mar98] tekemiä johtopäätöksiä: testitapausten automatisoinnin kustannus määräytyy sen perusteella, kuinka monta manuaalista testiä samalla kustannuksella olisi voitu suorittaa ja kuinka monta virhettä tämän johdosta jäi potentiaalisesti huomaamatta. Automatisoidun testitapausten arvoon vaikuttaa Marickin mukaan myös se, kuinka paljon uusia virheitä uudelleen suoritettu automatisoitu testitapausta löytää. Uusia testitapausta generoivan työkalun konfigurointi ja käyttöönotto tai oman generaattorin toteutus voi osoittautua joissakin tapauksissa halvemmaksi kuin vastaavien testitapausten kirjoittaminen käsin [BMu83]. Testitapausten generaattori toimii myös pitempiaikaisena sijoituksena kuin käsin kirjoitetut testitapaukset, koska sillä pystytään generoimaan myös tuotteen tuleviin versioihin liittyviä testitapausta.

Testauslähtöisessä ohjelmistokehityksessä testaukselle on pakko varata aikaa ja resursseja, sillä ilman sitä ei synny tuotantokoodiakaan. Kuitenkin aikaisempi tarkastelu on osoittanut, että ennen tuotantokoodin kirjoitusta tehtyjä testitapausta on myöhemmin usein täydennettävä uusilla testitapausta, jotta ohjelmisto saadaan testattua kattavasti, ja testitapausten generointi on yksi potentiaalisesti automatisoitavissa oleva työvaihe. Myös regressiotestaukseen liittyy toimintojen

automatisointia, kuten koko testitapausjoukon suoritus ja raporttien generointi, joten automatisoinnilla ja siihen liittyvillä työkaluilla on tärkeä sija testauslähtöisessä ohjelmistokehityksessä.

5 Tapaus Alma

Helsingin Yliopisto aloitti portaalihankkeen [Por06] vuonna 2002. Hankkeen tavoitteina oli tuoda käyttöön helppokäyttöinen työ- ja viestintäväline yliopiston henkilöstölle ja opiskelijoille, mahdollistaa hallittu verkkosisällön ja verkkopalvelujen tuottaminen sekä tukea yliopiston yhtenäisemmän ilmeen edistämistä. Portaalihankkeen piirissä tuotettiin vuosina 2002 - 2006 julkaisuvälineitä tarjoava intranet-järjestelmä Alma, päivitettiin yliopiston verkkoilmettä, järjestettiin koulutustilaisuuksia sekä tuotettiin verkkosisältöä niin sisäiseen kuin ulkoiseenkin käyttöön. Tapaus Alma on otettu mukaan tähän tutkielmaan, koska se tarjoaa lukuisia käytännön esimerkkejä testaukseen sekä sen automatisointiin liittyvistä haasteista ja ketterien prosessimallien käytöstä keskisuudessa ohjelmistotuotantoprojektissa. Tutkielmassa käytetyt tiedot perustuvat pääosin portaalihankkeen projektipäällikkö Maikki Heikkisen sekä testaja Virve Väyrysen haastatteluun sekä projektin julkisiin dokumentteihin [Por06]. Tämän lisäksi on käytetty julkistamatonta järjestelmätestausdokumenttia [Kek05], joka on sijoitettu julkisesti saataville projektipäällikön luvalla.

5.1 Prosessimallin valinta

Projektin elinkaaren aikana prosessimalleista kokeiltiin mm. Extreme Programming-mallia ja adhoc-lähestymistapaa, mutta lopulta päädyttiin Scrumiin [Bee00, RJa00]. Scrum on ketterä prosessimalli, joka perustuu peräkkäisiin, noin kuukauden mittaisiin työvaiheisiin eli *sprintteihin*. Projektiin kuuluvat tehtävät järjestetään ns. *backlogiin*, joka priorisoidaan ja josta valitaan joukko tehtäviä kuhunkin sprinttiin. Projektiryhmään kuuluu *Scrum Master*, joka johtaa päivittäisiä ryhmäpalaverieja, pitää kirjaa niissä tehdyistä päätöksistä sekä toimii näiden palaverien puheenjohtajana. Scrum Master myös vastaa backlogiin kuuluvien tehtävien priorisoinnista yhdessä asiakkaan kanssa sekä projektiryhmän yleisestä edistymisestä. Projektin aikana käytettiin myös pariohjelmointia [Bec99], jossa kaksi ohjelmoijaa työskentelevät fyysisesti saman tietokoneen ääressä.

Prosessimallista saatiin myönteisiä kokemuksia niin projektiryhmän kuin asiakkaan osalta. Projektin aikataulu piti kunkin sprintin osalta, vaikkakin Scrumin periaatteiden mukaisesti joitakin ominaisuuksia saatettiin joutua jättämään pois meneillään olevasta sprintistä mm. kiireellistä korjausta vaativien virheiden löytyessä järjestelmästä. Asiakas on myös satunnaisesti vaatinut mm. yksityiskohtaisempia suunnitteludokumentteja kuin mitä prosessimalli edellyttää. Projektin hallintaan käytettiin Jira -työkalua [Jir06], johon kirjattiin kuhunkin sprinttiin kuuluvat tehtävät sekä hallittiin virheraportteja ja tehtävien delegointia eri vastuuhenkilöille.

5.2 Testaus ja automatisointi

Projektihankkeen aikana ei harjoitettu varsinaista testauslähtöistä ohjelmistokehitystä, mutta kriittisille komponenteille kirjoitettiin yksikkötestejä komponenttien kehityksen yhteydessä. Muilta osin yksikkötestejä ei kirjoitettu. Koko ohjelmisto testattiin järjestelmätestauksessa, joka järjestettiin jokaisen sprintin loppuvaiheessa, ennen ohjelmiston julkaisua. Järjestelmätestaus koostui savutestauksesta, jossa suhteellisen korkean tason testeillä todettiin ohjelmiston perustoinnallisuuden olemassaolo sekä havaittiin pahimmat virheet. Savutestauksen jälkeen kyseisessä sprintissä lisätyt uudet toiminnot testattiin suorittamalla yksityiskohtaisempia testitapauksia, jonka jälkeen löydetyt virheet kirjattiin Jira-järjestelmään. Kun kaikki näin löydetyt virheet olivat korjattu ja korjaukset validoitu, koko järjestelmälle suoritettiin regressiotestaus, jossa suoritettiin kaikki savutestit. Kunkin sprintin loppuvaiheen testaukseen pyrittiin varaamaan noin kolme henkilötyöpäivää siten, että yksi päivä meni savutestaukseen, yksi päivä uusien toimintojen tarkempaan testaukseen ja virhekorjausten validointiin sekä yksi päivä regressiotestaukseen.

Savutestit suoritettiin käsityönä, sillä ne käsittivät järjestelmän käyttöä web-selaimessa toimivan käyttöliittymän kautta. Näin ollen varsinkin järjestelmätestauksen ensimmäinen ja viimeinen vaihe - savutestaus ja regressiotestaus - pitivät sisällään paljon mekaanista, toistuvaa käsityötä. Regressiotestaukseen kuuluvien testitapausten joukon koko kasvoi jokaisen sprintin myötä, jolloin myös samantapaisina toistettujen manuaalisesti suoritettavien testitapausten määrä kasvoi. Koko järjestelmän kattava regressiotestaus saattoi kasvaa niin suureksi, ettei sitä pystytty aikataulusyistä toteuttamaan. Tämä kirjattiin projektin dokumentaatioon yhtenä testausprosessin riskinä, ja ratkaisuna ehdotettiin erillisen regressiotes-

tausprojektin järjestämistä tarpeen mukaan. Testitapausten suorittaman ohjelmakoodin kulun seuranta eli kattavuusmittausta ei projektissa käytetty testaukseen allokoitujen rajallisten resurssien vuoksi.

Manuaalisesti suoritettavien savutestien joukon kasvava koko sekä kasvavan osajoukon säännöllinen toistuvuus puoltavat regressiotestausvaiheen automatisointia. Toisaalta savutestien suoritus web-selaimessa toimivan käyttöliittymän kautta nostaa automatisoinnin potentiaalisia kustannuksia, sillä käyttöliittymän muuttuvuus on suuri ja tämä vaikeuttaa testeissä tarvittavien syötteiden automaattista generointia ja tulosten automaattista validointia. Käyttöliittymän muuttuminen voi myös rikkoo helposti testitapauksia, jolloin testitapausten korjaamiseen tarvittavat kustannukset laskevat automatisoinnista saatavaa hyötyä.

Savutestejä pyrittiin automatisoimaan käyttäen Selenium-työkalua [Sel06], joka on web-selaimessa toimiva ohjelmisto. Selenium on kehitetty web-käyttöliittymää käyttävien testitapausten automaattiseen suoritukseen ja se toimii suoraan testajan web-selaimessa. Työkalun käyttö koettiin kuitenkin työlääksi ja aikaa vieväksi, sillä testattavan ohjelmiston käyttöliittymässä oleva JavaScript-koodi aiheutti ongelmia testitapausten automaattisessa suorittamisessa. Koska JavaScript-koodi oli oleellinen osa koko ohjelmiston käyttöliittymää, automatisoinnista päätettiin luopua. Seleniumin lisäksi evaluoituihin työkaluihin kuului JWebUnit-ohjelmisto [Jwe06], mutta sekään ei tarjonnut ratkaisua koettuihin ongelmiin. Automaattisten työkalujen tarkkuus koettiin myös joskus riittämättömäksi.

Ongelmista huolimatta testauksen automatisointia käytettiin muilla testauksen osa-alueilla. Ohjelmiston toiminnan kannalta kriittisten komponenttien yksikkötestit suoritettiin automaattisesti JUnit-työkalun [Jun06] avulla. Rasitustestaus järjestettiin käyttäen JMeter-työkalua [Jme06], mutta siihen ei oltu täysin tyytyväisiä. Rasitustestaus on kuitenkin hyvin vahvasti automatisointia puoltava testauksen osa-alue, joten tässä tapauksessa kyse on todennäköisesti ollut vain yksittäisen työkalun teknisestä heikkoudesta.

6 Yhteenveto

Useat tieteelliset tutkimukset sekä käytännön projektit aina 60-luvulta lähtien ovat osoittaneet, että perinteiset lineaariset prosessimallit, kuten vesiputousmalli eivät useinkaan vastaa todellisiin ohjelmistokehityksen haasteisiin. Niiden do-

kumenttilähtöisyys ja "kerralla oikein" -mentaliteetti rajoittaa ohjelmiston muuntautumiskykyä, joka on välttämätön jatkuvasti muuttuvassa reaali maailmassa. Tilalle on ehdotettu iteratiivisiin malleihin perustuvia ns. ketteriä malleja, joissa dokumentoinnin ja etukäteissuunnittelun määrää vähennetään, asiakkaalta saadun palautteen merkitystä painotetaan ja ohjelmistoa kehitetään lyhyissä iteraatioissa. Yhtenä näihin malleihin liittyvänä periaatteena on testauslähtöisyys, jossa testit kirjoitetaan jo ennen kuin testattavaa koodia on olemassa. Näissä malleissa testeillä ohjataan myös suunnittelua ja varmistetaan ohjelmiston korkeamman tason vaatimusten oikeellisuutta.

Testeistä puhuttaessa tulisi kuitenkin kiinnittää huomiota myös siihen, kuinka hyvin testit löytävät ohjelmakoodin virheitä suhteessa testien suunnitteluun, toteutukseen ja suoritukseen vaadittuihin resursseihin. Koska testauslähtöisessä ohjelmistokehityksessä testeillä on suuri vastuu ohjelmiston laadun valvonnassa, myös testien tehokkuuteen tulisi kiinnittää yhä enemmän huomiota. Testausta on kuitenkin perinteisesti pidetty toisarvoisena, paljon aikaa vievänä työvaiheena, josta ollaan herkästi valmiita luopumaan aikataulupaineiden tai resurssipulan vuoksi. Yksi tärkeimmistä seikoista testauksen tehokkuuden ja testien laadun kehittämisessä on organisaatiokulttuurin kehittäminen kohti "testausystävällisempiä" asenteita.

Testauksen tehokkuutta voidaan myös parantaa automatisoimalla joitakin testauksen työvaiheita kuten rasiustestausta, testitapausten generointia tai kattavuusmittausta. Kustannustehokas automatisointi on kuitenkin haastava tehtävä, sillä siihen liittyy lukuisia kustannustehokkuuteen vaikuttavia parametreja. Yhtenä käytännön esimerkeistä on esitelty Helsingin Yliopiston portaalihanke. Projektin regressiotestauksen automatisointipyrkimyksissä huomattiin, että vaikka testitapausten joukon jatkuva kasvu ja sen osajoukon säännöllinen toistuvuus testauksessa nostaisi automatisoinnin kannattavuutta, tekniset ongelmat käyttöliittymärajaajapinnassa estivät testitapausten automatisoinnin. Siksi regressiotestaus päätettiin pitää manuaalisesti suoritettavana.

Testaustehokkuus liittyy testauslähtöiseen ohjelmistokehitykseen monella tapaa. Vaikka ennen tuotantokoodin kirjoitusta kirjoitettujen testien (tai *esimerkkitapausten*) tehokkuutta ei yleensä valvota, laadukkaan tuotantokoodin saavuttamiseksi testitapauksia joutuu täydentämään tuotantokoodin ollessa valmista, jolloin testaustehokkuuden valvonta, kuten kattavuusmittaus, tulevat tarpeeseen. Lisäksi testauksen automatisointi kuuluu kiinteästi testauslähtöiseen ohjelmistokehityk-

seen muun muuassa yksikkötestaus- ja regressiotestausvaiheissa.

Tämän tutkielman tarkoituksena on ollut esitellä testauslähtöisyyden historiallisia taustoja sekä kertoa niistä pääperiaatteista, jotka kuuluvat testauslähtöisyyteen. Tämän lisäksi pääaiheina käsiteltiin testauksen tehostamista ja automatisointia sekä nykyisiä testaukseen liittyviä organisaatiokulttuurin haasteita. Testauksen haasteista kerrottiin myös Helsingin Yliopiston portaalihankkeen näkökulmasta.

Kyseinen työ toimii testauslähtöisen ohjelmistokehityksen sekä testauksen tehostamisen ja automatisoinnin yleisesittelyinä, jossa on pyritty ottamaan huomioon myös taloudellinen näkökulma. Käsiteltävien asioiden haasteet on esitetty varsin abstraktilla tasolla, ja käytännönläheisempi analyysi testauslähtöisten menetelmien sekä testauksen automatisoinnin kustannustehokkuudesta sekä esitettyjen haasteiden ratkaisusta voisi olla jatkotutkimuksen aiheena.

Lähteet

- Agi06 Agile alliance, 2006. <http://www.agilealliance.org>, 3.3.2006
- Amb03 Ambler, S., Introduction to test driven development, 2003. <http://www.agiledata.org/essays/tdd.html>, 6.2.2006
- Amb06 Ambler, S., Generalizing specialists: Improving your it career skills, 2006. <http://www.agilemodeling.com/essays/generalizingSpecialists.htm>, 26.4.2006
- Bac98 Bach, J., A framework for good enough testing. *Computer*, 31,10(1998), sivut 124 – 126.
- Bec99 Beck, K., Embracing change with extreme programming. *IEEE Computer*, 32,10(1999), sivut 70–77.
- Bee00 Beedle, M. e. a., *SCRUM: An Extension Pattern Language for Hyperproductive Software Development*. Addison-Wesley, 2000.
- BMu83 Bird, D. L. ja Munoz, C. U., Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22,3(1983), sivut 229–245.

- Boe02 Boehm, B., Get ready for agile methods, with care. *IEEE Computer*, 35,1(2002), sivut 64–69.
- BCS01 British Computer Society, *Standard for Software Component Testing. Working Draft 3.4*, april 2001. [Myös <http://www.testingstandards.co.uk/Component%20Testing.pdf>, 5.4.2006].
- Bro87 Brooks, F., No silver bullet: essence and accidents of software engineering. *Computer*, 20,4(1987), sivut 10–19.
- BTP01 Bai, X., Tsai, W., Paul, R., Shen, T. ja Li, B., Distributed end-to-end testing management. *Proc. of Fifth IEEE Enterprise Distributed Object Computing Conference*, 2001, sivut 140 – 151.
- CBo05 Coram, M. ja Bohner, S., The impact of agile methods on software project management. *IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, april 2005, sivut 363 – 370.
- CLW95 Chen, H., Lyu, M. R. ja Wong, W. E., An empirical study of the correlation between code coverage and reliability estimation. Tekninen raportti.
- Cob06 Cobertura, 2006. <http://cobertura.sourceforge.net/>, 26.4.2006
- Coc02 Cockburn, A., *Agile Software Development*. Addison-Wesley, 2002.
- CPT00 Chen, T., Poon, P., Tang, S. ja Yu, Y., White on black: a white-box-oriented approach for selecting black box-generated test cases. *Proceedings of First Asia-Pacific Conference on Quality Software*, october 2000, sivut 275 – 284.
- DBD04 Du Bois, B., Demeyer, S. ja Verelst, J., Refactoring - improving coupling and cohesion of existing code. *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004, sivut 144 – 151.
- DFG95 Del Frate, F., Garg, P., Mathur, A. ja Pasquini, A., On the correlation between code coverage and software reliability. *Proc. of Sixth International Symposium on Software Reliability Engineering*, 1995, sivut 124 – 132.

- DHK93 Dalal, S., Horgan, J. ja Kettenring, J., Reliable software and communication: software quality, reliability, and safety. *Proceedings of 15th International Conference on Software Engineering*, toukokuu 1993, sivut 425 – 435.
- DTB03 Diaz, E., Tuya, J. ja Blanco, R., A modular tool for automated coverage in software testing. *Eleventh Annual International Workshop on Software Technology and Engineering Practice*, september 2003, sivut 241 – 246.
- Dyn06 Dynamic Systems Development Method Ltd., *The History of the DSDM Consortium*, 2006. URL <http://www.dsdm.org/en/about/history.asp>, 3.3.2006.
- EMT05 Erdogmus, H., Morisio, M. ja Torchiano, M., On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31,3(2005), sivut 226–237.
- Fea02 Feathers, M., Emergent optimization in test driven design. *Sardinia XP 2002 conference*, 2002.
- GBC01 Ghosh, S., Bawa, N., Craig, G. ja Kalgaonkar, K., A test management and software visualization framework for heterogeneous distributed applications. *Proc. of Sixth IEEE International Symposium on High Assurance Systems Engineering*, 2001, sivut 106 – 116.
- Gco01 gcov: a test coverage program, 2001. http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html, 26.4.2006
- Gra02 Graham, D., Requirements and testing: seven missing-link myths. *IEEE Software*, 19,5(2002), sivut 15–17.
- HCo01 Highsmith, J. ja Cockburn, A., Agile software development: the business of innovation. *IEEE Computer*, 34,9(2001), sivut 120–127.
- Hen04 Hendrickson, E., Agility for testers. *Pacific Northwest Software Quality Conference*, october 2004. [Myös <http://www.qualitytree.com/feature/aftpnsqc2004.pdf>, 6.2.2006].
- HMe02 Hieatt, E. ja Mee, R., Going faster: testing the web application. *IEEE Software*, 19,2(2002), sivut 60 – 65.

- Hua75 Huang, J. C., An approach to program testing. *ACM Comput. Surv.*, 7,3(1975), sivut 113–128.
- HVZ04 Huo, M., Verner, J., L., Z. ja Babar, M., Software quality and agile methods. *Proceedings of the 28th Annual International*, osa 1, Computer Software and Applications Conference, 2004, sivut 520 – 525.
- IEE86 IEEE Standards Board, *IEEE standard for software unit testing*, december 1986.
- IEE88 IEEE Standards Board, *IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software*, june 1989.
- Jar99 Jarzombek, S., Joint aerospace weapons systems support. *Sensors and Simulation Symposium*. Gov't Printing Office Press, 1999. Viitattu [LaB03].
- Jef99 Jeffries, R. E., Extreme testing. *Software Testing and Quality Engineering*, 1, sivut 22–27.
- Jef06 Jeffries, R. E., Test quality management in tdd, April 2006. <http://groups.yahoo.com/group/agile-testing/message/8388>, 5.4.2006
- Jir06 Jira - bug tracking, issue tracking and project management software, 2006. <http://www.atlassian.com/software/jira>, 28.4.2006
- Jme06 Jmeter, 2006. <http://jakarta.apache.org/jmeter/>, 28.4.2006
- JSa05 Janzen, D. ja Saiedian, H., Test-driven development: Concepts, taxonomy, and future direction. *IEEE Computer*, 38,9(2005), sivut 43 – 50.
- Jun06 Junit, 2006. <http://www.junit.org/index.htm>, 26.4.2006
- Jwe06 jwebunit, 2006. <http://jwebunit.sourceforge.net>, 28.4.2006
- Kan96 Kaner, C., Software negligence and testing coverage. *Proc. of Fifth International Conference on Software Testing, Analysis, and Review*, 1996, sivut 64 – 73. [Myös <http://www.kaner.com/coverage.htm>, 10.4.2006].

- KoA96 Korel, B. ja Al-Yami, A. M., Assertion-oriented automated test data generation. *ICSE '96: Proceedings of the 18th international conference on Software engineering*, Washington, DC, USA, 1996, IEEE Computer Society, sivut 71–80.
- Kek05 Kekkonen, I., Alman järjestelmättestaus, 2005. <http://nikitazhuk.net/opiskelu/tiki/alma-jarjestelmatestaus.pdf>, 27.4.2006
- Ken00 Kennedy Space Center, *Project Mercury Goals*, 2000. URL <http://www-pao.ksc.nasa.gov/history/mercury/mercury-goals.htm>, 3.3.2006.
- KHG02 Kuball, S., Hughes, G. ja Gilchrist, I., Scenario-based unit testing for reliability. *Proc. of Annual Reliability and Maintainability Symposium*, 2002, sivut 222 – 227.
- KIA02 Kataoka, Y., Imai, T., Andou, H. ja Fukaya, T., A quantitative evaluation of maintainability enhancement by refactoring. *International Conference on Software Maintenance*, october 2002, sivut 576 – 585.
- LaB03 Larman, C. ja Basili, V., Iterative and incremental developments: A brief history. *IEEE Computer*, 36,6(2003), sivut 47–56.
- LWh89 Leung, H. ja White, L., Insights into regression testing. *Conference on Software Maintenance*, october 1989, sivut 60–69.
- Mac01 MacCormack, A., Product-development practices that work. *MIT Sloan Management Rev*, 42,2(2001), sivut 75–84.
- Mar98 Marick, B., When should a test be automated. *11th Int. Software Quality Week*, 1998. [Myös <http://www.testing.com/writings/automate.pdf>, 26.4.2006].
- Mar99 Marick, B., How to misuse code coverage, 1999. URL citeseer.ist.psu.edu/marick99how.html.
- Mar03 Marick, B., Agile testing directions: tests and examples, August 2003. <http://www.testing.com/cgi-bin/blog/2003/08/22>, 5.4.2006

- Mil99 Mills, H. D., The management of software engineering, part i: Principles of software engineering. *IBM Systems Journal*, 38,2/3(1999), sivut 289–295.
- Mis03 Misra, S., Evaluating four white-box test coverage methodologies. *Proc. of Canadian Conference on Electrical and Computer Engineering*, may 2003, sivut 1739 – 1742.
- McJ82 McCracken, D. ja Jackson, M., Life cycle concept considered harmful. *ACM SIGSOFT Software Engineering Notes*, 7,2(1982), sivut 29–32.
- MNB94 Malaiya, Y., Li, N., Bieman, J., Karcich, R. ja Skibbe, B., The relationship between test coverage and reliability. *Proceedings of 5th International Symposium on Software Reliability Engineering*, 1994, sivut 186 – 195.
- MTM03 Mens, T., Tourwe, T. ja Munoz, F., Beyond the refactoring browser: advanced tool support for software refactoring. *Sixth International Workshop on Principles of Software Evolution*, september 2003, sivut 39 – 44.
- Mun88 Munoz, C. U., An approach to software product testing. *IEEE Trans. Softw. Eng.*, 14,11(1988), sivut 1589–1596.
- Mur94 Murugesan, S., Attitude towards testing: A key contributor to software quality. *Proc. of First International Conference on Software Testing, Reliability and Quality Assurance*, 1994, sivut 111 – 115.
- OMG04 Object Management Group, *UML 2.0 Superstructure Specification*, july 2004. [Myös <http://www.omg.org/technology/documents/formal/uml.htm>, 6.3.2006].
- Ocu06 Ocunit, 2006. <http://www.sente.ch/software/ocunit/>, 26. 4. 2006
- One83 O'Neill, D., Integration engineering perspective. *Journal of Systems and Software*, 3,1(1983), sivut 77–83.
- PaC86 Parnas, D. ja Clements, P., A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 1, sivut 251–257.

- Por06 Portaalityöryhmä, H. Y., Portaalihanke, 2006. <http://erinyt.helsinki.fi/portaalihanke>, 27.4.2006
- Qui03 Quilt, 2003. <http://quilt.sourceforge.net/>, 26.4.2006
- Ram04 Ramler, R., Decision support for test management in iterative and evolutionary development. *Proc. of 19th International Conference on Automated Software Engineering*, 2004, sivut 406 – 409.
- Ras03 Rasmussen, J., Introducing xp into greenfield projects: lessons learned. *IEEE Software*, 20,3(2003), sivut 21 – 28.
- Rat01 Rational, *Rational Unified Process: Best Practices for Software Development Teams*, 2001. URL <http://www-128.ibm.com/developerworks/rational/library/253.html>, 3.3.2006.
- Rei97 Reid, S., An empirical analysis of equivalence partitioning, boundary value analysis and random testing. *Proc. of Fourth International Software Metrics Symposium*, 1997, sivut 64 – 73.
- RJa00 Rising, L. ja Janoff, N., The scrum software development process for small teams. *IEEE Software*, 17,4(2000), sivut 26 – 32.
- Roy70 Royce, W., Managing the development of large software systems. *IEEE CS Press*, sivut 328–339. Viitattu [LaB03].
- Sel06 Selenium, 2006. <http://www.openqa.org/selenium>, 28.4.2006
- Sho04 Shore, J., Continuous design. *IEEE Software*, 21,1(2004), sivut 20 – 22.
- Sot01 Sotirovski, D., Heuristics for iterative software development. *IEEE Software*, 18,3(2001), sivut 66 – 73.
- TRa99 Toeppe, S. ja Ranville, S., Model driven automatic unit testing technology tool architecture. *Proc. of 18th Digital Avionics Systems Conference*, 2001, sivut 10.A.4–1 – 10.A.4–11.
- Whi00 Whittaker, J., What is software testing? and why is it so hard? *IEEE Software*, 17,1(2000), sivut 70 – 79.