LAPPEENRANNAN TEKNILLINEN YLIOPISTO

Teknillistaloudellinen Tiedekunta

Ohjelmistotekniikan Laboratorio

Jukka-Pekka Strandén

Testausmenetelmien ja testauksen lähestymistapojen valintaperusteet kirjallisuudessa

Kandidaatintyön aihe on hyväksytty 10.9.2009

Työn tarkastajana toimii DI Jussi Kasurinen

TIIVISTELMÄ

Lappeenrannan Teknillinen Yliopisto

Teknillistaloudellinen Tiedekunta

Jukka-Pekka Strandén

Testausmenetelmien ja testauksen lähestymistapojen valintaperusteet kirjallisuudessa

Kandidaatintyö 2009

26 sivua, 1 taulukko, 2 kuvaa, 0 liitettä

Tarkastaja: DI Jussi Kasurinen

Hakusanat: Ohjelmistotestaus, testaustapaus, testaustapausten valitseminen

Keywords: Software testing, test case, test case selection

Testaustapausten valitseminen on testauksessa tärkeää, koska kaikkia testaustapauksia ei voida testata aika- ja raharajoitteiden takia. Testaustapausten valintaan on paljon eri menetelmiä joista eniten esillä olevat ovat malleihin perustuva valinta, kombinaatiovalinta ja riskeihin perustuva valinta. Kaikkiin edellä mainittuihin menetelmiin testaustapaukset luodaan ohjelman spesifikaation perusteella. Malleihin perustuvassa menetelmässä käytetään hyväksi ohjelman toiminnasta olevia malleja, joista valitaan tärkeimmät testattavaksi. Kombinaatiotestauksessa testitapaukset on muodostettu ominaisuuspareina jolloin yhden parin testaamisesta päätellään kahden ominaisuuden toiminta. Kombinaatiotestaus on tehokas löytämään virheitä, jotka johtuvat yhdestä tai kahdesta tekijästä. Riskeihin perustuva testaus pyrkii arvioimaan ohjelman riskejä ja valitsemaan testitapaukset niiden perusteella. Kaikissa menetelmissä priorisointi on tärkeässä roolissa, jotta testauksesta saadaan riittävä luotettavuus ilman kustannusten nousua.

ABSTRACT

Lappeenranta University of Technology

Faculty of Technology Management

Jukka-Pekka Strandén

Test strategy and testing approach selection in literature

Bachelor Thesis 2009

26 pages, 1 table, 2 figures, 0 appendixes

Examiner: M. Sc. Jussi Kasurinen

Keywords: Software testing, test case, test case selection

Test case selection is important in software testing because software projects have finite time and monetary resources. There are many methods for selecting test cases. The most prominent ones are model-based testing, combinatorial testing and risk-based testing. In model-based selection method, the test cases are selected based on a model of the program. The most important of the test cases are selected for testing. In combinatorial testing, the test cases are formed from program attribute pairs. These pairs are then tested and from the test results, the real functionality of the attributes can be evaluated. Combinatorial test case selection method is good at finding faults which are caused by one or two variables while it has a weakness of finding complex faults. In risk-based testing, the program risks are analyzed and the test cases are selected based on the risks. All of the techniques mentioned above also employ prioritization as a tool to re-order the test cases. Prioritization has proven to be an effective tool in all test case selection methods when working with limited time and money.

ESIPUHE

Syksy 2003 ja yliopisto-opintojeni aloitus tuntuvat nyt niin etäiseltä. Opiskelen nyt 7:ttä vuotta ja opinnot alkavat pikku hiljaa lähestyä loppua monista vastoinkäymisistä huolimatta. Tämä kandidaatintyö on ollut minulle samalla vaativin, mutta myös mielenkiintoisin projektini opiskeluaikana. Kuka olisi kuvitellut, että "yksinkertaisen" kirjallisuusarvioinnin kirjoittaminen voisi olla näin haastavaa. Silti tämän etapin saavuttamisesta huolimatta on muistettava, että varsinainen haaste, diplomityö, häämöttää vielä (lähi)tulevaisuudessa. Odotan haastetta innolla.

Haluan kiittää ensinnäkin tutkijakoulutettava Jussi Kasurista jonka kärsivällisen ohjauksen ansiosta sain tämän työn valmiiksi aikataulussa. Myös professori Kari Smolander ja Ossi Taipale ansaitsevat kiitokset siitä, että ohjasivat minut MASTO -projektiin.

Lappeenrannassa 24.11.2009

Jukka-Pekka Strandén

SISÄLLYSLUETTELO

1 JOHDANTO....2
1.1 Tausta....2
1.2 Tavoitteet ja Rajoitteet....2
1.3 Työn Rakenne....2

2 YLEISTÄ TESTAUKSESTA....3
2.1 Testaustapausten luonti ja valinta....4
2.2 Testauksen kattavuus....4
2.3 ISO 29119....5

3 KIRJALLISUUSKATSAUS....7
3.1 Testaustapausten valinta....7
3.1.1 Regressiotestaus....8
3.1.2 Kombinaatiotestaus....10
3.1.3 Malleihin perustuva testaus....12
3.1.4 Riskeihin perustuva testaus....13

4 POHDINTA....14
4.1 Testaustapausten luonti....14
4.2 Testaustapausten valintamenetelmät....16
4.3 Priorisointi....17
4.4 Johtopäätökset....18

5 YHTEENVETO....19

LÄHTEET....20

1 JOHDANTO

1.1 Tausta

Työ on tehty kirjallisuuskatsauksena osana MASTO -projektia. Projekti tutkii testausta eri yrityksissä. Masto -projekti on osa ESPA -projektia jonka tavoitteena on luoda referenssimalli joka perustuu ISO 29119 -standardiin [1].

1.2 Tavoitteet ja Rajoitteet

Työn tavoitteena on saada selville mitä kirjallisuus sanoo testaustapausten valinnasta ja testauksen lähestymistavoista. Työ suoritetaan kirjallisuusarviointina ja lähteinä käytetään verkkotietokantoja kuten ACM, IEEE ja Springer. Työ on rajattu testaustapausten valintaan eikä käsittele itse testausmenetelmiä elleivät ne suoraan liity testaustapausten valintaan.

1.3 Työn Rakenne

Työ on jaoteltu viiteen eri osioon. Ensimmäinen osio on johdanto. Toinen ja kolmas osio ovat kirjallisuuskatsaus. Toisessa osiossa esitellään testausta yleisesti oppikirjan näkökulmasta luoden pohjaa tieteellisille artikkeleille. Lisäksi toisessa osiossa esitellään ISO29119 -standardi tarkemmin. Kolmannessa osiossa on kirjallisuuskatsauksen toinen puoli, jossa käydään läpi artikkeleja. Osio on jaoteltu eri testaustyylien mukaan, niin että jokainen testaustapausten valintamenetelmä on oma aliotsikkonsa. Osiossa neljä on pohdintaa ja artikkeleiden sisällön yhdistämistä osiossa kaksi mainittuihin testeihin ja ISO29119 standardiin. Osio neljä sisältää yhteenvedon työstä.

2 YLEISTÄ TESTAUKSESTA

Testauksella tarkoitetaan systemaattista järjestelmän ominaisuuksien läpikäymistä, jonka tavoitteena on saada selville järjestelmässä mahdollisesti olevat virheet sekä tarkistaa kuinka hyvin järjestelmä toteuttaa sille annetut vaatimukset. Testaus on tärkeä osa ohjelmistotuotantoa ja arviolta 50% ohjelmistoprojektin kustannuksista tulee testauksesta. Koska testaus on rahallisesti iso osa ohjelmistoprojektia, voidaan testausmenetelmiä tehostamalla joko säästää kustannuksissa ja/tai parantaa tuotteen laatua. Testaustapoja ja kohteita on useita ja usein niitä yhdistellään. Usein ohjelmiston testausvaiheet voidaan jaotella seuraavasti: [2].

Moduulitestauksessa testattavana on yksittäinen moduuli joka koostuu yleensä 100-1000 ohjelmarivistä [2]. Moduulin toimintaa verrataan sen ohjelmiston spesifikaatiossa tehtyyn määritelmään. Testauksen suorittaa yleensä moduulin tuottaja ja se tehdään yleensä siinä vaiheessa, kun moduuli on valmis. Eli, moduulitestaus suoritetaan jo yleensä ohjelmiston koodausvaiheessa [2].

Integraatiotestauksessa valmiit moduulit yhdistetään moduuliryhmiä toimiviksi kokonaisuuksiksi ja testataan moduuliryhmien toimintaa. Testauksen pääpaino on moduulien välisten rajapintojen testauksessa [2].

Järjestelmätestauksessa testattavana on koko järjestelmä ja tuloksia verrataan määrittelydokumentteihin. Järjestelmätestaajien tulisi olla varsinaisesta ohjelmiston toteutuksesta mahdollisimman riippumaton joukko, koska ohjelman luoneet henkilöt eivät välttämättä pysty tarkastelemaan tuotostaan täysin objektiivisesti. Testaajien ollessa riippumattomia, saadaan järjestelmätestistä parempia tuloksia. Järjestelmätestauksessa testataan järjestelmän toiminnalliset sekä ei-toiminnalliset (luotettavuus, turvallisuus, kuormitustestit jne.) ominaisuudet. Testauksesta löydettyjen virheiden korjaus voi luoda uusia virheitä ja usein järjestelmälle tehdään regressiotestaus, jossa järjestelmätestaus suoritetaan ohjelmalle uudelleen [2].

Käytettävyystestauksella pyritään varmistamaan, että käyttäjä pystyy selviämään tehtävistä joita varten ohjelmaa ollaan tekemässä. Käytettävyystesteihin valitaan usein pieni joukko ohjelman loppukäyttäjistä ja heidän selviytymistään eri koetilanteissa valvotaan. Myös käyttöliittymien ja käytettävyyden arviointiin erikoistuvia ammattilaisia voidaan palkata arvioimaan ohjelman käytettävyyttä [2].

2.1 Testaustapausten luonti ja valinta

Testaustapausten luontiin on kaksi peruslähestymistapaa: lasilaatikkotestaus (white box testing) ja mustalaatikkotestaus (black box testing). Lasilaatikkotestaus keskittyy koodin testaamiseen ja on erittäin yleinen lähestymistapa moduulitestauksessa sekä moduulien regressiotestauksessa. Mustalaatikkotestauksessa testaustapaukset luodaan spesifikaation perusteella ja testeissä keskitytään tutkimaan toteuttaako ohjelma spesifikaation määrittelemät toiminnallisuuden. Harmaalaatikkotestauksessa (gray box testing) käytetään hyväksi tietoa ohjelman toteusperiaatteista ja yhdistellään musta- ja lasilaatikkotestaustapojen luontimenetelmiä [2].

Kun testaustapaukset on luotu, on kyseessä usein niin suuri joukko, ettei kaikkia niistä pystytä testaamaan. Tällöin testaustapauksista pitää valita ne jotka suoritetaan niin, että testattavan ohjelman ominaisuudet tulevat testattua mahdollisimman tarkasti [2].

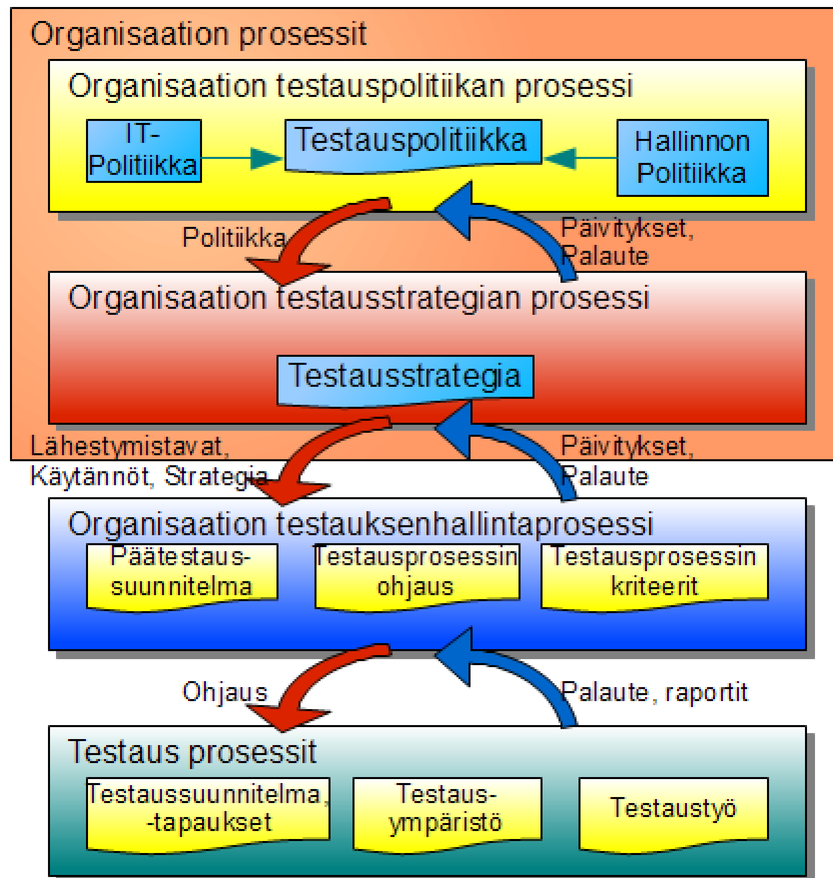
2.2 Testauksen kattavuus

Testauksen kattavuutta on vaikea mitata, koska kaikkia ison järjestelmän virheitä on lähes mahdoton havaita. Monet virheet eivät ole selvästi näkyvillä. Projektista löydetyt virheet ja niiden korjaukset eivät aina kohtaa. Koska projekteilla on aikarajaukset, kaikkia virheitä ei voida edes korjata. Koska resurssit ovat rajalliset, pitää testaustapaukset ja testattavat moduulit priorisoida jotenkin. Kompleksisuusmitoilla yritetään paikantaa monimutkaiset moduulit, jotka vaativat eniten testausta. Kattavuusmitoilla yritetään varmistaa, että testiaineisto aiheuttaa ohjelman kaikkien osien suorittamisen. Yksi kattavuusmitta on

Lausekattavuus, joka kertoo kuinka suuri osa ohjelman koodiriveistä suoritetaan ainakin kerran ohjelman suorituksen aikana. 100%:n lausekattavuuteen on mahdoton päästä. Toinen kattavuusmitta on päätöskattavuus joka tarkistaa, että ehtorakenne saa vähintään kerran molemmat arvonsa. Ehtokattavuus tarkistaa, että päätöksen kaikkien ehtojen on saatava kaikki arvonsa. Nämä mittarit yhdistämällä voidaan arvioida testauksen kattavuutta [2].

2.3 ISO 29119

ISO 29119 standardi määrittelee organisaation testauksen kolmelle eri tasolle joista ylin määrittelee vain organisaation testauspolitiikan ja -strategian. Alemmat tasot tekevät testaussuunnitelmat, luovat testaustapaukset ja suorittavat testit käyttäen ohjeinaan organisaation testausstrategiaa. Jokainen taso raportoi ylemmälle tasolle työnsä tulokset ja ylemmät tasot ohjaavat palautteen mukaan alempia tasoja. Kuvassa 1 on esitetty ISO29119 -standardin eri tasot [1].



Kuva 1: ISO29119 mallin organisaation tasot

ISO29119 -mallissa ylin taso on organisaation taso, joka voidaan edelleen erotella kahdeksi tasoksi "Organisaation testauspolitiikka" ja "Organisaation testausstrategia". Organisaation testauspolitiikka sisältää organisaation ohjeet ja näkemykset testaukseen. Se on hyvin lyhyt, muutaman sivun mittainen kuvaus miksi ja mitä testausta suoritetaan. Organisaation testausstrategia on tarkempi kuvaus testauksesta, mutta myös se koskee koko yrityksen testausta [1].

Organisaatiotason alapuolella ovat testaustasot joilla suoritetaan varsinainen ohjelmiston testaus. ”Testienhallinta” tasolla luodaan ohjelmistoille testisuunnitelma, aikataulu ja kriteerit. ”Testausprosessit” taso suorittaa varsinaisen testauksen ja luo päätestaussuunnitelman mukaan testaustapaukset ja yksityiskohtaisemman testaussuunnitelman [1].

3 KIRJALLISUUSKATSAUS

Testauksesta on kirjoitettu paljon artikkeleita ja tutkimuksia. Ehkä suurin määrä artikkeleita löytyy regressiotestauksesta, joka on erittäin tärkeä osa ohjelmistotuotantoa. Regressiotestaukseen liittyvät artikkelit käsittelevät usein testauksen automatisointia. Monen artikkelin yhteydessä on myös suoritettu empiirisiä tutkimuksia, mutta ne ovat pienen skaalan usein keskittyen tiettyyn ohjelmaan, sen eri versioihin tai vain muutamaan ohjelmaan.

Yliopistolla on myös omaa tutkimusta testauksesta, mutta se keskittyy enemmän ohjelmistoarkkitehtuureihin kuin testaustapausten valintaan, mikä on tämän työn kohde [3]. Testaustapausten valinnassa ja erityisesti regressiotestauksessa Gregg Rothermelin nimi tulee useasti esille [6, 7, 10].

3.1 Testaustapausten valinta

Kuten johdannossa on määritelty, testaustapaukset luodaan ensin lasilaatikko-, mustalaatikko- tai harmaalaatikkomenetelmällä. Tästä luodusta joukosta valitaan sitten suoritettavat testaustapaukset. Testaustapaukset valintaan liittyy myös selkeästi suoritettavan testauksen tyyppi. Regressiotestauksessa suoritetaan jo suoritetut järjestelmä-, integraatio- tai moduulitestaustapaukset uudestaan testattavan kohteen muokkauksen jälkeen. Kombinaatiotestauksessa, testaustapoja rajoitetaan niin, että saadaan testattua ohjelman kaikki ominaisuudet, mutta kaikkia yhdistelmiä ei testata. Esimerkiksi, järjestelmästä voidaan testata ominaisuus A käyttöjärjestelmässä X ja ominaisuus B käyttöjärjestelmässä Y ja mikäli testeissä ei ilmene virheitä, voidaan olettaa, että ominaisuudet A ja B toimivat järjestelmissä X ja Y.

3.1.1 Regressiotestaus

Regressiotestauksesta (Regression testing) on kirjoitettu paljon artikkeleja ja tutkimuksia. Yleisesti ne keskittyvät järjestelmätestaukseen ja testaustapausten automaattiseen valintaan sekä testaustapausten karsintaan joukosta. Uudemmissa artikkeleissa myös testattavan ohjelmiston elinikä on huomioitu paremmin. Regressiotestauksessa käytettävät testaustapaukset voidaan luoda joko mustalaatikko-, lasilaatikko- tai harmaalaatikkomenetelmällä.

Chenin, Probertin ja Simsin artikkelissa "Specification-based Regression Test Selection with Risk Analysis" [4] korostetaan, että lasilaatikkomenetelmällä luoduilla testaustapauksilla ei saada suoritettua järjestelmätestausta tehokkaasti, koska koodia muutettaessa joudutaan muuttamaan myös testaustapaukset ja ehkä jopa spesifikaatiota. Mutta jos testaustapaukset on luotu spesifikaation perusteella, ei testaustapauksia tai spesifikaatiota tarvitse muuttaa joka kerta, kun koodia muutetaan. Näin ollen voidaan regressiotestauksesta myös sulkea ulos testaustapauksia, jotka liittyvät koodiin jota ei ole muutettu [4].

Spesifikaatioon perustuvaan regressiotestauksen etuihin ovat myös päätyneet muut henkilöt. Muccini, Dias ja Richardson artikkelissaan "Towards Software Architecture-based Regression testing" toteavat samat koodiin perustuvan testaustapausten valinnan ongelmat kuin Chen, Probert ja Sims [4, 5]. He myös mainitsevat, että lasilaatikkomenetelmällä luoduilla testaustapauksilla ei voida testata ohjelman kaikkia ominaisuuksia. Isommaksi ongelmaksi he mainitsivat, että koodiin perustuva testaus ei huomaa virheitä ohjelmiston toiminnallisuudessa [5].

Ratkaisuksi ongelmaan he esittävät ohjelmistoarkkitehtuuriin perustuvaa valitsemistapaa. Testaustapaukset luodaan ohjelmistoarkkitehtuurille ja niistä muodostetaan varsinaiset suunnitelmat koodin testaukseen. Näin testausta ajetaan ohjelmiston arkkitehtuurin mukaan ja muutokset koodiin eivät välttämättä riko testaustapauksia elleivät ne muuta ohjelman toiminnallisuutta [5].

Chen, Probert ja Sims myös korostavat, että on tärkeä ottaa huomioon projektin riskit, kun suunnitellaan regressiotestausta. He ottavat huomioon testauksen tehokkuuden, kustannustehokkuuden ja riskiherkkyyden, kun arvioidaan testauksen onnistumista. IBM:n Websphere Commerce 5.4:lla suoritetussa kokeessa riskit huomioon ottava regressiotestaustapausten valinta oli tehokkaampi, kuin manuaalinen valinta [4].

Ohjelmistotuotantoprojekteissa on usein tiukat aikarajoitukset joiden takia testaustapausten määrää joudutaan rajaamaan [4, 6, 7]. Do ja Rothermel mainitsevat myös priorisoinnin tärkeyden. Koska testaustapauksia joudutaan tiputtamaan aikarajoitusten takia, kannattaa suorittaa testaus ohjelmiston kaikista kriittisimmille komponenteille ja jättää vähemmän kriittiset komponentit testaamatta [4, 6, 7].

Toinen priorisointitapa on arvioida testattavien ohjelmistokomponenttien kompleksisuutta ja näin päätellä mitkä komponentit tarvitsevat eniten testausta. Nämä komponentit testataan ensin ja ajan salliessa testataan todennäköisesti vähemmän virheitä sisältävät komponentit. Do ja Rothermel mainitsevat myös, että mikäli aikarajoituksia ei ole, ei priorisointi kannata tai ole kustannustehokasta. Kun ohjelmistoprojektilla on aikarajoituksia, priorisointi kannattaa aina, vaikka sitä ei olisi tehty hyvin [6, 7].

Julkaisupaine ohjelmilla on yleensä kova ja yritykset yrittävät usein arvioida testauksen tarpeellisuutta kustannusten ja voiton mukaan. Mikäli tuote julkaistaan ajoissa, on se myynnissä pitempään ja tuotto on suurempi. Mutta tuotteessa voi myös paljastua vakavia vikoja, jotka heikentävät tuotteen ja yrityksen mainetta ja ovat kalliita korjata. Toisaalta myös regressiotestaus ja myöhemmin julkaisu voi kostautua yritykselle. Ohjelmistoyritykselle, joka valmistaa tuotetta yksittäiselle asiakkaille pätee myös sama ongelma, koska testaukseen käytetty aika maksaa ja sopimuksessa voi olla sanktioita myöhästymisestä. Viallisen tuotteen jälkikorjaus voi aiheuttaa vielä suuremmat kustannukset. Do ja Rothermel arvioivat regressiotestauksen hinnan muodostuvan seuraavanlaisista asioista; Testien kokoonpano, turhien testaustapausten tunnistaminen, turhien testaustapausten korjaaminen, analysointi, tekniikoiden toteutus, tulosten arviointi, huomaamattomat viat ja muut kustannukset joita ei ole huomioitu. He esittävät

kustannustehokkaan testausmallin joka ottaa huomioon edellä mainitut asiat sekä ohjelman elinkaaren. Mallia testattiin viidellä Java ohjelmalla ja testauksessa käytettiin ”testaa kaikki”, testaustapausten priorisointia ja turvallista regressiotestaustapausten valitsemista (menetelmä löytää ainakin samat viat muutetusta ohjelmasta O' kuin O) menetelmiä. Edellä mainittuja menetelmiä tutkittiin aikarajoituksien yhteydessä sekä inkrementaalisten resurssien tapauksessa [6].

Testauksen tuloksena huomattiin, että priorisointi on testaustapausten valintaa parempi menetelmä joissain tapauksissa ja joissain ei. Testaustapausten määrä ja testattavan ohjelmiston tyyppi vaikuttavat menetelmien tehokkuuteen. Tutkimus oli suoritettu viidellä verrattain pienellä ohjelmalla ja vaikka ohjelman koolla ei näyttäisi olevan vaikutusta tuloksiin, tarvitaan testausta isommilla teollisuuskäytössä olevilla ohjelmistoilla joilla myös tulot ovat suuremmat [6].

3.1.2 Kombinaatiotestaus

Kombinaatiotestaus (Combinatorial testing) ei ole saanut yhtä paljon huomiota kuin regressiotestaus. On kuitenkin huomioitava, että kombinaatiotestaus on testaustapausten valintamenetelmä, kun taas regressiotestaus on testaus, jossa testataan moduuli, moduuliryhmä tai järjestelmä uudestaan käyttäen samoja testaustapauksia. Kombinaatiotestausta voi näin ollen soveltaa regressiotestauksessa. Kuitenkin, kombinaatiotestausta ei ole yleensä mainittu artikkeleissa joissa käsitellään testaustapausten valintaa regressiotestaukseen.

Poikkeuksiakin on ja artikkeli "Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization" käsittelee kombinaatiotestausta, kun regressiotestataan konfiguroitavaa ohjelmaa [10]. Ohjelman testaustapaukset voivat muuttua versiosta toiseen ja konfiguroitava ohjelma tuo lisää haasteita, kun käyttäjät voivat muuttaa sen toimintaa jo ennen käynnistystä tai ohjelman pyöriessä. Konfiguraatiot vaikuttavat suuresti myös testauksessa löydettyihin virheiden määrään ja minkälaisia virheitä löydetään [10].

Suurin osa järjestelmän virheistä johtuu yhdestä tai kahdesta muuttujasta. Tästä seikasta johtuen, kombinaatiotestaus on hyvä löytämään virheitä [8, 9]. Virheet, jotka riippuvat monesta seikasta ovat harvinaisia. Esimerkiksi virhe joka ilmenee vain, jos päivämäärä, fontti, käyttöjärjestelmä ja käyttäjän toimet ovat oikeat, on harvinainen ja kombinaatiotestaus ei todennäköisesti sitä löydä. Jo parittaisessa kombinaatiotestauksessa, jossa testataan testitapaukset pareittain, kombinaatiotestaus löytää jopa 50-97% järjestelmän virheistä. Mutta kuten edellä on mainittu, kombinaatiotestaus ei löydä virheitä, jotka johtuvat useammasta muuttajasta [8, 9].

Yleisimmin käytetty kombinaatiotestausmenetelmä on paritestaus (pairwise testing), jossa tavoitteena on testata jokainen pari kerran. Pari koostuu kahdesta eri muuttujasta, esimerkiksi käyttöjärjestelmästä ja prosessorista. Mallia voidaan soveltaa myös useamman muuttujan tapaukseen. Artikkelissa "Moduling Requirements for Combinatorial Software Testing" esiteltiin kombinaatiotestausmenetelmä, joka luo testaustapaukset automaattisesti [11]. Menetelmä käyttää ohjelman vuo- ja tilakaavioita joista luodaan testattavat parit. Menetelmää testattiin eri malleilla ja se kombinaatiotestaus arvioitiin tehokkaaksi lähestymistavaksi testaukseen [11].

Qu, Cohen ja Rothermel esittivät kombinaatiotestauksen käyttöä regressiotestauksessa [10]. Heidän menetelmässä, kuten muissa kombinaatiotestausmenetelmissä [10, 11, 12], luodaan testausparit ja niistä edelleen matemaattiset kattavuustaulukot (covering array), johon on sijoitettu ohjelman testattavat muuttujat/konfiguraatiot. Koska testejä pitää usein priorisoida, lasketaan kattavuustaulukosta uusi puolueellinen kattavuustaulukko (biased covering array). Tämä taulukko asettaa suoritetavat testausparit uuteen järjestykseen niiden tärkeyden mukaan. Arvot voidaan vielä kertoa painoilla jotka voidaan saada edellisen version testien tuloksista tai arvioida ohjelman spesifikaatiosta. Qu, Cohen ja Rothermel toteavat, että ohjelman konfiguraatiot vaikuttavat mitä virheitä kombinaatiotestaus löytää. Kuitenkin, vain melko pieni määrä konfiguraatioita tarvitsee testata, jotta saavutetaan iso vikojen löytymisprosentti. Menetelmä vie kuitenkin paljon aikaa ja siksi priorisointi on tärkeää, mikäli testaukselle on asetettu aikarajat [10, 11].

3.1.3 Malleihin perustuva testaus

Malleihin perustuvassa testauksessa (Model-based testing) testaustapaukset luodaan mustalaatikkomenetelmällä. Testaustapana malleihin perustuva testaus on noussut suosioon viime vuosina ohjelmien kompleksisuuden kasvun ja laadun varmistamisen tärkeyden takia [13]. Testaustapoja luodessa käytetään ohjelman spesifikaatioita ja erityisesti diagrammeja ohjelman käyttäytymisestä, kuten UML:n tilakaaviot.

Olioperustaisessa ohjelmistotuotannossa, ohjelmia tuotetaan käyttämällä valmiita malleja. Esimerkiksi tarkkailija mallia (observer design pattern). Malleihin perustuvassa testauksessa voidaan käyttää hyväksi edellä mainittujen suunnittelumalleihin perustuvia testauskäytäntöjä, kun suunnitellaan ohjelman testausta [13].

Malleihin perustuvaa testausta on erityisesti suositeltu ohjelmistojen tuotantolinjoihin, koska yhdellä kehitetyllä mallilla voidaan potentiaalisesti testata kaikki linjaston ohjelmat. Olinpiewin ja Gomaan artikkelissa testaustapaukset valitaan manuaalisesti ohjelman kriteerien mukaan [14]. Malleihin perustuvassa testauksessa testaustapaukset voidaan myös valita automaattisesti. Pretschner et al. tutkivat automaattisen ja manuaalisen malleihin perustuvaa testaustapausten valinnan tehoa verrattuna "käsin" tehtyyn testaustapausten joukkoon [15]. Heidän tutkimuksessaan manuaalinen ja automaattinen malleihin perustuva testaus löysi paremmin spesifikaatiovirheitä, kuin käsin luotu testaustapausten joukko [14, 15].

Boberg artikkelissaan "Early Fault Detection with Fault Based Testing" käsittelee aikaista virheiden havaitsemista malleihin perustuvassa testauksessa [16]. Yleisesti ottaen, mitä myöhäisemmässä vaiheessa virhe havaitaan, sitä kalliimmaksi sen korjaaminen tulee. Malleihin perustuva testaus on hyvä löytämään virheet aikaisessa vaiheessa [16]. Bobergin mukaan asiakkaan luottamus ohjelmistoprojektiin pysyy myös korkeampana, mikäli testaus tuottaa konkreettisia tuloksia.

3.1.4 Riskeihin perustuva testaus

Riskeihin perustuvassa testauksessa (Risk-based testing) testaustapaukset valitaan niin, että ne tapaukset joihin liittyy isoimmat riskit testataan ensin. Koska riskejä voidaan arvioida ja määrittää eri tavoilla, on riskeihin perustuva testaus terminä melko laaja. Redmill artikkelissaan "Exploring risk-based testing and its implications" kirjoittaa "'Risk-based testing' is neither consistently defined nor supported by literature on either theory or practice." [17, s. 4]. Lause summaa ongelman, että ohjelmistokehittäjät haluavat välttää riskejä, mutta eivät silti tiedä tarkkaan mitä ne ovat. Redmill pyrkii määrittämään mitä riskillä tarkoitetaan artikkelissaan ja mainitsee, että se on myös hyvin subjektiivinen.

Ohjelmistoprojektissa hän määrittää riskin, mahdollisuutena, että jokin virhe tapahtuu. Hän myös mainitsee kaksi riskin ominaisuutta, jotka tulisi ottaa huomioon ohjelmistoa suunniteltaessa; riskin todennäköisyys ja seuraukset. Erityisesti seurausten arvioinnissa ihmisten subjektiivisuus tulee esille. Jonkin riskin toteutuminen voi tarkoittaa kehittäjille lisätyötä. Mutta asiakkaalle se voi tarkoittaa menetettyä mainetta ja huonompaa liiketoimintaa. Niinpä riskejä arvioitaessa olisi tärkeää saada kaikkien osapuolien arviot riskien seurauksista [17].

Vaikka riskeihin perustuva testaus on melko epätarkasti määritelty, parantaa se silti testauksen tehokkuutta. Jos riskeistä saadaan tarkat arvioit, voidaan testaustapaukset valita ja/tai priorisoida niin, että ohjelman laatu paranee. Erityisesti virheet vähentyisivät niissä osa-alueissa joissa seuraukset olisivat vakavimmat eri osapuolille [17].

Riskien arviointi voidaan tehdä myös lähdekoodin perusteella. Wong, Qi ja Cooper artikkelissaan "Source Code-based Software Risk assessing" esittelevät staattisen ja dynaamisen riskimallin [18]. Riskin he määrittävät ohjelmistoprojektin potentiaalisena ongelmana. Staattinen ja dynaaminen riskimalli eroavat toisistaan sillä, että staattinen malli tutkii koodia ja sen rakennetta, kun taas dynaaminen käyttää testauksen kattavuusmittoja. Artikkelin yhteydessä suoritetussa tutkimuksessa huomattiin, että suurin osa ohjelman ongelmista löydettiin niistä funktioista jotka avioitiin korkeaksi riskiksi [18].

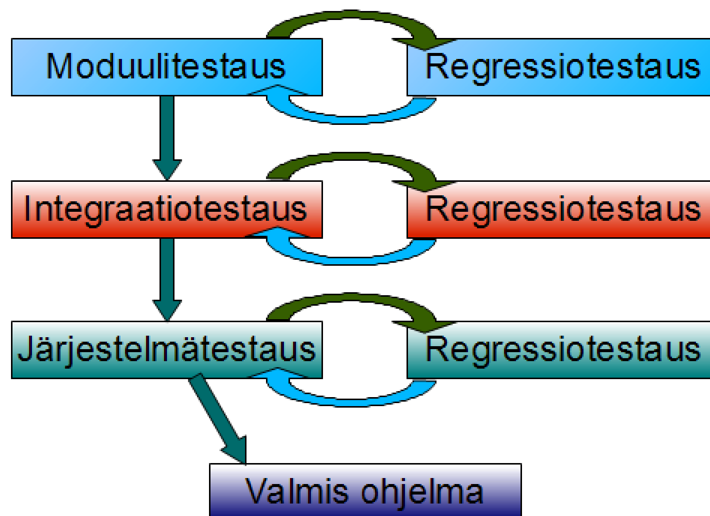
4 POHDINTA

Edellisessä osiossa esiteltiin eri testaustapausten ja testauksen lähestymistapojen valintaperusteita. Artikkeleja pyrittiin käsittelemään mahdollisimman neutraalisti. Tässä osiossa analysoidaan edellä esiteltyjä artikkeleja. Lopuksi pyritään yhdistämään tärkeimmät huomioitavat seikat.

Testaustapausten valinnasta on kirjoitettu paljon artikkeleita ja monen artikkelin yhteydessä on suoritettu empiirinen tutkimus [4, 6, 7, 8, 9, 10]. Nämä tutkimukset on usein suoritettu verrattain pienessä mittakaavassa joko yhden ohjelman usealla versiolla tai muutamalla ohjelmalla. Näin ollen, niiden esittämiä testausmenetelmien tehokkuutta ei ole testattu laajassa käytössä. Tästä huolimatta, nämä menetelmät ovat lupaavia ja niitä pitäisi testata enemmän käytännössä.

4.1 Testaustapausten luonti

Testauksen eri vaiheet esiteltiin toisessa osiossa seuraavanlaisesti: moduulitestaus, integraatiotestaus ja järjestelmätestaus. Lisäksi esiteltiin käytettävyystestaus joka voidaan katsoa erilliseksi testaukseksi. Kuvassa 2 on esitetty testauksen normaali eteneminen yksinkertaistetusti. Jokaisella tasolla voidaan aina suorittaa regressiotestaus ja jos tulokset ovat hyvät, siirtyä seuraavan testausvaiheeseen. Käyttöliittymätestaus on näistä irrallinen testaus, mutta myös sille voidaan suorittaa regressiotestaus ja näin varmistaa, että käyttöliittymään ei tule pahoja virheitä.



Kuva 2: Ohjelmiston testauksen eteneminen

Näihin testeihin luodaan testaustapaukset joko lasi-, musta- tai harmaalaatikkomenetelmällä. Ehkä perinteisesti, testaustapaukset on luotu koodin pohjalta lasilaatikkomenetelmällä, mutta kirjallisuudessa spesifikaatioon perustuvat menetelmät ovat suositellumpia [4, 5]. Taulukossa 1 on esitelty eri testausvaiheisiin soveltuvat testaustapausten luontimenetelmät.

Taulukko 1: Testausvaiheet

Testausvaihe	Testaustapausten luontimenetelmä
Moduulitestaus	Lasilaatikko
Moduuliryhmien testaus	Lasi- tai mustalaatikko
Integraatiotestaus	Mustalaatikko
Järjestelmätestaus	Mustalaatikko

Järjestelmän yksittäisille moduuleille ja ehkä moduuliryhmille, koodiin perustuvat menetelmät toimivat, koska niissä moduulien ottavat ja antavat arvot pysyvät verrattain yksinkertaisina. Kun siirrytään isompiin kokonaisuuksiin, kuten integraatiotestaukseen ja järjestelmätestaukseen, koodiin perustuvassa valinnassa on paljon puutteita. Mikäli koodia muutetaan, joudutaan myös testaustapaukset tekemään uusiksi. Erityisesti regressiotestauksessa tähän kuluu aikaa ja rahaa. Mikäli testaustapausten luonti on tehty spesifikaation pohjalta, voidaan koodia muuttaa ilman, että testaustapauksia täytyy

muuttaa, ellei muutos tule ohjelman käyttäytymiseen, joka vaatii myös spesifikaatioon muutoksen.

Koodiin perustuvassa testaustapausten luonnissa on myös mahdollista, että spesifikaatio ja koodi lähtevät "luistamaan" spesifikaatioon perustuvaa menetelmää herkemmin. Koodia muutokset voivat vaikuttaa spesifikaatioon, mutta niitä ei dokumentoida ja spesifikaatiota ei päivitetä jolloin se ja toteutus poikkeavat. Kun testattavan kohteen koko ja kompleksisuus kasvaa, on spesifikaatioon perustuva testaustapaustenluontimenetelmä parempi ratkaisu.

4.2 Testaustapausten valintamenetelmät

Testaustapausten valintaan on useita menetelmiä, mutta testaustapauksista voidaan myös valita suoritettavaksi kaikki. Kuvassa 2 esitetyssä prosessissa "valitse kaikki" metodia ei voi suositella kuin moduulitestaukseen. Pienissä ohjelmissa menetelmä voi vielä toimia integraatiotestauksessa. Nykyisessä ohjelmistotuotannossa järjestelmätestaukseen ja usein jo integraatio- ja moduulitestaukseen joudutaan kuitenkin rajoittamaan suoritettavien testaustapausten määrää aika- ja raharajoitteiden takia.

Ehkä helpoin valintamenetelmä on perinteinen manuaalinen valinta, jossa testaustapaukset valitsee testauksesta vastaava henkilö oman intuition pohjalta. Myös kirjallisuudessa on esitelty useita eri valintamenetelmiä, mutta muutamat menetelmät ovat suositumpia. Näitä on kombinaatiotestaus ja erityisesti malleihin perustuva testaus.

Malleihin perustuvassa testauksessa, testaustapaukset valitaan ohjelman spesifikaation perusteella käyttäen apuna mallia ohjelman toiminnasta. Testausmenetelmää perustellaan tehokkaaksi myös sillä, että siinä voidaan käyttää aloitusmalleina samankaltaisille ohjelmille tehtyjä testausjoukkoa. Malleihin perustuva testaustapausten valintaa voidaan myös käyttää pohjana ja yhdistää muita valintamenetelmiä siihen. Esimerkiksi kombinaatiotestaustapausten valintamenetelmää voidaan käyttää malliperustaisen

valintamenetelmän kanssa yhdessä. Testaustapaukset luodaan ensin mustalaatikkomenetelmällä, valitaan niistä ensimmäinen testausjoukko malliperustaisella valintamenetelmällä ja supistetaan jo valittua joukkoa kombinaatiomenetelmällä.

Kombinaatiotestaus on empiirisissä tutkimuksissa todettu erittäin tehokkaaksi tavaksi löytää ohjelman virheet. Sen ainoa puute on, että se ei huomaa monimutkaisia virheitä. Usein ohjelmien virheet johtuvat yhdestä tai kahdesta seikasta, joten menetelmä on silti tehokas löytämään suurimman osan virheistä.

Kombinaatiotestaus on tehokas työväline, kun ohjelman laadusta halutaan varmistua pienellä testaustapausten joukolla. Toisaalta se mahdollistaa myös kattavamman ominaisuuksien testauksen, kun kaikkia mahdollisia yhdistelmiä ei testata. Kombinaatiotestaus on myös tehokas regressiotestauksessa, jossa testattaville pareille voidaan laskea painoarvot edellisten testien tulosten periaatteella. Tässä tulee mukaan priorisointi.

4.3 Priorisointi

Priorisointi tuli esille useissa artikkeleissa. Priorisointiin suhtauduttiin aina myönteisesti ja sen antamat tulokset olivat myös vakuuttavia. Priorisointi ei kannata ainoastaan silloin, jos aikarajoituksia ei ole ja voidaan testata kaikki testaustapaukset. Reaalimaailmassa ohjelmistoprojekteilla on aina, usein tiukat, aikarajoitukset. Niinpä priorisointi, vaikka sitä ei olisi tehty optimaalisen hyvin, kannattaa aina.

Priorisointia pystytään tekemään monella eri tavalla. Usein testaustapauksille määriteltiin painot niiden kompleksisuuden mukaan ja regressiotestauksessa voidaan ottaa huomioon myös edellisten testien tuottamat tulokset painojen laskentaan. Testaustapaukset sitten järjestetään näiden painojen mukaan testausjärjestykseen tai niistä voidaan satunnaisotoksella ottaa testaustapaukset niin, että tapaukset joilla on isot painot, valitaan todennäköisemmin. Myös ohjelman ja ohjelmoijien historia pitäisi ottaa huomioon

priorisoinnissa.

4.4 Johtopäätökset

Testaus kannattaa suunnitella tuotteen spesifikaation pohjalta. Spesifikaatiota tehdessä tulisi käyttää jotain standardia mallintamistyökalua, esimerkiksi UML:ää. Näin malleille on laaja tuki. Monet testaustapausten valintatyökalut ja -menetelmät myös käyttävät UML:n aktiviteettikaavioita toiminnassaan. Spesifikaatioon perustuvaa testaustapausten luontimenetelmää kannattaa käyttää, koska silloin koodin muutokset eivät vaikuta yhtä herkästi testaustapauksiin. Kun ohjelmasta on kattavat kaaviot, eli ohjelman toiminta on mallinnettu hyvin, voidaan testaus myös suunnitella mallipohjaisesti, jolloin voidaan käyttää hyväksi myös samankaltaisten ohjelmien testaussuunnitelmia.

Myös historia kannattaa ottaa huomioon, erityisesti riskejä arvioitaessa. Historiaan kuuluu sekä testattavan ohjelman aiemmat versiot, että yrityksen aiemmin tuotetut ohjelmistot. Kun edellisistä ohjelmista on dokumentoitu virheet hyvin, niin voidaan näitä tietoja käyttää arvioidessa testattavan ohjelman virheiden mahdollista esiintymistä. Priorisoinnissa ja riskien arvioinnissa tulisi myös kuunnella kaikkia ohjelmistoprojektin osapuolia, jotta riskeistä ja niiden seurauksista saataisiin mahdollisimman tarkka kuva. Näin testaustapauksille voidaan laskea paremmat painoarvot.

Koska suurin osa virheistä johtuu yhdestä tai korkeintaan muutamasta virheestä, kannattaa testauksessa harkita kombinaatiotestausta, jolla testataan ominaisuuspareja. Tämä jättää osan yhdistelmistä testaamatta, mutta on kuitenkin kustannustehokas tapa löytämään suurimman osan ohjelman virheistä.

5 YHTEENVETO

Testauksesta on kirjoitettu paljon artikkeleja. Testaustapausten valinnasta on niinikään kirjoitettu paljon, mutta monet niistä eivät suoraan ole tämän työn aihealueeseen liittyviä. Erityisesti regressiotestauksesta on kirjoitettu paljon.

Spesifikaatioon perustuva testaustapausten luontimenetelmä näyttäisi olevan suositellumpi kirjallisuudessa, koodiin perustuvaan menetelmiin nähden. Syinä voi varmaan pitää nykyisten ohjelmistojen kompleksisuutta ja koodiin perustuvien menetelmien heikkoa skaalaavuutta isoihin ohjelmiin. Koodin perustuvaa menetelmää on kuitenkin myös suositeltu käytettäväksi moduulitesteissä, joissa testattavan kohteen mittakaava on verrattain yksinkertainen.

Varsinaisista valintamenetelmistä malleihin perustuva testaus ja kombinaatiotestaus tulivat eniten esille. Muuten esiteltiin menetelmiä, jotka perustuivat riskien ja ohjelmistovirheiden arviointiin. Kombinaatiotestauksesta oli tehty pieniä tutkimuksia ja se oli todettu tehokkaaksi menetelmäksi löytämään järjestelmän useimmat virheet. Monimutkaiset virheet, jotka johtuvat useasta eri muuttujasta eivät tule kombinaatiotestauksessa ilmi. Koska kombinaatiotestauksessa on heikko kohta, on riskien arviointi myös siinä tärkeää. Kuitenkaan sitä ei oltu huomioitu kaikissa artikkeleissa. Menetelmä on kuitenkin lupaava, mutta siitä pitäisi saada isomman mittakaavan tutkimuksia.

Kirjallisuudessa testaustapausten valinnassa on keskitetty paljolti riskien analysointiin ja testaustapausten valintaan niin, että löydetään mahdollisimman suuri osa testattavan ohjelmiston virheistä. Priorisointi, testien automatisointi ja riskianalyysi on löydetty kustannustehokkaiksi menetelmiksi. Kuitenkin, nämä tutkimukset on tehty pienessä mittakaavassa ja varsinaisia tuloksia niiden soveltumisesta reaalimaailman ohjelmistotuotantoon ei ole. Myös ohjelmien elinkaaria ja riskien seurauksia ei ole otettu huomioon useimmissa artikkeleissa. Nämä ovat kuitenkin tärkeitä alueita, jotka vaikuttavat ohjelmistojen laatuun ja hyväksi havaittujen menetelmien siirtymiseen akateemisesta maailmasta työelämään.

LÄHTEET

[1] ISO/IEC, ISO/IEC 29119-2, Software Testing Standard – Activity Descriptions for Test Process Diagram, 2008

[2] Haikala Ilkka, Märijärvi Jukka. 2004. Ohjelmistotuotanto. Talentum. Valikko -sarja. ISBN 952-14-0850-2

[3] Taipale Ossi, Smolander Kari. Improving software testing by observing practise. *International Symposium on Empirical Software Engineering. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering.* 2006. pp. 262-271

[4] Chen Yanping, Probert Robert L., Sims Paul. Specification-based Regression Test Selection with Risk Analysis. I*BM Centre for Advanced Studies Conference. Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*. 2002.

[5] Muccini Henry, Dias Marcio S., Richardson Debra J.. Towards Software Architecture-based Regression Testing. *International Conference on Software Engneering. Proceedings of the 2005 workshop on Architecture dependable systems.* Session: Workshop on Architeching Dependable Systems. 2005. pp. 1-7

[6] Do Hyunsook, Rothermel Gregg. An Empirical Study of Regression Testing Techniques Incorporating Context and Lifetime Factors and Improved Cost-Benefit Models. *Foundations of Software Engineering. Proceedings of the 14th annual ACM SIGSOFT international symposium on Foundations of software engineering.* 2006. pp. 141-151

[7] Do Hyunsook, Mirarab Siavash, Tahvildari Ladan, Rothermel Gregg. An Empirical Stydy of the Effect of Time Constraints on the Cost-Benefits of Regression Testing. *Foundations of Software Engineering. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 2008. pp. 71-82

[8] Kuhn Rick, Kacker Raghu, Lei Yu, Hunter Justin. Combinatorial Software Testing. *Computer.* Vol. 42. Issue 8. Elokuu 2009. Pp. 94-96. Doi: 10.1109

[9] Krishnan R., Murali Krishna S., Siva Nandhan P. Combinatorial Testing: Learning from our Experience. *ACM SIGSOFT Software Engineering Notes*. Vol. 32. Issue 3. Toukokuu 2007. pp. 1-8. ISSN: 0163-5948

[10] Qu Xiao, Cohen Myra B., Rothermel Gregg. Configuration-Aware Regression Testing: An Empirial Study of Sampling and Prioritization. *International Symposium on Software Testing and Analysis. Proceedings of the 2008 international symposium on Software testing and analysis.* Session: Regression Testing. 2008. pp. 75-86

[11] Lott C., Jain A., Dalal S., Modeling Requirements for Combinatorial Software Testing. *International Conference on Software Engineering. Proceedings of the 1st international workshop on Advanes in model-based testing.* Session: Advances in Model-Based Testing. 2005. pp. 1-7

[12] Bryce Renée C., Colbourn Charles J.. Test Prioritization for Pairwise Interaction Coverage. *ACM SIGSOFT Software Engineering Notes.* Vol. 30. Issue 4. Heinäkuu 2005. pp. 1-7

[13] Pretschner Alexander. Model-Based Testing. *International Conference on Software Engineering. Proceedings of the 27th international conference on Software engineering.* Tutorial Session: Tutorials. 2005. pp. 722-723.

[14] Olinpiew Erika Mir, Gomaa Hassan. Model-based Testing for Applications Derived from Software Product Lines. *International Conference on Software Engineering. Proceedings of the 1st International workshop on advances in model-based testing.* Session: *Advances in Model-Based Testing.* 2005. pp. 1-7.

[15] Pretschner A., Prenninger W., Wagner S., Kühnel C., Baumgartner M., Sostawa B., Zölch R., Stauner T.. One Evaluation of Model-Based testing and its Automation. *International Conference on Software Engineering. Proceedings of the 27th international conference on Software engineering.* Session: Empirical evaluation of testing. 2005. pp. 392-401

[16] Boberg Jonas. Early Fault detection with Model-Based Testing. *Annual ERLANG Workshop. Proceedings of the 7th ACM SIGPLAN workshop on ERLANG.* pp. 9-20.

[17] Redmill Felix, Exploring risk-based testing and its implications. *Software testing, verification and reliability.* vol 14. Issue 1. Maaliskuu 2004. pp. 3-15. ISSN: 0960-0833

[18] Wong W. Eric, Qi Yu, Cooper Kendra. Source Code-Based Software Risk Assessing. *Symposium on Applied Computing.* Session: *Software Engineering: Applications, practises and tools.* 2005. pp. 1485-1490