Ohjelmistojen suunnittelumenetelmät ja -työkalut

Seminaarityö testauksesta

Annemari Auvinen (annauvi@st.jyu.fi)

Anu Niemi (anniemi@st.jyu.fi)

10.6.2002

Sisällysluettelo

1 Testaus osana ohjelmistoprosessia

Vaikka testaus onkin ohjelmistoprosessin viimeinen vaihe, sitä suunnitellaan ja toteutetaan koko prosessin ajan. Testauksen suunnittelu aloitetaan jo analyysi- tai suunnitteluvaiheessa ja sitä tehdään koko toteutuksen ajan. Testauksen tarkoituksena on löytää virheitä sekä osoittaa, että ohjelmisto toimii asetettujen vaatimusten mukaisesti.

Virheiden löytäminen mahdollisimman aikaisessa vaiheessa ohjelmistoprosessia mahdollistaa taloudellisten resurssien paremman hyödyntämisen. Virheiden korjaaminen ohjelmiston julkaisemisen jälkeen on moninkerroin kalliimpaa kuin silloin, jos virhe olisi löydetty jo prosessin aikaisemmassa vaiheessa. Lisäksi virheen löytyminen julkaistusta tuotteesta aiheuttaa yritykselle huonoa mainetta. Kuitenkin testaus on usein se, missä säästetään ja puolivalmis ohjelmisto laitetaan markkinoille ja ajatellaan, että asiakas testaa. Lisäksi yleinen myytti on, että virheet ovat huonojen ohjelmoijien vikoja.

Sopivien testaajien valinta on tärkeää. Joskus uusi työntekijä saattaa olla hyvä valinta, koska hänellä on kova halu löytää virheitä ohjelmistosta ja näyttää omat kykynsä. Toisaalta testausryhmä ei saa myöskään olla kokemattomin, vaan sen pitäisi olla mahdollisimman monipuolinen. Huonoa ohjelmoijaa ei ikinä saa laittaa testaamaan. Testaajan pitäisi olla myös ongelma-alueen asiantuntija ja osata käyttää ohjelmistoa kuten loppukäyttäjä. Ohjelmoijan tekemää omaa ohjelmistoa pitää olla aina testaamassa joku muukin. Ohjelmoija kyllä tietää oman koodinsa vaaranpaikat, mutta saattaa testata sitä liian helläkätisesti.

Koska täydellinen testaus on yleisesti ottaen mahdotonta, testin kattavuudelle asetetaan yleensä arvo, joka on testauksessa läpikäytyjen koodirivien prosentuaalinen osuus kaikista riveistä. Kattavuuden minimivaatimus on tavallisesti 75%. Kattavuutta käytetään luokka- ja komponenttitestauksessa.

Ennen testauksen aloittamista täytyy sopia testauksen aloittamis- ja lopettamiskriteerit. Lisäksi jokaisella testivaiheella täytyy olla tarkasti määritelty lopettamiskriteeri eli ehto, jonka saavuttamisen jälkeen voidaan siirtyä seuraavaan testivaiheeseen. Yleensä tämä ehto koostuu onnistuneiden testitapauksien

suhteellisesta lukumäärästä ja kattavuudesta, jotka pitää saavuttaa. Lisäksi ehdossa on määritelty korjaamattomien virheiden sallittu lukumäärä kategorioittain (kts. luku 1.4), kuitenkin niin, että yleensä fataaleja ja vakavia virheitä ei saa esiintyä lainkaan. Käytännössä testausehtoon kuuluvat usein myös projektin virstanpylväät.

1.1 Testivaiheet

Testaus jakautuu luokka-, komponentti-, integraatio- ja järjestelmätestaukseen sekä mahdollisesti hyväksymistestaukseen. Jokaisessa vaiheessa testataan tiettyä ohjelmiston toiminnallisuutta ja henkilöt, menetelmät ja työkalut vaihtelevat. Kussakin projektissa vaaditut testausvaiheet on määriteltävä projektisuunnitelmassa. Testauksen edetessä siirrytään vaihe vaiheelta yleisemmälle tasolle ja samalla testikattavuus pienenee. Löydettyjä virheitä on vähemmän, mutta ne ovat hankalampia korjata.

1.1.1 Luokkatestaus

Luokkatestaus on erittäin tärkeää etenkin käytettäessä C++-ohjelmointikieltä, joka on osoittimiensa ja muistin hallintansa takia alttiimpi virheille kuin esimerkiksi Java. Käyttöliittymä jätetään yleensä luokkatestauksen ulkopuolelle.

Henkilö, joka on toteuttanut luokan, testaa sen normaalisti jo heti luokan valmistuttua eikä tästä kirjoiteta testisuunnitelmaa. Ajureita kannattaa käyttää aina kun se on mahdollista, koska niiden avulla voidaan säästää aikaa. Ajuri on ohjelma, joka kutsuu luokan metodeja ja se voi toimia myös testisuunnitelmana.

Testikattavuus on luokkatestauksessa normaalisti 75%, mutta käytännössä kannattaa pyrkiä parempaan kattavuuteen. Kirjastoille asetettu kattavuus on yleensä 85%-100% ja turvallisuuskriittisissä ohjelmistoissa vaaditaan täydellistä kattavuutta.

Luokkatestauksessa kaikki metodit tulisi testata kaikilla mahdollisilla syötteillä, erityisesti kaikki syötteiden erikoistapaukset tulisi käydä läpi. Myös kaikki mahdolliset vasteet tulee saada. Toteutusta ei tule ikinä muuttaa vain testikattavuuden saavuttamiseksi. Kattavuuden laskemiseen sekä ajonaikaisten virheiden ja muistivuotojen huomaamiseen kannattaa käyttää työkaluja, joiden mahdollisesti tuottamat raportit käyvät testiraporteista.

1.1.2 Komponenttitestaus

Komponenttitestauksessa testataan luokkien välistä toimintaa. Tätä vaihetta kutsutaan myös moduulitestaukseksi ja siitä kirjoitetaan aina testisuunnitelma. Testiskriptejä ja simulaattoreita kannattaa käyttää apuna, jos se on mahdollista.

Suunnitteludokumenttia käytetään pohjana komponenttitestaukselle. Testauksessa on käytävä läpi kaikki siinä asetetut toiminnallisuudet. Tyypillisiä testitapauksia ovat esimerkiksi käynnistäminen, pysäyttäminen ja tietokantayhteydet.

Testiraporttiin kirjataan päivämäärä ja testaaja, testitapauksen tulos, mahdollisesti kommentteja sekä yhteenveto suoritetuista testeistä.

1.1.3 Integraatiotestaus

Integraatiotestaus suoritetaan, jos ohjelmisto kostuu useammasta kuin yhdestä komponentista. Tässä vaiheessa testauksen kohteena on komponenttien välinen toiminta. Testisuunnitelma voidaan kirjoittaa ja katselmoida arkkitehtuurin suunnittelun jälkeen, kuitenkin viimeistään ennen toteutusta. Testitapaukset luodaan arkkitehtuurisuunnitelmassa kuvatun toiminnallisuuden pohjalta.

Testauksen suorittavat projektin jäsenet yleensä kontrolloiduissa laboratorio-olosuhteissa. Toimivan testausympäristön luomisesta vastaavat tavallisesti muut kuin projektin jäsenet, koska testaajien on pystyttävä keskittymään pelkästään ohjelmiston eikä ympäristön testaamiseen. Integraatiotestauksessa tulisi simulaattoreiden sijaan käyttää oikeaa materiaalia ja oikeita laitteita.

1.1.4 Järjestelmätestaus

Järjestelmätestaus on viimeinen testausvaihe. Se varmistaa, että kaikki toimii niin kuin on suunniteltu. Testaus suoritetaan loppukäyttäjän näkökulmasta käyttötapauksien avulla ja sen tarkoituksena on löytää virheitä järjestelmän toiminnassa. Testauksen suorittavat tavallisesti projektin ulkopuoliset henkilöt.

1.1.5 Vaiheiden toistaminen

Luokkatestaus suoritetaan yleensä vain kerran. Integraatio- tai järjestelmätestauksessa löydetyn virheen korjaamisen jälkeen suoritetaan komponenttitestaus uudelleen.

Iteratiivisessa ja inkrementaalisessa prosessissa testaus suoritetaan monta kertaa. Muutetut tai uudet ominaisuudet tulee testata sekä vanhat testitapaukset ajaa uudelleen.

Aikaisemmat testitapaukset tulee ajaa uudelleen myös aina virheiden korjaamisen tai uuden ominaisuuden lisäämisen jälkeen. Tämän tarkoituksena on varmistaa, etteivät muutokset aiheuta virheitä niihin ohjelmiston osiin, jotka toimivat aikaisemmin oikein.

1.2 Testausmenetelmät

Mustalaatikkotestaus keskittyy toiminnallisuuden testaamiseen loppukäyttäjän näkökulmasta. Ohjelmiston syötteet ja vasteet nähdään, mutta ei varsinaista toteutusta. Testaajalta ei vaadita ohjelmointikokemusta ja yleensä testaaja onkin joku muu kuin suunnittelija, esimerkiksi loppukäyttäjä.

Lasilaatikkotestauksen tavoitteena on virheiden löytäminen toteutuksesta. Sovellus testataan järjestelmällisesti läpikäymällä mm. kaikki funktiot, luokat, dialogit ja käyttöliittymäkomponentit. Testaajalta vaaditaan toteutuksen rakenteen ymmärtämistä, jotta myös mahdolliset heikot kohdat löydetään. Testaaja voi olla suunnittelija tai muu projektin jäsen.

Harmaalaatikkotestaus on mustalaatikko- ja lasilaatikkotestauksen välimuoto. Testaajan täytyy ymmärtää jotakin koodin toiminnasta ja testitapauksia etsitään koodin avulla.

Alhaalta ylöspäin suuntautuvassa testauksessa testaus aloitetaan komponenteista, jotka eivät riipu muista moduuleista, ja seuraavat mukaan otettavat komponentit riippuvat aina edellisistä testatuista komponenteista. Tätä jatketaan kunnes kaikki komponentit on testattu ja yhdistetty toisiinsa. Tämä menetelmä on suosituin komponentti- ja integraatiotestauksessa.

Ylhäältä alaspäin tehtävässä integraatiossa käytetään inkrementaalista lähestymistapaa ohjelman rakenteeseen. Siinä aloitetaan pääohjelmasta ja siirrytään aina alaspäin hierarkiassa. Moduuleita voidaan yhdistää joko syvyys- tai leveyssuunnassa. Syvyyssuunnassa otetaan tavallisesti ensin kaikki tärkeimmässä kontrollipolussa olevat moduulit ja sen jälkeen käydään läpi seuraava haara. Leveyssuuntaisessa läpikäynnissä moduulit otetaan aina taso kerrallaan.

1.3 Testisuunnitelma

Jokainen testauksen vaihe jakautuu testausympäristön dokumentointiin ja testitapausten luomiseen. Testisuunnitelmat täytyy aina katselmoida ja hyväksyä. Testiympäristön dokumentoinnin avulla voidaan ympäristö tarvittaessa luoda uudestaan sekä löytää ympäristön aiheuttamat virheet. Ympäristöstä tulisi dokumentoida mm. käytetyt käyttöjärjestelmät, kääntäjät, verkko-ohjelmistot, tietokannat, testiajurit, testidatan generoijat, testianalysaattorit, laitteiston osat ja ympäristön pystyttämisohjeet.

Testitapauksilla tarkoitetaan skenaarioita, jotka kattavat testattavan asian toiminnallisuuden. Tapaus koostuu esiehdoista, sen kuvauksesta, käytetyistä työkaluista, suoritusohjeista, oletetuista tuloksista sekä tuloksien tarkastusohjeista ja kommenteista, joita ovat esimerkiksi testauspäivämäärä, testin tulos ja testaaja. Jokaisen testitapauksen tulee olla yksikäsitteinen eikä useampi tapaus saa käsitellä samaa asiaa. Testitapauksen tulisi olla niin yksityiskohtainen, että kuka tahansa projektin jäsen tai ulkopuolinen henkilö pystyy sen suorittamaan. Lisäksi testitapauksen tulisi olla yksinkertainen, mielellään alle kymmenen vaihetta yhdessä tapauksessa. Mahdottomia tapauksia kannattaa välttää, kuten operaation suorittamista ja samanaikaista tietokannan alasajoa, jossa operaation suoritus saattaa olla niin nopea, ettei muuta ehditä tekemään.

Testaussuunnitelman yleisiin virheisiin kuuluu yksittäisten toimintojen testaus käyttötapauksien sijaan. Testaaminen eri ympäristöissä saatetaan myös unohtaa. Asennusskiriptien testaus pitää muistaa myös lisätä suunnitelmaan. Beetatestaajien lisäksi pitää muistaa testata muillakin ryhmillä, koska beetatestaaja saattaa pitää tärkeinä aivan eri asioita kuin loppukäyttäjä tai hyväksyä epävakaankin ohjelman, koska ei ole maksanut siitä.

1.4 Testauksen suorittaminen

Testit suoritetaan tehdyn suunnitelman mukaan tapaus kerrallaan. Jos yksi testi epäonnistuu, muut pyritään silti suorittamaan, mikäli tapaukset eivät riipu toisistaan. Jokaisesta suorituksesta raportoidaan komponentin versio, testin tulos ja mahdollisten virheiden kuvaus. Virheet pitää kuvata niin tarkasti, että sama virhe voidaan generoida tarvittaessa uudestaan.

Virheet pitäisi pystyä kategorisoimaan esimerkiksi fataaleihin, vakaviin, vähäisiin ja kosmeettisiin virheisiin. Virhe on fataali, jos jokin toiminnallisuus ei toimi lainkaan. Vakava virhe taas voi olla esimerkiksi vaadittua hitaampi suoritus. Jos jokin ominaisuus on hieman hankala käyttää, mutta se voidaan kiertää helposti, voidaan puhua vähäisestä virheestä. Virheet kannattaa kategorisoida järjestelmällisesti ja lisäksi käyttää alakategorioita. Lisäksi kannattaa luoda omat kategoriat virheille, jotka mahdollisesti korjataan seuraavissa versioissa sekä niille virheille, jotka huomataan myöhemmin aiheettomiksi. Kategoriointijärjestelmästä pitää tehdä mahdollisimman yksinkertainen ja selkeä sekä virheiden luokittelusta kategorioihin tulee antaa selkeät ohjeet testaajille.

Uusien tapausten luonti testin aikana on suositeltavaa. Ne lisätään suunnitelmaan ja voidaan hyväksyä myöhemmin. Testitapauksen ulkopuolella löydetyt virheet pitää myöskin raportoida, koska suunnittelemattomat virheet toteutuvat todennäköisemmin kuin suunnitellut. Näitä virheitä voidaan käyttää myös uusien testitapausten pohjana. Uusia tapauksia tulee luotua automaattisesti myös silloin, kun ohjelmistoa käytetään. Eri ihmiset käyttävät samaakin ohjelmistoa eri tavalla eikä suunnitelmaa usein noudateta kirjaimellisesti.

Käyttöliittymää testataan komponenttitestauksessa sekä integraatio- ja järjestelmätestauksessa, joissa testitapaukset ajetaan käyttöliittymän kautta. Testitapauksia saattaa olla satoja, esimerkiksi ohjelman käynnistäminen ja sulkeminen, avustukset, painikkeiden ja valikoiden toiminnallisuus, siirtyminen ikkunasta toiseen, ikkunoiden koon muuttaminen, ikkunoiden siirtäminen ja sulkeminen, kopiointi ja liittäminen, raahaaminen ja tiputtaminen sekä ohjelman vakaus. Käyttöliittymää testataan käytettävyystestauksella jo käyttöliittymän suunnittelun yhteydessä.

Integraatio- ja järjestelmätestaukseen kuuluu myös usein suorituskyvyn, kuormituksen ja vakauden testaus sekä joskus myös kuormituksesta erillään stressitestaus. Näiden testien jättäminen viime hetkiin on yleinen virhe. Mikäli jotain hälyttävää löytyy, saatetaan joutua tekemään suuria muutoksia ja testaamaan uudelleen, mikä saattaa viedä paljon aikaa. Toisaalta nämä ovat asioita, joita on vaikea testata sovelluksen ollessa keskeneräinen.

Yhteensopivuustestauksessa testataan, että ohjelmisto tukee kolmannen osapuolen toimittamien komponenttien eri versioita ja usein testaus erotetaan varsinaisesta ohjelmiston kehitysprosessista. Hyväksymistestauksen suorittaa asiakas sen jälkeen, kun ohjelmisto on toimitettu. Toimittaja voi avustaa valmistautumisessa esimerkiksi kirjoittamalla testisuunnitelman.

Testausta voi helpottaa ja nopeuttaa huomattavasti sopivilla työkaluilla. Testikattavuuden laskemiseen on olemassa esimerkiksi C++-kielelle CTC++ ja PureCovegare –työkalut sekä Javalle JProbe Coverage ja Optimizeit Code Coverage. Näillä voidaan esimerkiksi laskea kattavuus jokaiselle funktiolle ja tiedostolle.

Ajonaikaisten virheiden ja muistivuotojen havaitsemisen avuksi löytyy C++-kielelle Purify ja Insure sekä Javalle JProbe Profile ja Optimizeit Profiler –työkalut. JUnit on työkalu testiajureiden kirjoittamiseen Java-luokille.

Lisäksi löytyy automatisoituja työkaluja käyttöliittymätestaukseen. Niissä testiskriptit nauhoitetaan kuten makrot, skriptit tukevat useita ohjelmointikieliä ja testit voidaan ajaa hiljaisena aikana. Yksi näistä työkaluista on Vermont HighTest, joka on tarkoitettu Windows-ympäristöön. Rational Robot puolestaan toimii usealla eri alustalla ja sama generoitu skripti voidaan ajaa useilla alustoilla.

Luokkatestaus pystytään helposti automatisoimaan käyttämällä komentotiedostoja esimerkiksi ajurien käynnistämiseen ja tuloksien tarkastamiseen. Komentotiedostoja voidaan käyttää myös komponenttitestauksessa.

Automaattisella testauksella samoja testejä pystytään ajamaan yhä uudelleen ja uudelleen. Se on kuitenkin investointi, jonka edut verrattuna kustannuksiin tulee arvioida. Aivan kaikkea ei pystytä automatisoimaan, esimerkiksi tulosten arviointia, esiehtojen määrittelemistä tai ympäristön asettamista. Automatisointi on kuitenkin

käytännöllistä ja kannattavaa erityisesti inkrementaalisessa ja iteratiivisessa prosessissa tai kun ohjelmiston pitää tukea useita alustoja. Automatisointi ei kuitenkaan korvaa hyvää testisuunnitelmaa.

2 Huomaamatta jääneiden virheiden lukumäärän arviointi

Valitsimme kaksi testaukseen läheisesti liittyvää artikkelia, jotka molemmat käsittelivät huomaamatta jääneiden virheiden lukumäärän arviointia. James Millerin artikkeli [2] käsitteli katselmoinnin jälkeistä tilannetta. Toisessa Stringfellowin, Andrewsin, Wohlinin ja Peterssonin artikkelissa [3] arvioitiin niiden komponenttien lukumäärää, joissa esiintyy virheitä julkaisun jälkeen, mutta ei testauksen aikana. Katselmoinnissa huomaamatta jääneiden virheiden määrän arviointi

Katselmointitapahtuman tarkoitus on löytää dokumenteista mahdollisia virheitä. Mitä aikaisemmassa vaiheessa virheet kyetään havaitsemaan sitä suurempi on säästö. Valitettavasti useat yritykset eivät kuitenkaan määrittele katselmointiprosessiaan tarpeeksi tarkasti eikä esimerkiksi katselmointitapahtuman lopetuspäätös perustu mihinkään suoranaiseen mittariin.

Mittareita on kuitenkin olemassa. Yksi käytetyimmistä on se, että arvioidaan kuinka monta virhettä katselmoitavassa dokumentissa on jäljellä. Tämä arvio voidaan perustaa joko siihen, kuinka paljon virheitä on dokumenteista löytynyt tai historiaan. Lähde [2] keskittyy arvioimaan jäljellä olevien vikojen määrää katselmoinnissa jo löydettyjen vikojen lukumäärän pohjalta.

Katselmointitapahtumaan ja etenkin siihen käytettäviin resursseihin liittyy aina bisnestavoitteet, jotka vaihtelevat tapauskohtaisesti. Mikäli katselmoitavalle tuotteelle on asetettu esimerkiksi korkeat saatavuustavoitteet, on todennäköisesti tärkeämpää, että löydetään jäljellä olevat virheet, kuin se, että minimoidaan resurssien käyttöä. Tällöin käytetään yleensä jäljellä olevien vikojen arvioinnissa estimaattoria, joka mielellään yliarvioi jäljellä olevien vikojen määrän.

Usein tuotteen luotettavuus on tärkeä, mutta ei kuitenkaan kriittinen tekijä tuotteen menestyksen kannalta. Tällöin tuotteen valmistuskustannus on yksi tärkeä tekijä mietittäessä resurssien käyttöä. Näissä tapauksissa voidaan käyttää katselmoinnin jälkeen jäljellä olevien vikojen arviointiin estimaattoria, joka saattaa antaa hieman todellisuutta paremman kuvan.

2.1.1 Capture-recapture -mallit

Capture-recapture-malleissa oletetaan, että kun katselmoitava dokumentti on julistettu katselmoitavaksi, sitä ei enää muuteta. Lisäksi katselmoijan työn tulee olla tasalaatuista, esimerkiksi jos sama dokumentti annettaisiin uudelleen katselmoitavaksi, katselmoijan tulisi löytää yhtä paljon puutteita. Katselmoijien ei tule myöskään paljastaa löydöksiään muille katselmoijille ja heidän pitää dokumentoida kaikki löytämänsä virheet. Lisäksi katselmointiprosessin aikana ei tule hylätä yhtään oikeaa huomiota puutteista. Kaikille katselmoijille tulee toimittaa sama katselmoitavaan materiaaliin liittyvä informaatio.

Capture-recapture-mallia on kehitetty yli 40 vuotta. Tänä aikana tilastotieteilijät ovat luokitelleet malleja kolmen vaihtelevuutta aiheuttavan tekijän, käyttäytymisen, heterogeenisyyden ja ajan, mukaan.

Käyttäytymiseen perustuvissa malleissa yksittäisen virheen löytymistodennäköisyys vaihtelee ajanhetken mukaan. Tämän luokan mallien soveltamista katselmoinnissa käytettäviksi ei ole juuri harkittu.

Heterogeenisyydellä tarkoitetaan sitä, että jokaisella yksittäisellä virheellä on eri todennäköisyys jäädä kiinni. Tähän luokkaan perustuvien arviointimallien toteuttaminen on varsin vaikeaa, koska mallien hyödyntäminen vaatisi kaikkien mahdollisten katselmointiskenaarioiden tyypittämisen. Lisäksi luokittelun tulisi olla toistettavaa.

Ajan mukaan luokiteltaessa virheen löytymisen todennäköisyys vaihtelee ajankohdan mukaan. Katselmointiin sovellettaessa aikatekijänä käytetään yksittäistä katselmoijaa, koska katselmoijien kyky löytää virheitä vaihtelee. Tämän luokan malleista kaikista yksinkertaisin on Lincoln-Peterson-estimaattori. Estimaattorin käyttöä katselmoitiin hankaloittaa kuitenkin se, että kyseinen estimaattori on tarkoitettu ainoastaan kahden katselmoijan tapaukselle.

Vaikka mallit voidaan pääsääntöisesti jakaa näihin luokkiin, estimaattorit voivat myös ylittää luokkien rajat.

2.1.2 Erilaiset estimaattorit

Esiteltävissä kaavoissa M on arvio virheiden lukumäärästä, D on erillisten löydöksien lukumäärä, *N* kuuluu kokonaislukujoukkoon N joka on alkaa luvusta D, n on löydöksien lukumäärä ja t on katselmointitapahtumien lukumäärä.

Null-estimaattori perustuu hypergeometrisiin funktioihin. Muista esiteltävistä estimaattoreista poiketen Null-estimaattoria ei voida luokitella mihinkään ylläolevista luokista.

$$M = \max_{N \in \mathrm{N}} \left[\ln\left(\frac{N!}{(N-D)!} \right) + n\ln(n) + (tN-n)\ln(N-n) - tN\ln(tN) \right]$$

Jacknife-estimaattori on heterogeeninen estimaattori. Sitä käytettäessä f_i-termillä kuvataan sitä, kuinka monesti sama virhe i löydettiin, eli

$$D = \sum_{i=1}^{t} f_i$$

Useita erillisiä estimaattoreita on rakennettu Jacknife-estimaattorin pohjalta. Jatkossa Jacknife-estimaattoreista käytetään termejä J1, J2, J3, J4 ja J5.

$$M_{J1} = D + \left(\frac{t-1}{t} \right) f_1$$

$$M_{J2} = D + \left(\frac{2t-3}{t} \right) f_1 - \left\{ \frac{(t-2)^2}{t(t-1)} \right\} f_2$$

$$M_{J3} = D + \left(\frac{3t-6}{t} \right) f_1 - \left\{ \frac{3t^2-15t+19}{t(t-1)} \right\} f_2 + \left\{ \frac{(t-3)^3}{t(t-1)(t-2)} \right\} f_3$$

$$M_{J4} = D + \left(\frac{4t-10}{t}\right) f_1 - \left\{\frac{10t^2-36t+55}{t(t-1)}\right\} f_2 + \left\{\frac{3t^3-42t^2+148t-175}{t(t-1)(t-2)}\right\} f_3 - \left\{\frac{(t-4)^4}{t(t-1)(t-2)(t-3)}\right\} f_4$$

$$M_{J5} = D + \left(\frac{5t-15}{t}\right) f_1 - \left\{\frac{10t^2-70t+125}{t(t-1)}\right\} f_2 + \left\{\frac{10t^3-120t^2+485t-660}{t(t-1)(t-2)}\right\} f_3 - \left\{\frac{(t-4)^5-(t-5)^5}{t(t-1)(t-2)(t-3)}\right\} f_4 + \left\{\frac{(t-5)^5}{t(t-1)(t-2)(t-3)(t-4)}\right\} f_5$$

Chaon heterogeenisyysestimaattorin mukaan sama virhe löydetään todennäköisimmin kerran tai kahdesti toisin kuin taas Jacknife-estimaattori oletti. Tähän estimaattoriin viitataan merkinnällä Chao H.

$$M = D + \frac{f_1^2}{2f_2}$$

Maximum likelihood -estimaattori MLE voidaan ajatella olevan laajennus Null-estimaattoriin lisäparametrina aika.

$$M = \max_{N \in \mathrm{N}} \left[\ln\left(\frac{N!}{(N-D)!}\right) + \sum_{i=1}^{t} n_i \ln(n_i) + \sum_{i=1}^{t} (N-n_i)\ln(N-n_i) - tN\ln(N) \right]$$

Chaon aikaestimaattoriin viitataan merkinnällä Chao T.

$$M = D\frac{f_1^2 - \sum_{i=1}^{t} Z_i^2}{2(f_2+1)}$$

Chaon heterogeenisyys-aikaestimaattoriin kuuluu kolme eri estimaattoria, joissa todennäköisyys vaihtelee aika- ja heterogeenisyysmuuttujien mukaan. Jatkossa näihin estimaattoreihin viitataan merkinnöillä Chao TH1, Chao TH2 ja Chao TH3.

$$M_1 = 1 - \frac{f_1}{\sum_{k=1}^{t} k f_k}$$

$$M_2 = 1 - \frac{f_1 - 2\frac{f_2}{t-1}}{\sum_{k=1}^{t} k f_k}$$

$$M_3 = 1 - \frac{f_1 - 2\frac{f_2}{t-1} + 6\frac{f_3}{(t-1)(t-2)}}{\sum_{k=1}^{t} k f_k}$$

2.1.3 Tutkimustulokset

Tutkimuksessa ryhmän koon ollessa kolme Chaon mallit tuottivat äärimmäisen suuria maksimiarvoja. Chaon TH -mallit olivat erittäin epävakaita, kuitenkin TH1-mallin mediaani oli lähimpänä nollaa. Aikaluokkaan kuuluvat mallit aliarvioivat virheiden lukumäärän 75% tapauksista, heterogeeniset mallit pärjäsivät paremmin vertailussa. Jacknife-estimaattorit antoivat tässä tapauksessa kaikista stabiileimmat arviot. Etenkin J1-estimaattori vaikutti käyttökelpoiselta kolmen katselmoijan tapauksessa.

Neljän katselmoijan tapauksessa Chaon mallit tuottivat parempia tuloksia. Ne olivat stabiilimpia kuin aiemmin, vaikka edelleen ne antoivat suuria maksimiarvoja. J1-malli oli tässäkin tapauksessa paras vaihtoehto.

J1-malli oli edelleen paras viiden katselmoijankin tapauksessa. Muutkin mallit näyttivät tulevan vakaammiksi, vaikka edelleen aliarviointia tapahtuikin.

Ryhmän koon ollessa kuusi J1-malli oli edelleen paras, mutta tällä ryhmäkoolla sen rinnalla voitaisiin käyttää myös Chaon TH –estimaattoreita, jotka ryhmän koon kasvamisen myötä muuttuivat vakaammiksi.

2.2 Niiden komponenttien lukumäärän arviointi, joissa esiintyy virheitä julkaisun jälkeen, mutta ei testauksen aikana

Komponentit, joissa on virheitä julkaisun jälkeen, mutta ei testauksen aikana, osoittavat, että testausprosessissa on aukkoja. Joko komponentteja ei ole testattu tarpeeksi tai vanhoihin komponentteihin on tullut virheitä uudelleenkäytössä. Viimeksi mainittu on pahempi, koska asiakkaat antavat usein helpommin anteeksi sen, että jokin uusi piirre ei toimi kunnolla, kuin että jo olemassa oleva toiminto lakkaa toimimasta.

Tutkimuksessa keskityttiin nimenomaan virheellisten komponenttien eikä yksittäisten virheiden lukumäärän arviointiin käyttäen capture-recapture ja curve-fitting-malleja. Vertailuna käytettiin yksinkertaista kokemusperäistä metodia. Arvioita voidaan käyttää päätökseen siitä, lopetetaanko testaus ja julkaistaanko tuote vai ei. Lopputuloksesta näkyy, että metodit toimivat melko hyvin.

Kokemuspohjaiset arviointimetodit käyttävät historiatietoja, jotka on saatu saman julkaisun aikaisemmista vaiheista tai aikaisemmista julkaisuista. Sen sijaan capture-recapture ja curve-fitting-metodit pohjautuvat useiden katselmoijien katselmointiraportteihin eivätkä tarvitse historiatietoja.

Tutkimuksessa käytettiin capture-recapture –mallin seuraavia neljää kategoriaa:

1. Testitapauksilla oletetaan olevan sama kyky löytää virheitä ja eri virheet löydetään samalla todennäköisyydellä. Mallia kutsutaan M0. Tästä kategoriasta mukana oli **Maximum-likelihood -metodi (m0ml)**.
2. Testitapauksilla oletetaan olevan sama kyky löytää virheitä, mutta erilaiset virheet löydetään eri todennäköisyyksillä (malli Mh). Tutkimuksessa käytettiin **Jacknife-metodia mhjk**.
3. Testitapauksilla oletetaan olevan eri kyky löytää virheitä ja kaikki virheet löydetään samalla todennäköisyydellä. Mallista Mt mukana olivat **Maximum-likelihood -metodi (mtml)** sekä **Chapman-metodi (mtChpm)**.

4. Testitapauksilla on eri kyky löytää virheitä ja eri virheet löydetään eri todennäköisyyksillä. Mallista Mth mukana oli **Chaon metodi (mthChao)**.

Curve-fitting –mallit ottavat testidataa testitapauksista ja sijoittavat ne matemaattiseen funktioon, jota käytetään arvioinnissa. Perusajatuksena on käyttää datan graafista esitystä arvioinnissa. Tästä on olemassa kaksi mallia:

1. Laskeva malli: Malleissa verrataan löydettyjä virheitä niiden testitapauksien lukumäärään, jotka löysivät virheitä. Virheet järjestetään laskevasti testitapauksien lukumäärän perusteella. Tämän mallin estimaattorit perustuvat Detection Profile –metodiin (Dpm). **Dpm(exp)-estimaattori** perustuu eksponentiaaliseen käyrän sovitukseen, kun taas **dpm(lin)** lineaariseen.
2. Kasvava malli: Malleissa verrataan kumulatiivista löydettyjen virheiden määrää etsimistapahtumien kokonaislukumäärään. Järjestäminen tehdään samoin kuin laskevassakin mallissa. Arviointimallina käytettiin **kumulatiivista metodia**.

2.2.1 Virheellisten komponenttien lukumäärän arviointi

Testauksen alussa testipäällikkö asettaa kynnysarvoksi sallittujen virheellisten komponenttien lukumäärän. Jos testausta suorittaa useampi testiryhmä, voidaan arvioinnissa käyttää capture-recapture ja curve-fitting-malleja. Jokainen testiryhmä raportoi komponentit, joista virheitä löytyi.

1. Jokaisen viikon lopussa kerätään eri testiryhmien antama virhedata komponenteista. Komponentille annetaan arvoksi 0, jos siitä ei löytynyt virheitä, muuten 1.
2. Capture-recapture ja curve-fitting-arviointimetodeja sovelletaan dataan. Arviointi antaa summan niiden komponenttien määrästä, joista löytyi virheitä lisättynä niiden komponenttien lukumäärällä, joissa oletetaan olevan virheitä, vaikka testeissä ei mitään löytynytkään.
3. Kokemuspohjaista arviointimetodia käytetään dataan. Metodi käyttää tekijää, joka lasketaan käyttämällä aikaisempien julkaisujen tietoja. Tekijää käytetään niiden virheellisten komponenttien lukumäärän laskemiseen, jotka olivat virheettömiä testauksessa. Arviointimenetelmä viittaa siis niihin

komponentteihin, joissa järjestelmätestauksen olisi pitänyt löytää virheitä. Arviointia verrataan capture-recapture ja curve-fitting-metodeilla saatuihin arvoihin. Huomaamatta jääneiden virheellisten komponenttien laskeminen:

1. Tekijän laskeminen:

$$f_i = \sum_{k=1}^{i-1} p_k / \sum_{k=1}^{i-1} t_k$$

Kaavassa i on nykyinen julkaisu, f_i on tekijä julkaisulle i, p_k on niiden komponenttien lukumäärä julkaisussa k, joissa esiintyi virheitä vasta julkaisun jälkeen eikä testauksessa ja t_k on niiden komponenttien lukumäärä, joissa ei löytynyt virheitä testauksessa julkaisussa k.

2. Kerrotaan niiden komponenttien lukumäärä, joissa ei löytynyt virheitä testauksessa nykyisessä julkaisussa, lasketulla tekijällä.

$$\lceil f_i * t_i \rceil$$

4. Lasketaan niiden komponenttien lukumäärä, joissa oletetaan löytyvän virheitä jäljellä olevassa järjestelmätestauksessa ja julkaisun jälkeen, mutta jotka ovat nyt virheettömiä.
5. Verrataan niiden komponenttien arvioitua lukumäärää, jotka eivät ole virheettömiä, mutta joista ei ole virheitä raportoituna järjestelmätestauksessa ennalta määritettyyn kynnysarvoon. Tämän perusteella päätetään, lopetetaanko testaus ja julkaistaanko ohjelmisto vai ei.

2.2.2 Case study

Tutkimuksen virhedata tuli suuresta lääketieteellisestä järjestelmästä, joka sisältää 188 ohjelmistokomponenttia. Jokainen komponentti sisältää vaihtelevan määrän tiedostoja, jotka ovat loogisesti yhteydessä toisiinsa. Järjestelmässä on yhteensä noin 6500 tiedostoa. Lisäksi komponentit saattavat sisältää alikomponentteja. Kolmessa julkaisussa lisättiin yhteensä 15 uutta komponenttia ja monia muita komponentteja muokattiin. 188 komponentista 99:ssä oli vähintään yksi virhe ensimmäisessä, toisessa tai kolmannessa julkaisussa.

Kolme testiryhmää saivat järjestelmän testattavakseen samaan aikaan. Ryhmien vastuualueet olivat:

1. Kehitysorganisaation järjestelmätestiryhmä testasi järjestelmää suunnitteludokumentteja vastaan.
2. Sisäinen asiakas testasi järjestelmää spesifikaatiota vastaan.
3. Ulkoinen asiakas testasi järjestelmää toiminnallisen käytön mukaan.

Arviointiin vaikuttivat myös ne seikat, että sisäisellä asiakastestiryhmällä saattoi olla aliraportoituja virheitä ja testiryhmän jäsen saattoi jättää merkitsemättä jo raportoidun virheen. Tämän vuoksi tutkimuksessa ei arvioitu virheiden määrää vaan virheellisten komponenttien määrää.

Tutkimuksessa valittiin kynnysarvoiksi kaksi, viisi ja kymmenen virheellistä komponenttia. Tähän päädyttiin järjestelmätestipäällikön haastattelun perusteella. Tälle järjestelmälle 2-5 virheellistä komponenttia oli vielä hyväksyttävissä, mutta kymmenen ei. Arvot tulisikin valita sen mukaan, mitkä ovat järjestelmän laadulliset vaatimukset.

Eri arviointimenetelmien arvioiminen

Kolmen testiryhmän tapauksessa tulokset olivat rohkaisevia: suhteelliset virheet osoittivat, että suurin osa arviointimenetelmistä osui lähelle oikeita arvoja, lukuunottamatta mthChao-menetelmää, joten se ei ole suositeltava.

Kaiken kaikkiaan kokemuspohjainen estimaattori toimi parhaiten, sen sijaan mthChao, dpm(exp) ja kumulatiivinen yliarvioivat liikaa. Kaikissa julkaisuissa capture-recapture ja curve-fitting-estimaattoreista parhaiten toimivat mtml, dpm(lin), m0ml ja mhjk ja ne olivat melkein yhtä hyviä kuin historiatietoja tarvitseva metodi. Mtml ja dpm(lin) aliarvioivat hieman, kun taas mhjk hieman yliarvioi.

Kahden testiryhmän tapauksessa testiryhminä olivat kehitysorganisaation järjestelmätestausryhmä ja ulkoinen asiakas, mikä on varmasti yleisin tilanne. Luonnollisesti voitaisiin olettaa, että capture-recapture ja curve-fitting-metodit antaisivat vähemmän tarkkoja ennusteita, kun katselmoijien lukumäärä laskee.

Minkään estimaattorin tulos ei kuitenkaan eronnut oikeasta arvosta tai muista arvioista huomattavasti. Mtml, m0ml, dpm(lin), mtChpm ja mhjk-estimaattorit toimivat parhaiten. MthChao, kumulatiivinen ja dpm(exp) taas huonoiten. M0ml,

mtml, mhjk ja mtChpm olivat melkein yhtä hyviä kuin kokemuspohjainen metodi eivätkä ne tarvitse historiatietoja. Mikäli historiatietoja on saatavilla ja julkaisut ovat samankaltaisia, niin kokemuspohjaista metodia tulisi kuitenkin käyttää.

Aikaisemmilla viikoilla saatuja arviointeja ja niihin perustuvia päätöksiä arvioitiin kaikissa kolmessa julkaisussa käyttäen sekä kahta että kolmea testiryhmää. Koska tiedettiin, että osa estimaattoreista toimii tässä vaiheessa huonosti, valittiin arvioitaviksi m0ml, mtml, mhjk ja dpm(lin), mtChpm-estimaattorit ja kokemuspohjainen metodi.

Kaiken kaikkiaan mhjk-estimaattori yliarvioi sekä kolmen että kahden testiryhmän tapauksessa. M0ml, mtml ja mtChpm toimivat lähestulkoon yhtä hyvin kummassakin tapauksessa. Dpm(lin) toimi ensimmäisessä julkaisussa paremmin, toisessa ja kolmannessa huonommin.

Arviointimenetelmiin pohjautuvan julkaisupäätöksen arvioiminen

Koska osa estimaattoreista toimi huonosti, kaikkia ei otettu mukaan. M0ml, mtml, mhjk ja dpm(lin)-metodeita käytettiin kolmen testiryhmän tapauksessa. Näiden lisäksi mtChpm-estimaattorilla arvioitiin kahden testiryhmän tapauksessa. MthChao, dpm(exp) ja kumulatiiviset metodit jätettiin ulkopuolelle, koska niiden suhteelliset virheet olivat liian suuret ja testaus olisi jatkunut liian kauan.

Julkaisupäätöksen arviointi testauksen viimeisellä viikolla kolmelle testiryhmälle

Kynnysarvoja verrattiin niiden komponenttien lukumäärään, joissa ilmeni julkaisun jälkeisiä virheitä, mutta jotka olivat virheettömiä testauksessa ja sen perusteella pääteltiin, oliko julkaisupäätös oikea.

Ensimmäisessä julkaisussa m0ml ja mhjk-estimaattorit antoivat oikean vastauksen useimmiten. M0ml suositteli lopettamista hieman liian aikaisin, kun taas mjhk suositteli testauksen jatkamista hieman liian pitkään. Kuitenkin testauksen jatkuminen hieman liian kauan on useimmissa tapauksissa suositeltavampaa kuin liian aikaisin lopettaminen. Toisessa julkaisussa mtml ja dpm(lin) suoriutuivat parhaiten, molemmat antoivat kolme oikeaa päätöstä. M0ml ja mhjk suosittelivat testauksen jatkamista hieman pitempään kuin muut. Kolmannessa julkaisussa mtml oli ainoa

menetelmä, joka antoi kolme oikeaa päätöstä. Dpm(lin)-estimaattori olisi lopettanut testauksen liian aikaisin. M0ml ja mhjk-menetelmiin perustuvat päätökset olisivat jatkaneet testausta kauemmin kuin muut.

Kokemuspohjaiseen metodiin perustuvat arvioinnit analysoitiin myös tästä näkökulmasta. Koska päätökseen tarvitaan historiatietoja, voitiin arviointi tehdä vain toiselle ja kolmannelle julkaisulle. Verrattuna capture-recapture ja curve-fitting –metodeihin perustuviin päätöksiin kokemuspohjainen metodi toimi hyvin. Toisessa julkaisussa kokemuspohjainen metodi antoi saman tuloksen m0ml:n kanssa. Kolmannessa julkaisussa metodi antoi saman tuloksen mtml:n kanssa.

Julkaisupäätöksen arviointi testauksen aikaisemmassa vaiheessa kolmelle testiryhmälle

Ensimmäisessä julkaisussa m0ml, mtml ja dpm(lin) suosittelivat testauksen lopettamista jo niinkin aikaisessa vaiheessa kuin viisi viikkoa ennen testauksen loppumista. Mhjk antoi oikean päätöksen kynnysarvoilla kaksi ja viisi kuudella viimeisellä testausviikolla, arvolla kymmenen se teki oikean päätöksen kaikilla muilla viikoilla paitsi neljänneksi viimeisellä ja viimeisellä viikolla. Näillä viikoilla mhjk suositteli testauksen jatkamista, vaikka oikea päätös olisi ollut lopettaminen.

Suurin osa estimaattoreista lukuunottamatta mhjk:ta paransivat tulosta toisessa ja kolmannessa julkaisussa. Mhjk olisi suositellut testauksen jatkamista kaikilla viikoilla kaikissa kynnysarvoissa. Antaessaan vääriä päätöksiä m0ml, mtml ja dpm(lin) suosittelivat testauksen lopettamista liian aikaisin. Mhjk:n antaessa väärän päätöksen, se yleensä suositteli jatkamista, kun oikea päätös olisi ollut lopettaa. Kokemuspohjainen metodi suoriutui erittäin hyvin toisessa ja kolmannessa julkaisussa, joissa historiatiedot oli saatavilla.

Jos halutaan säästää aikaa, kannattaa valita m0ml ja kokemuspohjaiset metodit. Mhjk-estimaattoria suositellaan siinä tapauksessa, että testaajat haluavat olla konservatiivisempia.

Julkaisupäätöksen arviointi testauksen viimeisellä viikolla kahdelle testiryhmälle

Chapman-estimaattori oli mukana, koska sillä oli vähän suhteellisia virheitä. M0ml, dpm(lin) ja mtChpm antoivat oikeat päätökset kahdelle kynnysarvolle ensimmäisessä julkaisussa. M0ml, mtml, dpm(lin) ja mtChpm suoriutuivat hyvin toisessa julkaisussa, kaikki kolme antoivat kolme oikeaa päätöstä. Kaikki estimaattorit dpm(lin)-metodia lukuunottamatta suoriutuivat samanvertaisesti kolmannessa julkaisussa. Mhjk antoi kolme oikeaa päätöstä ensimmäisessä ja kolmannessa julkaisussa, mutta toisessa julkaisussa se suositteli jatkamaan liian kauan. M0ml, mtChpm ja mhjk näyttivät olevan parhaat estimaattorit kahden testiryhmän tapauksessa.

Toisessa julkaisussa kokemuspohjainen metodi toimi paremmin kuin capture-recapture ja curve-fitting-metodit. Se suositteli jatkamista hieman kauemmin kuin mitä oikea päätös oli, mutta ei niin kauan kuin mhjk. Kolmannessa julkaisussa kokemuspohjainen metodi toimi yhtä hyvin kuin useat muut estimaattori mukaan lukien mhjk. Mikäli historiatietoja on saatavilla ja julkaisut ovat samanlaisia virheiden kannalta, kokemuspohjaista metodia tulisi käyttää täydentämän capture-recapture ja curve-fitting -metodeita.

Julkaisupäätöksen arviointi testauksen aikaisemmassa vaiheessa kahdelle testiryhmälle

Oikeiden päätösten lukumäärän mukaan arvioituna mhjk, m0ml ja mtChpm suoriutuivat kaiken kaikkiaan parhaiten. Mikäli halutaan konservatiivinen päätös, mhjk:ta tulisi käyttää, muutoin m0ml-estimaattoria.

Tässä tutkimuksessa parhaat tulokset saatiin käyttämällä kahta testiryhmää ennemmin kuin kolmea. Sisäinen asiakas saattaa jättää virheitä raportoimatta, millä on vaikutusta kolmen testiryhmän tapauksessa, joten tällaiselle ympäristölle suositellaan kahta testiryhmää.

3 Testauksen liittyminen muihin luentoihin sekä aikaisemmin käytyihin kursseihin

Luennoilla käsiteltiin lähinnä ohjelmistoprosessin eri vaiheita, joista viimeinen on testaus. Testaus on kuitenkin yhteydessä muihin vaiheisiin, sitä suunnitellaan ja toteutetaan eri muodoissa kaikissa prosessin vaiheissa. Luentosarjassa oli oma luentonsa käytettävyystestauksesta, joka on erittäin tärkeä osa testausta, vaikka se toteutetaankin jo käyttöliittymäsuunnittelun aikana ja ymmärretään monesti erillisenä osana. Toki käyttöliittymää testataan myös varsinaisessa testausvaiheessa.

Testausta on aiemmin käsitelty myös Ohjelmistotuotannon sekä Ohjelmistoprojektin vaiheet ja OMT++ -kursseilla, joissa sisältö oli samankaltainen kuin tälläkin kurssilla ja asiaa käsiteltiin melko yleisellä tasolla. Sen sijaan lisäarvoa tälle luennolle toi erilaisten työkalujen esittely, vaikka niiden esittelyä olisi toivonut olevan enemmänkin, mainitaanhan työkalut kurssin nimessäkin.

Työprojekteissa testaus aloitettiin siinä vaiheessa, kun ohjelma oli valmis, eikä sitä suunniteltu oikein mitenkään. Mielestämme projekteissa tulisi kiinnittää hieman enemmän huomiota testauksen suunnitteluun ja toteuttamiseen. Testaus pitäisi sielläkin ottaa mukaan yhtä tärkeänä osana projektia kuin itse ohjelman suunnittelu ja toteuttaminenkin. Tosin työprojektien aika- ja henkilöresurssit asettavat rajoituksia.

4 Lähdeluettelo

[1] Yomi Solution Oy, Testing-luentomateriaali, 2002

[2] James Miller, Estimating the number of remaining defects after inspections, Software Testing, Verification and Reliability, Volume 9 Issue 3, 1999

[3] C.Stringfellow, A. Andrews, C. Wohlin ja H. Petersson, Estimating the number of components with defects post-release that showed no defects in testing, Software Testing, Verification and Reliability, Volume 12 Issue 2, 2002