TEKNILLINEN KORKEAKOULU
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan tutkinto-ohjelma

Automaattinen yksikkötestaus

Kandidaatintyö

Korhonen Olli-Pekka

Tietotekniikan laitos
Espoo 2008

TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto
Tietotekniikan tutkinto-ohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä: Korhonen Olli-Pekka

Työn nimi:
Automaattinen Yksikkötestaus

Päiväys: 5.10.2008 **Sivumäärä:** 24
Pääaine: Ohjelmistotuotanto ja liiketoiminta
Vastuuopettaja: Lauri Savioja
Työn ohjaaja: Jari Vanhanen

Monimutkaisemmat ohjelmat ja kireämmät aikataulut ovat nykypäivän ”trendejä” ohjelmistotuotannossa. Nämä ”trendit” vaativat yhä enemmän kehitysprosessilta, varsinkin testausprosessilta ja muutosten hoitamiselta. Automaattista yksikkötestausta on tarjottu vastaukseksi näihin ongelmiin.

Tässä tutkielmassa kysyn: Voiko automaattisella yksikkötestauksella parantaa ohjelman laatua, prosessia ja onko se kustannustehokasta. Tutkielmassa tämä on rajattu kahteen alakysymykseen: Mitkä ovat automaattisen yksikkötestauksen hyödyt ja mitä pitää tehdä, jotta nämä hyödyt saavutettaisiin.

Tutkimus on puhtaasti kirjallisuus tutkielma, jossa on etsitty artikkeleita ja tutkielmia automaattisesta yksikkötestauksesta, varsinkin sen oikean elämän implementoinneista. Tätä ei täysin tavoiteta tässä tutkielmassa, vaan monet lähteenä käytetyistä artikkeleista käsittelevät automaattista testausta kokonaisuudessaan, eivätkä vain automaattista yksikkötestausta.

Tutkimuksen alussa luettelen automaattisen yksikkötestauksen mahdolliset hyödyt. Tämän jälkeen siirryn selvittämään mitä tarvitsee tehdä, jotta nämä hyödyt saavutetaan. Tähän sisältyy automaattisen yksikkötestauksen implementoinnin yleinen strategia, yleisimmät sudenkuopat ja miten väistää nämä sudenkuopat. Loppu puolella syvennytään laatuun automaattisessa yksikkötestauksessa eli tarkemmin siihen, minkälainen on hyvä automaattinen yksikkötesti, mikä on kustannustehokas yksikkötesti ja miten automaattisia yksikkötestejä tulee ylläpitää.

Loppusanoissa ja yhteenvetona todetaan, että automaattinen yksikkötestaus voi olla vastaus kiristyneisiin aikatauluihin, testauksen vahvistamiseen ja muutosten hallitsemiseen, mutta se vaatii paljon työtä, jonka harva tiedostaa. Tämän takia moni automaattisen yksikkötestauksen implementointi projekti joko epäonnistuu tai jää paljon tavoitteistaan. Samoin se ei sovi kaikkiin projekteihin.

Avainsanat: Ohjelmistotuotanto, Testaus, Automaattinen yksikkötestaus, xUnit
Kieli: Suomi

Lyhenteet / Käsitteet

Yksikkötesti : Tässä työssä yksikkötestillä tarkoitetaan koodin pätkää joka käyttää toista koodin pätkää tavoitteenaan testata tätä. Testi voi olla joko puhdas yhden toiminnon testaaja tai jonkin suuremman kokonaisuuden alusta loppuun suorittava testi. Esim. kaikki testit joita voidaan kirjoittaa xUnit(http://en.wikipedia.org/wiki/XUnit) työkalulla.

AUT: ***A**utomated **U**nit **T**esting*. Automaattinen yksikkö testaus. Automaattisella yksikkötestauksella tarkoitetaan tässä tutkimuksessa käsintehtyjä testejä joita ajetaan automaattisesti

ATML ***A**utomated **T**esting **L**ifecycle **M**ethodology.*

PDCA **PDCA**-sykli (**Plan, Do, Check, Act**) on ongelman ratkaisumalli ja kehittämismenetelmä.

Sisällyluettelo

Luku 1

1. Johdanto

Nykypäivän monimutkaiset ohjelmat vaativat yhä enemmän testausprosessilta joka voi olla aikaa vievää ja samalla kuitenkin halutaan toimittaa tuote asiakkaalle mahdollisimman nopeasti. Tämä ristiriita on varmasti tuttu monelle ohjelmistokehityksen parissa työskentelevälle. Aikataulujen kiristyminen ja vaatimus pystyä reagoimaan nopeasti muutoksiin on luonut pohjan ketterille menetelmille. Nämä uuden sukupolven ohjelmistokehitysprosessit nojaavat laatu asioissa vahvasti yksikkötestaukseen [1][2], mutta yksikkötestejä pidetään yleisesti raskaina sekä tehdä että ylläpitää ja tämä sotii kiristyviä aikatauluja vastaan. Niin kuin Ellis & al.[3] toteavat tutkimuksensa yhteenvedossa ”However, unit tests are difficult to maintain over the life of a project so some practical strategy is required to integrate their use with the customer’s and management’s desire to reduce time scales and cost (faster, better, cheaper–pick any two).” Voiko hyvin suunniteltu automaattinen yksikkötestaus tuoda helpotusta tähän ongelmaan?

Tässä työssä haen vastausta ohjelmistotuotannon, ehkä yleisimpään kysymykseen oli, alue mikä tahansa: Voiko tällä tavalla parantaa tuotetta/lopputulosta ja onko se kustannustehokasta? Olen ottanut lähestymissuunnakseni seuraavat kaksi kysymystä:

1. Mitkä ovat automaattisen yksikkötestauksen hyödyt ?
2. Mitä pitäisi tehdä ja mitä varoa, jotta nämä hyöyt saataisiin realisoitua ?

Aloitan ensimmäisestä kysymyksestä luvussa kaksi. Siinä käyn läpi kirjallusuudessa yleisimmin esiintyneet automaattisen yksikkötestauksen hyödyt. Luvussa kolme isken kiinni toiseen tutkimus kysymykseeni, antamalla muutaman yleisstrategian automaattiseen yksikkötestaukseen, listaamalla mahdollisia sudenkuoppia ja kertomalla miten välttää näitä sudenkuoppia. Neljännessä luvussa menen tasoa alaspäin ja tutkin miten saavuttaa korkealaatuista automaattista yksikkötestausta, tämä luku tarkentaa monia asioita, jotka käsiteltiin vain pinnallisesti luvussa kolme.

1.1 Tutkimusmenetelmä

Tämä työ on kirjallisuustutkielma aiheesta, jota ei voi matemaattisin kaavoin todistaa, eikä ole olemassa standardia automaattisesta yksikkötestauksesta. Olen pyrkinyt aineistokseni löytämään kokemuksia todellisenelämän implementoinneista. Pääasialliset tiedonhakupaikat olivat ”The ACM digital library”* ja IEEE Xplore**. Hakusanoina käytin ”auto / automatic / automated”, ”unit” ja ” test / testing” sekä näiden kaikkia mahdollisia yhdistelmiä. Tein myös muutamia tarkennettuja hakuja käyttäen edellä mainittujen kanssa sanoja ”economic”, ”experience” ja ”implementation”. Otin myös lähteisiin muutaman kirjan, joita käytin kuitenkin vain yleisten väitteiden löytämiseen ja automaattisen yksikkötestauksen perusteiden tutkimiseen.

ACM:stä ja IEEE:stä hauillani saamani artikkelit, joita oli tuhansia, kävin ensiksi läpi otsiko tasolla ja kun näin lupaavan otsikon luin sen pitemmän kuvauksen, jos artikkeli käsitteli edes yhtä tutkimuskysymyksistäni otin sen. Läpikäytäväksi selvisi lopulta noin viitisenkymmentä artikkelia. Lopullisen valikoinnin artikkeleista tein, kun olin lukenut ne läpi, jolloin niitä jäi viitteistä löytyvät kaksitoista kappaletta.

* http://portal.acm.org/
** http://ieeexplore.ieee.org

Luku 2

2. Automaattisen yksikkötestauksen hyödyt

Automaattista yksikkötestausta käsittelevissä kirjoissa, sekä sitä kouluttavien konsulttien puheista löytyy pitkä lista automaattisen yksikkötestauksen mahdollisista hyödyistä. Tässä luvussa käyn mielestäni yleisimmät ja eniten mainostetut läpi.

2.1 AUT-testeille löydät ongelmat koodista aikaisessa vaiheessa

On yleistä tietoa ohjelmistotuotannossa, että mitä aikaisemmin virheet löydetään sitä halvempaa niiden korjaus on. Automaattiset yksikkötestit mahdollistavat koodin testaamisen tehokkaasti ja ajoissa, kauan ennen kuin koodi päätyy asiakkaalle tai edes testi ryhmälle. [4]

2.2 Automaattinen yksikkötestaus = regressiotestaus

Automaattiset yksikkötestit varmistavat koodisi toiminnan tulevaisuudessakin, kerran kirjoitetut testit voidaan ajaa koska vain uudestaan ja näin varmistua että koodi toimii yhä. Tämä pätee myös, jos koodiin tehdään muutoksia, koska kun muutokset on tehty ajetaan automaattiset testit, jolloin varmistutaan siitä, että vanha koodi toimii vieläkin.[5]

2.3 Kehittäjät uskaltavat helpommin muuttaa koodia

Kun ohjelma kasvaa kehittäjät eivät enää välttämättä muista täsmälleen miten jokainen ohjelman osa toimii. Tämä johtaa siihen, että he eivät uskalla muuttaa olemassa olevaa koodia, koska pelkäävät sen rikkovan jotain. Tämä on pahasta, koska tällöin kehittäjät kirjoittavat ennemmin kokonaan uusia osia koodia, eivätkä hyväksi käytä vanhoja osia. Tämä johtaa, varsinkin pitempään kehityksessä olevissa ohjelmissa, huonolaatuiseen koodin, jota on vaikea

ylläpitää.

Automaattiset yksikkötestit tuovat tähän ratkaisun. Ne voidaan ajaa pienenkin muutoksen jälkeen helposti, jolloin voidaan todeta, joko koodin toimivan tai löytää helposti testissä ilmenneet ongelmat.[5]

2.4 AUT-testit toimivat aikaisen vaiheen dokumentointina

Automaattiset yksikkötestit ovat periaatteessa kuvauksia siitä miten ohjelman tulisi toimia. Hyvin tehtyinä ne helpottavat henkilöä, joka ei tunne koodia entuudestaan, sen ymmärtämisessä. Tämä auttaa varsinkin kehittäjiä, koska varsinkaan projektin alkupuolella, minkään muun laista dokumentaatiota ei välttämättä koodista löydy.[6]

2.5 AUT-testit mahdollistavat ketterämmän kehityksen

Automaattiset yksikkötestit mahdollistavat nopean vastaamisen muutokseen. Esimerkiksi joskus saattaa tulla tilanne, jossa pitäisi korjata virhe ohjelmassa ja toimittaa korjaus nopeasti käyttäjille. Automaattiset yksikkötestit ovat näissä tilanteissa kultaakin arvokkaampia: tehdään korjaus ja ajetaan automaattiset yksikkötestit, jolloin varmistutaan, että uusia ongelmia ei ole tullut koodiin.

2.6 Yhteenveto

Tarkkaavaisimmat lukijat ehkä huomasivat, että en kritisoinut millään tavalla aiemmin mainittuja hyötyjä automaattisesta yksikkötestauksesta. Tämä johtuu siitä, että mielestäni nämä ovat kaikki oikeita hyötyjä, mitä on mahdollista saavuttaa tekemällä automaattista yksikkötestausta hyvin, painotus tässä kohtaa sanalla hyvin. Näistä varmasti ollaan kuitenkin montaa mieltä ja näiden hyötyjen paikkansa pitävyys voisi myös olla mielenkiintoinen tutkimuksen aihe, mutta olen rajannut sen pois tämän tutkielman alueesta. Mikä tässä luvussa jäi myös mainitsematta on se, että nämä hyödyt eivät putoa suoraan käteen, kuin kypsä omena, vaan vaativat kovaa työtä. Seuraavat luvut ovat omistettu määrittämään, mitä tämä kova työ pitää sisällään.

Luku 3

3. Automaattisen yksikkötestauksen strategiat ja sudenkuopat

Tässä luvussa tutkin yleisimpiä ongelmia ja hyviä strategioita automaattisen yksikkötestauksen implementoinneissa. Alussa esittelen yleisstrategian automaattisen yksikkötestauksen implementoitiin, tämän jälkeen käyn läpi erilaisia sudenkuoppia mitä implementoinnissa saattaa tulla vastaan. Loppupuolella tutkin mitä tulisi tehdä, jotta menestyisi automaattisessa yksikkötestauksessa.

3.1 Yleisstrategia

PDCA-malli on hyväksi todettu perusmalli prosessin kehittämiseen, joten se sopii myös hyvin automattisen yksikkötestauksen implementointiin ja kehittämiseen. Suunnittelu vaiheessa paneudutaan siihen, mitä halutaan saavuttaa automaattisella yksikkötestauksella ja tämän pohjalta valitaan työkalut siihen. Teko vaiheessa laitetaan suunnitelmat toimeen. Tarkastus vaiheessa katsotaan miten meni, esimerkiksi erilaisten mittarien perusteella ja viimeisessä vaiheessa tehdään korjaukset ja aloitetaan sykli alusta.

PDCA:sta on myös kehitetty oma versionsa automaattiselle testaukselle, nimeltään ATML(automated testing lifecycle methodology). En tiedä onko tämä paras mahdollinen strategia, koska en löytänyt empiirisiä tutkimuksia asiasta, mutta esimerkiksi Perssonin ja Yilmaztürkin[7] ABB:llä tekemä empiirinen tutkielma mainitsee, että ABB käytti ATML:lää implementoidessaan automattista testausta projekteihinsa. Mainittakoon vielä, että tämä strategia on tarkoitettu kaikelle automaattiselle testaukselle eikä vain automaattisille yksikkötesteille, mutta se miten tässä työssä AUT-testi on määritelty eroaa vain GUI testauksen osalta siitä mitä ATML:ssä automaattisella testauksella tarkoitetaan. Ainiin ja tietenkin strategiasta löytyy, niin kuin kaikista hyvistä ohjelmistotuotannon strategioista, kaikkien rakastama vaihe kuvaaja(Kuva 3).

Strategian on alkujaan esitellyt Elfriede Dustin kirjassaan ”Automated Software Testing:

Introduction, Management, and Performance ”[12]. Se koostuu kuudesta vaiheesta:

1.Päätös automatisoida

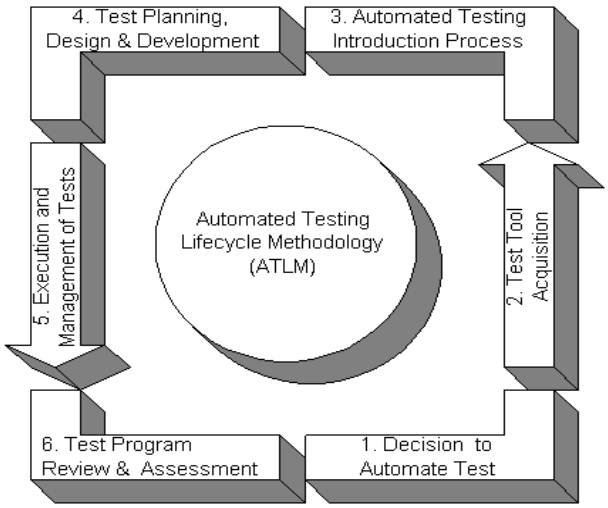
2.Testi työkalun hankinta

3.Automaattisen testauksen esittely prosessi

4.Testien suunnittelu

5.Testien suorittaminen ja hallinta

6.Testi ohjelman läpikäyminen ja arviointi



Tämä ei välttämättä ole paras strategia, mutta ainakin Elfrieden mukaan tällä välttää suurimmat ongelmat, kuten testauksen ilman testaus prosessia(ad-hoc testing), väärän testityökalun valinnan ja huonosti suunnitellut testit. Hän myös kertoo sen soveltuvan minkälaiseen kehitysprosessiin tahansa, oli se sitten vesiputous tai ketterä malli.

3.2 Yleisimmät sudenkuopat

Tässä kappaleessa olen luetellut yleisimpiä sudenkuoppia automaattisessa yksikkötestauksessa. Esittämäni sudenkuopat pohjautuvat pääasiallisesti Perssonin ja Yilmaztürkin [7] tutkimukseen, jossa he käsittelevät automaattisen yksikkötestauksen implementointia sekä kirjallisuuspohjalta, että kertovat käytännön kokemuksia ja havaintoja ABB suoritetuista projekteista, joissa implementointiin automaattinen yksikkötestaus.

En ole laittanut ongelmia tärkeys järjestykseen, mutta jonkinlainen ohjenuora on mielestäni se, että mitä aikaisemmassa vaiheessa prosessia se on, sitä suuremmat vaikutukset sillä on ja sitä kalliimpi ongelmaa on enää korjata myöhemmin. Ongelmat on jaettu kahteen ryhmään: ongelmiin suunnittelussa ja ongelmiin itse tekemisessä.

3.2.1 Automaattisen yksikkötestauksen suunnittelu vaihe

Epäkypsä organisaatio: Organisaatio joka on tarpeeksi kypsä hallitakseen prosessin hallitun kehittämisen on minimivaatimus automaattisen yksikkötestauksen implementoinnille. Ilman tätä AUT-testauksen tuominen voi johtaa kaaokseen

tuotannossa.[7]

Organisaation/projektin johdon mukaan saaminen: Tämä on tietenkin tärkeätä kaikissa tilanteissa joissa tuodaan muutosta organisaatioon, mutta AUT-testien kohdalla tämä tulee esille varsinkin kun keskustellaan resursseista. Liian vähän rahaa ja liian vähän henkilöstöä on varma tapa epäonnistua ja jos organisaation/projektin johto ei ole ajatuksen takana näitä on hyvin vaikea saada.[7]

Regressiotestauksen ymmärtäminen: On tärkeätä, että ymmärretään regressio testauksen käsite. Tulee kalliiksi, jos automaattisia yksikkötestejä käytetään vain manuaalisten testien sijasta eli ajatus, että korvataan kokonaan manuaalinen testaus automaattisella testauksella johtaa tuhoon. Samoin jos ei tarkkaan määritellä mitä automatisoidaan ja mitä ei voi johtaa suuriin ongelmiin.

Kustannustehokkuuden arviointi: Vääränlainen lähestymistapa AUT-testejen tekemiseen johtaa helposti kustannustehottomuuteen. Ohjeet miten ja mitä testejä automatisoida ovat erittäin tärkeitä, kun halutaan tuottaa kustannustehokkaita AUT-testejä. Siirtyminen pelkästä manuaalisesta testauksesta automattiseen testaukseen, ei itsessään vähennä testaukseen meneviä kuluja.[7][8]

Työkalun valinta ja arviointi: On tärkeää varmistaa, että valittu testityökalu tai ympäristö tukee kaikkia asioita, mitä projektissa on tarvetta testata. Väärin valittu työkalu aiheuttaa vähintään lisäkustannuksia kun sitä joudutaan muokkaamaan, mutta pahimmillaan se saattaa tehdä testauksesta tehotonta ja siten johtaa AUT-testauksen kaatumiseen. Ongelma tämän arvioinnissa on monesti siinä, että ei yksinkertaisesti tiedetä mitä ominaisuuksia työkalun pitäisi sisältää, tämän takia on järkevää testata muutamia ennen kuin tehdään päätös käyttää jotain tiettyä työkalua. [7]

Testi suunnitelma: Suunnitelma automaattiselle yksikkötestaukselle pitäisi aina tehdä, joko omana kokonaisuutenaan tai osana yleistä testisuunnitelmaa. Ilman hyvin määriteltyä testisuunnitelmaa testaus ajautuu helposti ad-hoc testaukseksi, joka saattaa johtaa täysin tuntemattomaan testien kattavuuteen, sekä testien ylläpidon vaikeutumiseen. [7][9]

3.2.2 Automaattisen yksikkötestauksen toteutus vaihe

Terminologia: On tärkeää määrittää yhteinen terminologia, kun puhutaan AUT-testauksesta. Terminologian määrittelemättömyys saattaa johtaa väärin ymmärryksiin ja sekaannuksiin. Hyvä tapa tähän, on tehdä testaussanakirja, jossa kaikki termit on määritelty tarkasti. [7]

Pätevyys ja koulutus: On turha olettaa, että kehittäjät ja testaajat osaisivat laatia hyviä automaattisia yksikkötestejä ilman kunnollista koulutusta. Toinen asia, mitä pitää välttää, on laittaa uudet/kokemattomat kehittäjät tekemään testausta, sillä automaattinen yksikkötestikoodi on aivan yhtä tärkeää kuin normaalikin koodi jos ei jopa tärkeämpää. Huono laatuinen testikoodi automaattisessa testauksessa johtaa vähintäänkin korkeisiin ylläpito kustannuksiin ja pahimmillaan automaattisen testauksen kuolemiseen. [7][9][1]

Testien kattavuus: Tuntematon automaattisten yksikkötestien kattavuus, varsinkin kun

niitä käytetään regressiotestaukseen, voi johtaa siihen, että testaajat tuhlaavat aikaansa turhiin testeihin ja tietämättään ajavat samoja testejä useaan kertaan. [7]

Toistuvasti muuttuvien osien testaus: Automaattiset yksikkötestit osille, jotka muuttuvat toistuvasti, ovat riski tekijä, koska testien ylläpito kustannukset nousevat pilviin. [7][10] [11][5]

Koodaaminen testattavuus mielessä: Ohjelman koodia kirjoittaessa on hyvä pitää mielessä testattavuus. Koodia, jota ei ole suunniteltu testausta varten, esimerkiksi jos ohjelmaa ei ole jaettu osiin/moduuleihin, johtaa helposti siihen, että automaattisia testejä on hankala suunnitella, tehdä sekä ylläpitää. [7][1]

Tarvittavat resurssit: Automaattinen yksikkötestaus täytyy ottaa huomioon suunniteltaessa resurssien jakamista projektille. Yleisesti AUT-testit vaativat enemmän resursseja suunnittelu ja koodaus vaiheessa, kuin manuaalinen testaus.[7]

Konfiguraation hallinta ja virheiden seuranta : Automaattisen yksikkötestauksen osat kuuluvat konfiguraation hallinnan alaisuuteen, aivan kuin muutkin ohjelman osat. AUT-testit tulisi siten laittaa myös version hallinnan alaisuuteen, jos sellainen on projektissa käytössä. Sama koskee virheiden seurantaa, kaikki virheet mitkä löytyvät testikoodista, pitäisi kirjata ja seurata, samoin kuin normaalista koodista löydetyt virheet.[7]

Analyysi ja raportointi: Testi tulosten analyysi ja raportointi on aina ihmisten käsissä, sitä voidaan automatisoida jonkin verran, mutta se aina tarvitsee ihmisen tekemän interaktion vähintäänkin tulosten tarkastuksessa. Tuloksista olisi hyvä aina tehdä raportti ja säilöä se eri osapuolien tarkastusta varten. Vaikka tämä johtaisi turhien raporttien laadintaan, on tämä pieni ongelma, joka on helppo korjata seuraamalla mitä raportteja eri osapuolet seuraavat ja mitä eivät.[7]

3.3 Hyviä käytäntöjä riskien välttämiseksi

Tässä kappaleessa listaan muutamia käytäntöjä, joilla voi estää tai ainakin ennaltaehkäistä edellisessä luvussa mainittuja suden kuoppia ja muutenkin varmistaa automaattisen yksikkötestauksen onnistumisen. Käytännöt ovat Berner & al. tutkielmasta[10], jossa kirjoittajat seurasivat automaattisten testien implementaatiota viiteen projektiin.

3.3.1 Riskien tunnistaminen

Ei ole riittävää vain listata mahdollisia riskejä/sudenkuoppia riskilistaan, vaan näille on hyvä olla olemassa vaihtoehtosuunnitelmat. Vaikka kuinka valmistuisi näihin riskeihin, ei niitä aina voi välttää. Ulkopuoliset konsultit ja AUT spesialistit, jotka ovat kokeneet monet ongelmat ja epäonnistumiset, ovat kultaakin kalliimpia projekteissa, joissa yritetään ensimmäistä kertaa automaattista yksikkötestausta.[7]

3.3.2 Käytä kunnollista testaus strategiaa

Kunnollinen testaus strategia on elinehto menestyksekkäälle automaattiselle

yksikkötestaukselle. Tarkka strategia antaa mahdollisuuden arvioi kustannuksia etukäteen, jolloin resurssien määrittäminen helpottuu huomattavasti. Berner & al. antavat neljä askelta, joissa edetä testaus strategian kanssa. Nämä on tarkoitettu automaattiselle testaukselle, eivätkä tarkasti automaattiselle yksikkötestaukselle, mutta ovat silti tämän työn AUT määritelmän huomioon ottaen päteviä myös AUT-testaukseen.[10]

1.Määrittele testaus strategia, joka ottaa huomioon laatu tavoitteesi ja tarvitsemasi testauksen erityispiirteet. Yhdistä toimintoja, jotta pystyt parantamaan ja ylläpitämään testejä kokoajan.

2.Määrittele testi automaation tavoitteet. Esimerkiksi lyhyemmät julkaisu syklit, kustannussäästöt tai parempi tuotteen laatu, taikka mikä tahansa näiden yhdistelmä.

3.Älä käytä vain yhtä tapaa tehdä automaattista testausta. Strategiat, jotka käyttävät montaa eri lähestymistapaa, ovat yleensä tehokkaampia, kuin tiukasti yhdessä pitäytyminen.

4.Tee usein arviointeja, miten automaattinen testauksesi toimii. Käytä mittareita kertomaan, oletko saavuttanut asettamasi tavoitteet ja pyri kokoajan parantamaan testausta.

3.3.3 Suunnittele arkkitehtuuri testausta varten

Missä huonoa testistrategiaa voi parantaa ajan myötä, voi olla lähes mahdotonta muuttaa ohjelman arkkitehtuuria jälkeenpäin testaukseen sopivaksi. Ota huomioon ohjelman testattavuus jo heti, kun alat miettimään ohjelman arkkitehtuuria. Hyvä tapa tähän, on laittaa ei-toiminnallisiin vaatimuksiin asioita testaus strategiasta, kuten modulaarisuus ja mahdollisuus liittää mock-up:ja.[10]

3.3.4 Integroi päivittäin

Estääksesi automaattisten testien hajoamisen ajan myötä, pyri integroimaan ohjelmasi kerran päivässä ja samalla ajamaan kaikki automaattiset testit integraation aikana. Tämä yksinkertainen, mutta tehokas keino estää ongelmat, jotka nousevat siitä kun automaattisia testejä ei ajeta tarpeeksi usein. Ongelmat, joita automaattiset testit saavat selville, pitäisi korjata heti, riippumatta siitä ovatko ne vikoja testeissä vai ohjelmassa. [10]

3.3.5 Käytä hyvän ohjelmoinnin ja suunnittelun periaatteita

Automaattiset testi projektit ovat normaaleja ohjelmistokehitys projekteja siinä missä muutkin ja ovat alttiita samoille ongelmille. Jotta menestyisit, käytä samoja hyviä käytäntöjä ja tarkkaavaisuutta kuin muissakin projekteissa.[10]

Luku 4

4. Laatu automaattisessa yksikkötestauksessa

Luvussa kaksi, listasin automaattisen yksikkötestauksen hyötyjä ja luvun lopussa vihjasin, että näitä hyötyjä ei saavuta ilman kovaa työtä. Kolmannessa luvussa kerroin mitä pitäisi tehdä ja mitä pitäisi varoa, jotta mainitut hyödyt voisi saada realisoitua, samalla mainitsin useampaakin kertaan AUT-testejen laadukkaan tekemisen, kustannus tehokkuuden ja ylläpidon. Seuraavaksi syvennyn näihin asioihin tarkemmin.

Tässä luvussa käyn läpi asioita, jotka liittyvät läheisesti yksittäisen automaattisen yksikkötestin laatuun. Alussa, jotta lukija saa käsityksen minkälainen on automaattinen yksikkötesti, käyn lyhyesti läpi, miten tehdä perus automaattinen yksikkötesti. Tämän jälkeen käsittelen, mitä attribuutteja hyvällä AUT-testillä on, sekä minkälainen on kustannustehokas AUT-testi. Lopuksi syvennyn automaattisten yksikkötestien ylläpitoon.

4.1 Minkälainen on perus automaattinen yksikkötesti

AUT(Automatic unit testing) tapahtuu, kun on tuotettu testi, joka voidaan ajaa ilman että ihmisen tarvitsee tehdä muuta kuin käynnistää se ja tarkastaa lopputulos. Helpoiten tällaisen saa tuotettu siihen tarkoitetuilla työkaluilla. Ehkä tunnetuin yksikkötestaus työkalu on JUnit, xUnit perheen ”lippulaiva”, joka on tarkoitettu Java-ympäristöön. xUnit työkalujen periaatteena on kirjoittaa testiin sarja assertioita (esim. Assert.equal(1,1)), jotka testaavat, pitävätkö tietyt asiat paikkaansa. Tämän jälkeen ajaa testi erillisellä testi ohjelmalla joka edellä mainittujen assertioiden perusteella kertoo menikö testi läpi vai ei.

4.2 Minkälainen on hyvä AUT testi

Attribuutteja hyvällä automaattiselle testille löytyy paljon kirjallisuudesta, mutta näkemistäni Lanubilen ja Mallardon yhteenveto tutkimuksessaan[1] automaattisen testikoodin laadun tarkistamiseen on paras. Tutkimuksessaan he käyttävät sekä alan kirjallisuutta aineistona sekä tekevät oman empiirisen testin asiasta käyttämällä opiskelijoita testiryhmänä. Seuraavaksi kaksitoista heidän antamaansa attribuuttia hyvälle automaattiselle testille:

Lyhyt: Testin pitää olla lyhyt mutta ytimekäs.

Itsensä tarkistava: Testin pitää ilmoittaa testauksen tulos (hyväksytty/hylätty) ilman interaktiota ihmisen kanssa.

Toistuva: Testi pitää pystyä ajamaan monta kertaa ilman ihmisen interaktiota.

Robusti: Testin pitää aina tuottaa sama tulos.

Kattava: Testin pitää kattaa testattava asia tarpeeksi hyvin.

Tarpeellinen: Testi pitää rajoittaa testaamaan vain asioita, joita tarvitsee testata. Turhien asioiden testaaminen vain lisää sekaisuutta, esimerkiksi setterit ja getterit.

Selkeä: Testi pitää olla helposti ymmärrettävissä.

Tehokas: Testin ajaminen ei saa kestää liian kauaa. Tehokas testi takaa sen että kehittäjät ajavat sitä mahdollisimman usein. Mitä hitaampi testi on, sitä harvemmin kehittäjät sitä ajavat[6].

Tarkasti rajattu: Testin pitää testata jotakin tiettyä toiminnallisuutta tai aluetta.

Itsenäinen: Testin pitää antaa sama tulos ajettiinpa se sitten yksin tai muiden testien kanssa.

Ylläpidettävä: Testiä pitäisi olla helppo muokata ja jatko kehittää.

Jäljitettävissä: Testin pitää pystyä jäljittämään koodista tai vaatimuksista.

Lanubile ja Mallardo kirjoittavat myös miten yllä mainittujen attribuuttien huomiotta jättäminen näkyy. Näitä voidaan myös käyttää huono laatuisen testikoodin metsästykseen.

Sekavia testejä: Testeistä ei saa ensilukemisella selvää mitä ne testaavat.

Duplikaatti testit: Sama testi koodi esiintyy monessa testissä.

Testejä tuotteen koodissa: Itse tuotteen koodi sisältää testi koodia jonka pitäisi olla ennemmin testi tapauksissa.

Assertioruletti: Kun testi epäonnistuu ei tiedetä mikä assertioista on vastuussa epäonnistumisesta

Epäsäännöllinen testi: Testi antaa eri tuloksia riippuen koska sitä ajetaan ja kuka sitä ajaa.

Helposti särkyvä testi: Testi joka ei käänny tai epäonnistuu kun tehdään pienikin muutos koodiin.

Toistuva debuggaus: Joudutaan manuaalisesta debuggaamaan, jotta löydetään syy testit epäonnistumiselle.

Manuaalinen muokkaus: Testi vaatii manuaalista muokkausta ennen ajoa.

Hidas testi: Testin ajaminen kestää niin kauan että kehittäjät välttelevät sen ajamista vedoten ajan tuhlaukseen.

Samassa tutkimuksessa Lanubile ja Mallardo totesivat testiryhmässään yleisimmät ja vakavimmat ongelmat:

Yleisimmin esiintyvä ongelma oli tarve manuaalisille muutoksille testien ajamisessa. Suurin syy tähän oli se, että testit muokkasivat ohjelman tilaa, eivätkä palauttaneet sitä enää alkuperäiseen tilaan. Esimerkiksi testi lisäsi alussa tavaraa tietokantaan, mutta testin epäonnistuttua sitä ei poistettukaan, jolloin vaadittiin ihmisen väliintulo ohjelman saamiseksi alkuperäiseen tilaan. Tällaiset ongelmat vaikuttavat huomattavasti mahdollisuuteen ajaa testejä sujuvasti monta kertaa peräkkäin.[1]

Muita heidän kohtaamiaan ongelmia olivat ”Assertioruletti” ja ehtopohjainen testilogiikka(esim. Switch-lauseet), joissa yhdessä AUT-testissä testataan montaa luokan ominaisuutta yhtä aikaa monella assertiolla ja vertailulauseella. Tämä heikensi huomattavasti testin ylläpidettävyyttä ja selkeyttä. He myös löysivät jonkin verran duplikaatioita testeistä, mikä johtui monesti copy-paste oikaisuista testien pohjan luomisessa. Tämä lisää turhaan testi koodia ja aiheuttaa päänvaivaa testejä tutkittaessa.

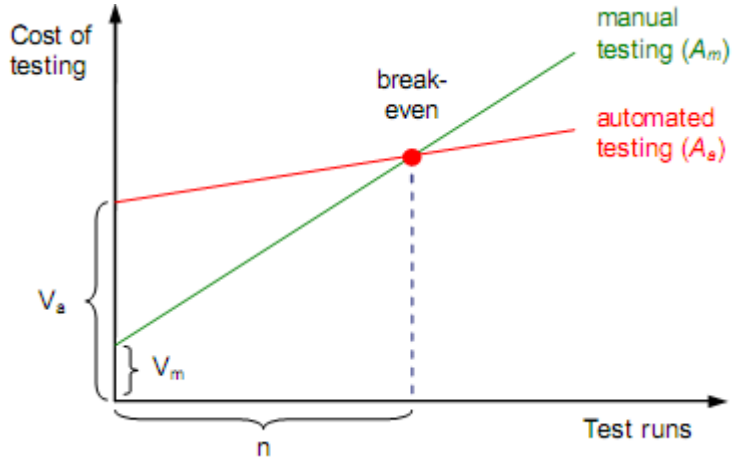
Lisäisin vielä kaksi asiaa, jotka on hyvä pitää mielessä:

1.Testien pitäisi luoda oma testiaineistonsa eikä luottaa olemassa olevaan dataan. Esimerkiksi tällä välttää sen, että kiinnostus testeihin ei romahda kun joku vetää kehitys kannan vahingossa sileäksi. [6]

2.Testikoodi on aitoa koodia. Sen tulee täyttää samat vaatimukset kuin normaalinkin koodin. Jotkut väittävät, että se on ehkä vielä aidompaa kuin julkaistava koodi. [4]

4.3 Kustannustehokas AUT-testi

Yleensä automaattisen testauksen kirjallisuudessa, testin kustannustehokkuudesta käytetään hyvin yksinkertaista kaavaa: E(n) = Aa / Am = (Va + n * Da)/ (Vm + n * Dm), jossa V = testin suunnittelun ja toteutuksen kustannus, D = testin yhden ajokerran kustannus ja n = testin odotettu ajomäärä. Jos tulokseksi saadaan yli 1, kannattaa testi automatisoida(Kuva 1.). Kaikissa ei mennä edes näin pitkälle vaan todetaan, että jos testin ajokerrat menevät jonkun tietyn luvun yli, se kannattaa automatisoida. Tämä luku liikkuu yleensä 2-20 välillä.[8]



Kuva 1. ([5] Figure 1)

Ramler ja Wolfmaier [8] kuitenkin toteavat , että tällainen lähestymistapa huomioi vain kustannukset eikä muita AUT-testin tuomia etuja, kuten näiden helppokäyttöisyyttä regressiotesteissä. He myös osoittavat neljä muuta asiaa miksi tämänlainen lähestymistapa ei ole riittävä:

1. AUT ja manuaalinen testaus eivät ole vertailukelpoisia, ne ovat ennemmin kaksi eri prosessia, kuin kaksi tapaa tehdä sama asia.
2. Lähestymistapa olettaa, että kaikki testitapaukset ovat yhtä arvokkaita, vaikka näin ei tosimaailmassa ole
3. Projektin kontekstia ei ajatella vaan päätös tehdään vain ajatellen yhtä testitapausta. Etenkin projektin testi budjettia ei huomioida ollenkaan
4. Iso osa kustannuksista sivuutetaan, kuten AUT-työkalujen kustannukset ja automaattisen testin ylläpitokustannukset lisäksi automaattinen testi saattaa olla viallinen.

Ramler ja Wolfmaier [8] esittelevät oman tapansa kustannustehokkuuden laskemiseen, tapa perustuu kiinteään testausbudjettiin. Tavassa arvioidaan kaikkia testitapauksia ja automaattisen testin arvo lasketaan menetettyjen manuaalisten testien perusteella. Suurin ero edellä mainittuun tapaan on se, että tämä ei yritä minimoida testien kustannuksia vaan maksimoida testauksen hyödyt suhteessa kiinteään budjettiin. Tässä tavassa automaattisen testin arvo lasketaan suoran kaavan sijasta käyttämällä kolmea rajoitetta ja yhtä maksimointifunktiota:

Rajoitteet:

R1: $na * Va + nm * Dm \leq B$. jossa na = testitapausten lukumäärä, nm = manuaalisten testien ajojen määrä, Va = testiautomaation kustannus, Dm = manuaalisen testin ajon kustannus, B = kiinteä budjetti

R2: $na \geq mina$, mina = minimimäärä automaattisia testitapauksia

R3: $nm \geq minm$, minm = minimi määrä manuaalisia testiajoja

Maksimointi:

$$Ra(na) + Rm(nm) \rightarrow Max$$

jossa Ra ja Rm ovat riskin vähennys funktioita tietyllä määrällä testejä.

Ramler ja Wolfmaier [8] toteavat itsekin, että tämä ei ole täysin kattava malli ja jättää joitain ominaisuuksia huomioimatta, mutta heidän mukaansa loppujen tai edes muutaman asian mukaan ottaminen tekisi laskemisesta liian monimutkaista, jolloin jo pelkän laskennan kustannukset alkaisivat vaikuttaa kustannustehokkuuteen. He kuitenkin listaavat kuusi asiaa, jotka pitäisi miettiä yllä mainitun kaavan lisäksi, kun lasketaan automaattisen yksikkötestin kustannustehokkuutta:

1. On testejä, joita ei voi järkevästi toteuttaa manuaalisina, kuten suorituskykytestit. Toisaalta on testejä, joita ei voi taas järkevästi suorittaa

automaattisella testauksella.

2. Testien suunnittelun tehokkuus muuttuu ajan myötä. Tämä pitää paikkansa varsinkin projekteissa, joissa otetaan ensimmäistä kertaa automaattinen yksikkötestaus käyttöön. Alussa testin suunnitteluun ja tekoon voi mennä monta tuntia, mutta kun työkalut ja tavat alkavat tulla tutummiksi, tämä aika saattaa tippua jopa puoleen.
3. Nouseva testauksen määrä iteratiivisissa projekteissa. Iteratiivisissa projekteissa alussa yleensä tehdään vain runko, jonka jälkeen aletaan toteuttamaan toiminnallisuutta tärkeysjärjestyksessä. Tästä johtuen testauksen pääpaino yleensä siirtyy projektin loppupuolelle, jossa automaattisista testeistä ei enää saada niin paljoa irti kuin manuaalisista.
4. Automaattisia testejä täytyy ylläpitää. Kun ohjelma muuttuu, samalla moni testi saattaa lopettaa toimimasta, jolloin se on korjattava/suunniteltava uudelleen. Tämä voi tuoda huomattavia lisäkustannuksia.
5. Aikainen/myöhäinen sijoituksen palautus. Useimmissa tapauksissa automaattisilla testeillä kestää kauemmin saada hyöty kuin manuaalisella testauksella.
6. Jokainen testi ei ole saman arvoinen. Kaikkien testien tehokkuus ei ole sama: toinen saattaa olla huomattavasti tehokkaampi löytämään virheitä kuin toinen. Niin kuin yleensä todetaan, 20 % ohjelmasta luo 80 % virheistä.

4.4 Automattisten yksikkötestien ylläpito

Nopeiden muutosten kanssa pärjääminen on yksi syistä, miksi automaattista testausta käytetään, mutta tällä samaisella syyllä on myös suuri haitta automaattiselle testaukselle, nimittäin testien ylläpidon kannalta. On hyvin tärkeää muistaa, että automaattista yksikkötestiä ei voi vain tehdä ja unohtaa, niitä pitää myös ylläpitää.

Tämän luvun olen jakanut kahteen osaan: ensimmäisessä käyn läpi käytännön tutkimuksen siitä, kuinka paljon pienikin muutos ohjelman koodiin saattaa vaatia muutoksia testikoodiin. Toisessa osiossa käsittelen huomioita automaattisten yksikkötestien ylläpidosta.

4.4.1 Käytännön tutkimus AUT-testien ylläpitoon

Käytännön tutkimuksen asiasta ovat tehneet Mats Skoglund ja Per Runeson[5]. Kyseisessä tutkimuksessa tutkijat ottivat vanhan järjestelmän, jolle oli olemassa automaattisia yksikkötestejä ja tekivät tälle ohjelmalle lisäosan kolmella eri tavalla. Mielestäni tämä tutkimus on erittäin osuva kertomus regressiotestauksen, joka on tehty automaattisella yksikkötestauksella, ylläpidon määrästä, kun tehdään pieniäkin muutoksia koodiin. Seuraavaksi käyn läpi tarkemmin tätä tutkimusta ja sen paljastamia tuloksia.

Niin kuin mainitsin yllä ohjelmalle oli olemassa automaattisia yksikkötestejä,

tarkalleen ottaen 385 kappaletta, jotka sisälsivät yhteensä 1369 assertiota. Työkaluna näiden testien tekemiseen oli käytetty JUnit:tia. Ohjelmaa, joka koostui 19 lähdekoodi tiedostosta, jatko kehitettiin kolmella eri tavalla: Ensimmäinen oli nimeltään ”*Change propagation*”, tässä kehittäjät muuttivat yhtä olemassa olevaa luokkaa ja tämän jälkeen kaikkia luokkia, jotka olivat riippuvaisia tästä luokasta. Toinen tapa oli ”*Refactoring*”, siinä kehittäjät ottivat kaikki luokat, joihin muutos tulisi vaikuttamaan ja korvasivat nämä vähemmällä määrällä uusia luokkia. Kolmannessa tavassa ”*Role splitting*”, vanhaa koodia ei muokattu ollenkaan, vaan kaikki muutokset/lisäykset tehtiin uusina funktioina ja luokkina.

Kun muutokset oli tehty laskettiin muutettujen koodirivien määrä, jonka jälkeen yritettiin testata vanhoilla yksikkötesteillä. Prosessi testauksessa toimi siten, että aluksi pyrittiin kääntämään ohjelma testien kanssa, jos tämä ei toiminut testejä muokattiin kunnes testit kääntyivät. Kääntämisen jälkeen testejä yritettiin ajaa, jos testit eivät toimineet yksikkötestejä muokattiin. Kun lopulta kaikki testit kaikilla tavoilla oli tehty, olivat löydökset seuraavan laisia:

Change propagation. Tavassa muutettiin 91 (0.5%) riviä vanhaa koodia. Testien kääntäminen ei onnistunut ensimmäisellä kerralla, tuloksena oli käännettäessä 246 virhettä. Kun testit saatiin kääntymään, niin ensimmäisellä ajo kerralla saatiin 10 virhettä ja 110 testiä epäonnistui. Lopulta, kun testit saatiin menemään läpi, oli testeihin tehty muutoksia 124 (2.5%) testi koodi riviin.

Refactoring. Tässä tavassa loppujen lopuksi muutettiin 95 riviä vanhaa koodia, tämä on lähes sama määrä kuin change propagationisa. Koska refactorningissa, koodin luokka rakenteet muuttuivat huomattavasti eivät vanhat testit tietenkään edes kääntyneet. Tässä kohtaa käsiteltävää tutkimusta on mielestäni hassu reikä. Tutkijat eivät yritä muokata testejä, jotta ne kääntyisivät refactoringilla tehtyyn koodiin vaan toteavat, että edelliset Change propagationin testit toimivat myös tällä tavalle. Tuloksiksi he kirjaavat samat kuin change propagationillekkin.

Role Splitting. Tässä muutoksia tehtiin kaikkein vähiten, 87 riviin vanhaa koodia. Tämä johtui tietenkin siitä, että pyrittiin muokkaamisen sijaan luomaan uusia funktioita ja luokkia. Tästä samaisesta syystä, myös testit kääntyivät ja menivät läpi ensimmäisellä yrittämällä.

Tutkimuksen yhteenvedossa tutkijat toteavat, että *role splitting* oli kaikkein kustannus tehokkain testien ylläpidon kanalta. Toisaalta he myös kirjoittavat, että minkään näköistä tutkimusta ei tehty ennen tai jälkeen testien muokkausta, niiden laadulle tai kattavuudelle. Tutkijat epäilevät, että varsinkin role splittingin tapauksessa kattavuus olisi ollut huomattavasti muita heikompi. Myös muissa laatu olisi kyseenalaista, koska heidän tavoitteensa oli saada vain testit menemään läpi, ei tehdä niistä laadukkaita. He myös toteavat, että testien laadun tutkiminen olisi erittäin kiinnostava jatko tutkimuksen kohde.

Yhtenä lisähavaintona he listaavat sen, että heidän mielestään muutosten tekijän olisi hyvä tehdä testien uudelleen muokkaus tai olla ainakin siinä mukana. Syyksi he toteavat sen, että tarvitaan hyvä tuntemus muuttuneista implementoinneista, jotta pystytään

muokkaamaan testejä siten, että ne testaavat samoja asioita uudessakin implementaatiossa. [5]

4.4.2 Huomioitavaa ylläpidossa

Edellisen kappaleen tutkimuksesta voidaan huomata, että pienetkin muutokset vanhaan koodiin voivat vaikuttaa monen testitapauksen toimivuuteen. Samoin näiden testitapausten takaisin toimintaan saaminen ei ole triviaali asia, varsinkin kun halutaan säilyttää testien korkea laatu ja laaja kattavuus. Tutkimus antaa myös perusteet sille miksi regressio/AUT testejä tulisi ajaa mahdollisimman usein jotta ei tule vastaan hetkeä jolloin iso prosentti testeistä ei toimi. Tähän hyvä lääke mielestäni on sellainen AUT-järjestelmä joka mahdollistaa testien ajamisen nopeasti ja kätevästi.

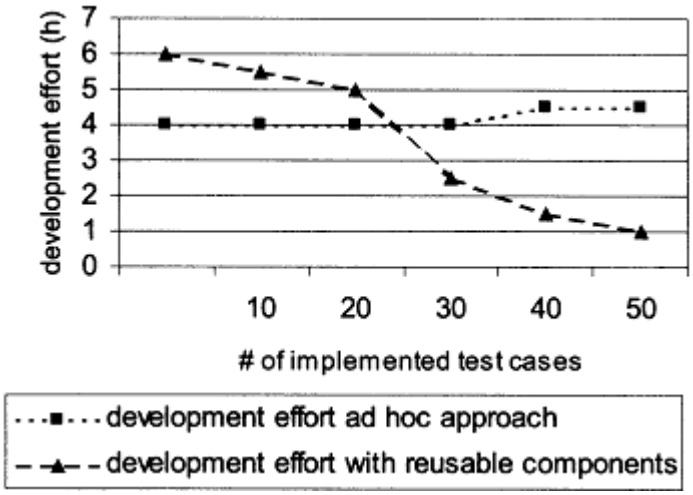
Brewer & al.[8] huomasivat saman myös omassa tutkimuksessaan. He seurasivat viittä projektia, joissa implementoitiin automaattinen testaus. He antavat neljä asiaa, jotka heidän mielestään johtavat ylläpidettävyyden heikentymiseen ja samalla joko automaattisen testauksen kaatumiseen tai vähintäänkin kustannusten jyrkkään kasvuun.

Dokumentoimaton testien arkkitehtuuri. Testien tekemiseen ei käytetä samaa tarkkaavaisuutta ja hyviä käytäntöjä kuin ”oikeaan” koodiin. Yrityksen koodaus ohjeet ja käytännöt monesti sivuutetaan kun tehdään testejä. Tärkeitä testien arkkitehtuurisia päätöksiä tehdään ad-hoc mentaliteetilla. Tämä kaikki johtaa alkuperäisen testi arkkitehtuurin nopeaan rapistumiseen. [10]

Huonosti rakennetut testit. Testit pitää olla rakenteeltaan yksinkertaisia ja hyvin dokumentoituja, jotta kehittäjät tehdessään testejä pystyvät helposti uudelleen käyttämään vanhoja. Jos näin ei ole kehittäjät keksivät pyörän joka kerta uudestaan, jolloin päädytään moniin duplikaatteihin testeissä sekä ongelmiin muutettaessa testi aineistoa.[10]

Testaamattomat testit. Myös automaattisia testejä pitää testata, joskus varsinkin jos testejä suunnitellaan uudelleen käyttömielessä, niistä saattaa tulla hyvinkin monimutkaisia. Tällöin on hyvä varmistua testien oikeellisuudesta vähintään siten, että ajaa testin tilanteessa jolloin sen pitäisi epäonnistua.[10]

Uudelleen käyttö mahdollisuudet ohitetaan. Testit ovat monesti toistavia. Samat interaktiot testi objektin kanssa toistetaan uudestaan ja uudestaan. Naiivilla suhtautumisella automaattiseen testaukseen, nämä toistuvat toiminnot kirjoitetaan moneen kertaan, johtaen korkeisiin kustannuksiin, kun koodia muokataan. Testi komponentteja ei pidä myöskään unohtaa kun projekti vaihtuu, monesti aikaisemmin suunnitellut yleiset testi työkalut, ajurit ja simulaattorit ovat käyttökelpoisia muissakin projekteissa. Kuva 2 esittää vaadittavan työnmäärän suhteessa testi tapausten määrään käytettäessä ad-hoc ja uudelleen käyttämisen lähestymistapaa. [10]



Kuva 2 [10] Figure 4

Brewer & al. [10] kirjoittavat yhteenvedossaan, että heidän tutkimukseensa osallistuneista projekteista automaattisten testien ylläpidosta tuli kaikille iso taakka ja syyt olivat pääasiallisesti neljä edellä mainittua.

Ylläpidettävyys on tärkeä asia AUT-testien laadun kannalta, mutta se on myös erittäin tärkeä koko automaattisen testaus prosessin kannalta, koska ilman tehokasta automaattisten testien ylläpidon suunnittelua ja toteutusta on erittäin todennäköistä, että koko automaattinen testaus epäonnistuu. Miksi painotin asiaa näin paljon työssäni on se, että ihmiset jotka ovat tottuneet manuaaliseen testaukseen, vieroksuvat ajatusta testien ylläpidosta, jonka takia se monesti jää joko heikosti toteutetuksi tai kokonaan huomioon ottamatta.

Luku 5

5. Johtopäätökset ja yhteenveto

Automaattiseen yksikkötestaukseen ei voi lähteä vain suin päin, tämä on varma resepti epäonnistumiselle. Niin kuin toisen luvun lopussa totesin ja mikä myöhemmissä luvuissa käy selväksi on se, että automaattisen yksikkötestauksen tekeminen vaatii työtä. Tietenkään pelkkä kova työ ei riitä, se on vasta hyvä lähtökohta, asiat pitävät olla muutenkin kohdallaan: Tarvitaan kypsäorganisaatio, joka osaa käsitellä prosessin muutoksen ja kehityksen, täytyy olla osaavat kehittäjät ja testaajat varmistamaan, että testit ovat laadukkaita, täytyy olla valmis pistämään resursseja AUT-testaukseen ja tietenkin pitää ymmärtää sen antama arvo.

Kaikkien edellä mainittujen asioiden jälkeen ei voi olla ajattelematta, että onko automaattinen yksikkötestaus sen arvoista? Tähän vastaisin kyllä, koska nykypäivän asiakas osaa odottaa jo laatua ohjelmaltaan, mutta myös nykypäivän ihminen vaatii sitä heti eikä huomenna ja AUT-testaus antaa tähän hyvät valmiudet. Automaattinen yksikkötestaus ei ole kuitenkaan kaikille. Esimerkiksi jos et tee alunperinkään yksikkötestaus, siirtyminen tyhjästä tekemään automaattista yksikkötestaus on hyvin vaikeaa. Samoin se sopii paremmin iteratiiviseen tuotanto tapaan, kuin puhtaaseen vesiputousmalliin, koska suurin osa sen hyödyistä saadaan vasta n+1 julkaisusta ja sen jälkeisistä. Eli toisin sanoen, jos olet aloittamassa lyhyttä projektia, josta tullaan tekemään vain yksi julkaisu ilman ylläpito sopimusta, automaattinen yksikkötestaus ei ole välttämättä sinua varten. Mutta jos olet aloittamassa vuosia kestävän projektin(tai olet keskellä sellaista), jossa tehdään lukemattomia julkaisuja, sekä sisältää pitkän ylläpito sopimuksen, sinun kannattaa ehdottomasti harkita automaattista yksikkötestausta.

Tässä tutkielmassa yritin parhaani mukaan löytää oikeanelämän esimerkkejä automaattisesta yksikkötestauksesta, mutta tyhjästä on paha nyhjäistä. Automaattista testausta on tutkittu hyvin paljon ja siitä löytyy monenlaista artikkelia ja kirjaa, mutta ne käsittelevät automaattista testausta laajalla alalla, ne hyvin harvoin menevät automaattiseen yksikkötestaukseen. Edellä mainitusta syystä suurin osa tämän tutkielman tekemiseen menneestä ajasta, kului lukiessa artikkeleita automaattisesta

testauksesta ja yrittäen löytää niistä asioita, jotka soveltuvat automaattiseen yksikkötestaukseen. Yksi asia missä lopulta otin kevyen oikopolun, oli määritelmäni AUT-testille, joka on hyvin laaja. Jotkut käyttävät samaa määritelmää, mutta useammin se määritellään huomattavasti tiukemmin koskemaan vain puhtaasti kehittäjien tekemiä testejä, jotka testaavat heidän kirjoittamiaan funktioita yksikkötasolla.

Niin kuin aikaisemmin totesin, jatkotutkimus automaattisen yksikkötestauksen hyödyistä voisi olla hyvin mielenkiintoista. Kritiikin aiheita ovat esimerkiksi automaattisen yksikkötestauksen tuomat lisätyötunnit, koska yleensähän testit löytävät suurimman osan virheistä kun niitä tehdään. Onko oikeasti sen vaivan arvoista, mikä joudutaan näkemään jos nämä testit yritetään säilyttää ja ylläpitää, jos kerran ne ovat löytäneet jo suurimman osan virheistään, mitä ne tulevat löytämään? Tietenkin automaattisella yksikkötestauksella on muitakin hyötyjä ja niitä ei voi erillään tarkastella vaan pitää laskea näiden kaikkien summa, mutta niin kuin sanoi tämä olisi mielenkiintoinen jatkotutkimus.

Viitteet

[1] Filippo Lanubile and Teresa Mallardo. Inspecting Automated Test Code: A Preliminary Study. *Agile Processes in Software Engineering and Extreme Programming. 8th International Conference*, 2007. ISBN 3-540-73100-8

[2] Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. ECOOP 2002 — Object-Oriented Programming. Sivut 1789-1901. 2002. ISSN 0302-9743 (Print) 1611-3349 (Online). DOI 10.1007/3-540-47993-7_10

[3] Michae Ellims James Bridges and Darrel C.Ince. The Economics of Unit Testing. Journal of Empirical Software Engineering. [Verkkolehti] Volume 11, Number 1 / March, 2006. ISSN 1573-7616. DOI 10.1007/s10664-006-5964-9.

[4] Andy Hunt ja Dave Thomas. Pragmatic Unit Testing in C# with NUnit: The Pragmatic Starter Kit, Volume II (Volume Set). 2004. ISBN 9780974514024

[5] Mats Skoglund ja Per Runeson. A Case Study on Regression Test Suite Maintenance in System Evolution. Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04) sivut 438-442. 2004. ISBN 0-7695-2213-0

[6] Peter Schuh. Integrating Agile Development in the Real World. 2005. ISBN 1584503645

[7] Christer Persson, Nur Yilmaztürk. Establishment of Automated Regression Testing at ABB: Industrial Experience Report on 'Avoiding the Pitfalls'. Automated Software Engineering Proceedings of the 19th IEEE international conference on Automated software engineering Pages: 112 – 121. 2004. ISBN 1068-3062 , ISSN 0-7695-2131-2

[8] Ramler, R. and Wolfmaier, K. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international Workshop on Automation of Software Test* AST '06. ACM, New York, NY, 85-91. 2006. ISBN 1-59593-408-1

[9] Cem Kaner. Pitfalls and Strategies in Automated Testing. Computer [Verkkolehti] Volume 30 Issue 4 Sivut 114-116. 1997. DOI 10.1109/2.585164. ISSN 0018-9162

[10] Berner, S., Weber, R., ja Keller, R. K.. Observations and lessons learned from automated testing. In Proceedings of the 27th international Conference on Software Engineering ICSE '05. ACM, New York, NY, 571-579. 2005. ISBN 1-59593-963-2

[11] Marcos Kalinowski Hugo Vidal Teixeira Paul Johan Heinrich v'an Oppen . ABAT: An Approach for Building Maintainable Automated Functional Software Tests. 2007. DOI 10.1109/SCCC.2007.10. ISBN 978-0-7695-3017-8.