Wipron Suomen toimipisteen ohjelmistotestauksen kehittäminen

Marko Isoaho

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Marko Helenius
Toukokuu 2007

Tampereen Yliopisto

Tietojenkäsittelytieteiden laitos

Marko Isoaho: Wipron Suomen toimipisteen ohjelmistotestauksen kehittäminen

Pro gradu -tutkielma, 51 sivua, 5 liitesivua

Toukokuu 2007

Avainsanat: testaus, testausmenetelmät, testauksen työkalut, testauksen automatisointi, testauksen kehittäminen, tehokas testaaminen, testitapaus

Testauksella voidaan ohjelmistotuotteesta löytää virheitä ja näin parantaa ohjelmiston laatua. Ohjelmiston kehitysprosessissa virheiden poistaminen on merkittävä osa koko prosessia, sillä ohjelmistotuotteessa oleva virhe on voinut tulla sattumanvaraisesti ohjelmiston luonnin aikana. Ohjelmiston laatu paranee joka kerta, kun on löydetty uusi vika ja korjattu se. Testausta suoritetaan eri tavoin kaikissa ohjelmistontuotantoprosessin vaiheissa ja testauksen kohde vaihtelee prosessin vaiheiden mukaan.

Tässä pro gradu -tutkielmassa tullaan selvittämään ja perehtymään ohjelmistotestausta parantaviin menetelmiin ja käyttökohteisiin. Keskeinen kysymys oli, kuinka voitaisiin kehittää ja automatisoida Wipro:n Suomen toimipisteen ohjelmistotestausta, jolloin saadaan ohjelmistotuotanto aikaisempaa tuottavammaksi ja tehokkaammaksi. Tutkimuksen kohteena olivat eri ohjelmistotestausmenetelmät ja tavoitteena oli luoda uusi testauksen kehityssuunnitelma. Tutkimuksen teoriaosuudessa esitellään testauksen eri vaiheita, testauksen käsitteistöä ja eri menetelmiä sekä testauksen automatisointia.

Teoriaosuuden ja testauksessa ilmenneiden puutteiden avulla tein testauksen kehityssuunnitelman, joka sisälsi uuden työkalun käyttöönoton, jolla voidaan nopeuttaa rutiinien tekemistä ja itse testausta. Lisäksi tein muitakin parannuksia testaukseen ja siihen liittyviin prosesseihin. Lopuksi vertailin kehityssuunnitelman tuloksia vanhoihin testauksen menetelmiin. Parannuksien jälkeen testaus nopeutui erittäin paljon ja kaikki testaukseen tarvittavat välineet ja tulokset löytyivät helposti. Tulevaisuudessa testaus nopeutuu ja turhat rutiinit vähenevät testauksessa.

Osoitin tutkimuksella, mitä puutteita Wipron testausprosessissa on ja kuinka testausta voidaan pienilläkin parannuksilla parantaa ja kehittää. Tämän tutkimuksen tulokset voivat auttaa myös muita organisaatioita tehostamaan testausta sekä tunnistamaan mahdollisia ongelmia ja kehityskohteita siinä.

Tampere University
Information science institution
Marko Isoaho: The software testing enhancement of Wipro finland’s office
Master’s Thesis, 51 pages, 5 appendix pages
May 2007

Keywords: testing, testing methods, testing tools, test automation, effective testing, testing enhancement, test case.

By using software product testing faults can be found and the quality of a software product is enhanced. Removing faults during the software development process is very important phase of the entire software manufacturing process, because the faults can appear random. Quality of software product will be enhanced every time, when a new fault is found and corrected. Testing need to be executed variously in all the phases of software manufacturing process. The focus of testing need to be changed corresponding to the manufacturing process phase.

The main goal of this thesis is to chase up and make conversant efficient methods and targets of software testing. More specifically, how can software testing be enhanced and automated at the Wipro finland’s office? As a consequence software production got more productive and more efficient. I will narrow down to different methods of software testing. The goal is to create a new testing evolution plan. Theory part contains phases of testing, testing concepts, different methods and testing automation.

I made an evolution plan for testing with theory and the found deficiencies of testing. This plan contains a new tool for testing. It enables faster testing routines itself. Evolution plan contains also improvements for testing and its processes. Finally, evolution plan’s results were compared with old methods. Testing was faster after changes and all testing instruments and results were found easily. In the future, testing will be faster and the needless routines of testing will be become less.

I attested with this research, what are the flaws in Wipro’s testing process and how can testing be enhanced with the small improvements. This finding can be benefit to other organizations with enhancing testing, identifying the problems of testing and the objects of improvement.

SISÄLLYS

1. Johdanto **1**
1.1. Tutkimuskysymys 2
1.2. Tutkimusmetodi 2
1.3. Tutkielman rakenne 3

2. Ohjelmiston testaus **5**
2.1. Yleistä testauksesta 5
2.2. Testauksen määrittely 5
2.3. Testauksen tarkoitus 6
2.4. Testauksen suunnittelu 7
2.5. Testauksen vaiheiden jako 8
2.5.1. Moduuli- eli yksikkötestaus 10
2.5.2. Integrointitestaus 11
2.5.3. Systeemi- eli järjestelmätestaus 11
2.5.4. Hyväksymistestaus 12
2.5.5. Regressiotestaus 12
2.5.6. Käytettävyystestaus 13
2.6. Testausmenetelmät 13
2.6.1. Mustalaatikkotestaus 14
2.6.2. Lasilaatikkotestaus 14
2.6.3. Harmaalaatikkotestaus 15
2.6.4. Katselmoinnit 15
2.7. Testauksen apuvälineet 16
2.7.1. Analysointi/tarkastusapuvälineet 18
2.7.2. Testauksen suunnittelun ja hallinnan apuvälineet 18
2.7.3. Dynaamiset työkalut testauksessa 19
2.7.4. Automatisointityövälineet testauksessa (GUI-ajurit) 19
2.7.5. Kuormitus- ja suorituskykytyökalut 20
2.7.6. Staattisen analyysin työkalut 21
2.8. Työvälineen hankinta 21
2.9. Testauksen tulosten ja logien käsittely 22
2.10. Testausympäristön ylläpidettävyys 23

3. Testauksen automatisointi.. 25
3.1. Testauksen automatisoinnin hyötyjä.. 25
3.2. Automatisoinnin kompastuskiviä... 28

4. Ohjelmistotestausprosessin kehittäminen... 30
4.1. Lähtökohdat kehittämiselle.. 30
4.2. Käytännön toimenpiteet kehitettäessä testausta.. 30

5. Wipron ohjelmistotestauksen kehittäminen ja automatisointi................................. 32
5.1 Toimintatavat.. 32
5.2 Rational Visual Tester... 32

6. Ohjelmistotestauksen kehityssuunnitelma.. 34
6.1. Rational Visual Test -työväline käyttöönotto... 34
6.2. Testien tulosten tallentaminen.. 35
6.3. Testitapausten kirjoittaminen ja katselmoinnit.. 37

7. Mittaukset ja arvioinnit.. 38
7.1. Työvälineen käyttö.. 38
7.2 Testausten tulokset ja lokit... 39
7.3. Testitapausten kirjoittaminen.. 41
7.4. Esikatselmointi... 43

8. Yhteenveto... 45

Viiteluettelo... 47
Liitteet

1. Johdanto

Tutkielmassa tullaan selvittämään ja perehtymään ohjelmistotestausta parantaviin menetelmiin ja käyttökohteisiin, eli kuinka voitaisiin kehittää ja automatisoida Wipro:n Suomen toimipisteen ohjelmistotestausta. Wipron Suomen toimipiste on keskittynyt tietoliikenneohjelmistojen tuotekehitykseen ja tuotekehityksen alihankintatöihin. Toimipisteessä hyödynnetään eri ohjelmistotestausmenetelmiä, kuten moduulitestausta, integrointitestausta, ominaisuustestestausta, toiminnallisuustestausta ja systeemitestausta.

Ohjelmistoyritykset joutuvat nykyään panostamaan testaukseen ohjelmistomarkkinoiden kiristyessä. Laadunvarmistus ja kustannustehokkuus
ovat ohjelmistomarkkinoiden kannalta tärkeitä tekijöitä. Oikeilla välineillä ja menetelmillä voidaan testausprosessin laatua ja käytettyä työpanosta vähentää huolimatta ohjelmiston koon kasvaessa ja sen toimintojen monipuolistuessa.

Testauksella voidaan ohjelmistotuotteesta löytää virheitä ja siten parantaa ohjelmiston laatua. Ohjelmiston kehitysprosessissa virheiden poistaminen on merkittävä osa koko prosessia, sillä ohjelmistotuotteessa oleva virhe ei ole koskaan sattumanvarainen. Ohjelmiston laatu paranee joka kerta, kun on löydetty uusi vika ja korjattu se.

Testaaminen on monimutkainen prosessi varsinkin, jos ohjelmisto on osa suurempaa järjestelmää. Testaus nähdäänkin nykyään tärkeäksi tekijäksi ohjelmistotuotannossa, joka jatkuu läpi koko ohjelmistotuotuotteen elinkaaren.
Testaus ei ole enää vain ennen käyttöönottoa tehtävä ohjelmiston toimintovaihe, vaan erittäin tärkeä toiminto ohjelmiston suunnittelussa ja sen kehityksessä.

Testitapausten suunnittelu ja toteutus vievät paljon aikaa. Virheiden löytämiseksi ohjelmakoodista joudutaan testauksessa suorittamaan samoja testitapauksia toistuvasti. Testien toistettavuus synnyttääkin haasteita testauksen kehittämiseen, sen tehostamiselle ja nopeuttamiselle. Tähän voidaan löytää ratkaisu esimerkiksi testauksen automatisoinnilla. Varsinkin usein toistettavia toimintoja testauksessa kannattaisi automatisoida. Jokainen tehty muutos tai lisäys ohjelmakoodiin lisää testauksen määrää samoilla testitapauksilla.

Tässä tutkimuksessa olen rajannut testauksen kehittämisen sisältämään ominaisuus-, toiminnallisuus-, ja järjestelmätason testauksen, koska teen näitä testauksia työpaikallani Wiprossa. Nämä eri testaukset ovat tulleet minulle tutuiksi töiden kautta, joten näitä testauksia on helpompi kehittää ja parantaa. Tutkimusmotiivi on testauksen nopeuttaminen ja tiettyjen testausrutiinien automatisointi ja mahdollisten makrojen teko, joilla voidaan nopeuttaa testausta.

Tässä tutkimuksessa esitetyt ehdotelmat kehitysideoiksi ovat omiani eikä Wipro vastaa kyseisen näkemyksen asiasisällöstä, todenperäisyydestä tai totuudenmukaisuudesta.

1.1. Tutkimuskysymys

Tutkimuksella avulla pyrin kehittämään ja parantamaan ohjelmistotestausta, jolloin ohjelmistotuotanto ja testaus saadaan entistä tuottavammaksia ja tehokkaammaksi. Tärkeimmät muutokset, jotka näillä keinoilla halutaan saada aikaan, ovat testauksen ja testausrutiinien sekä testausympäristön konfiguroimisen nopeuttaminen; esimerkiksi muuttamalla testitapausten kirjoittamista, jolloin jää aikaa enemmän ongelmatilanteiden miettimiseen. Testitapausten teko ja testaaminen voivat nopeutua jopa viikkoja. Tutkimuksessa testausmenetelmämuutokset toteutetaan ja niiden käyttökelpoisuus varmistetaan. Tutkimuksen valittömät tulokset ovat tällöin (1) ohjelmistotestaussuunnitelma, joka sisältää tehtävät muutokset testauksessa ja testausmenetelmissä (2) mittaukset ja arviot, joilla todennetaan testauksen nopeutuminen testauksen eri kehityskohteissa.

Varsinaista tilaajaa tutkimukselle ei ole, mutta ohjelmistotestauksen kehittämistä tarvitaan Wiprossa, koska tietyt ohjelmistotestausrutiinit ja perustestaukset pitäisi automatisoida.

1.2.Tutkimusmetodi

Tutkimusmenetelmä on konstruktiivinen eli tutkimuksessa pyritään luomaan uusia ohjelmistotestausmenetelmiä ja automatisoidaan joitain ohjelmistotestausrutiineja. Käytannössä luodaan ohjelmistotestauksen automatisointi- ja kehityssuunnitelma, jota mahdollisesti tullaan käyttämään apuna tulevaisuuden testauksessa. Tutkimuksen lähestymistapa on käytännönläheinen deduktiivinen päättely. Deduktiivista päättelyä sovelletaan siten, että käydään läpi testausasioita ja teoriaa sekä itse testausmenetelmiä. Näistä tehdään loogisia johtopäätöksiä ja niiden avulla muodostetaan testauksen kehityssuunnitelma.

1.3. Tutkielman rakenne

Tämä tutkielma tulee sisältää kahdeksan eri lukua. Työssä annetaan kokonaiskuva ohjelmiston testauksesta ja testauksen automatisoinnista. Lisäksi selvitetään Wipron Suomen toimipisteen testauksen kehittämisistä ja muutoksia sekä niiden toteutus.

Toisessa luvussa selvitetään ohjelmiston testausta. Luvussa annetaan yleiskuva testauksen tarkoituksesta, esitetään testauksen määritelmä, kuvataan testauksen eri vaiheita sekä erilaisista testausmenetelmiä, joilla testausta suoritetaan ja kuvataan testauksen toteutusta ohjelmistotuotannon eri tasoilla. Luvussa esitellään myös mahdolliset testauksen apuvälineet ja niiden sijoittuminen ohjelmistotuotannon eri vaiheisiin.

Luvussa kolme selvitetään testauksen automatisoinnin prosessia. Luvussa esitetään automatisoinnille saavutettavia hyötyjä ja toisaalta myös automatisointiin liittyviä ongelmia. Luvussa selvitetään myös, mitä testauksen automatisointi tarkoittaa.

Neljännessä luvussa käsitellään ohjelmistotestausprosessin kehittämistä yleisesti. Luvusssa esitetään lähtökohtia yrityksen testauksen kehittämiseen ja käytännön toimenpiteitä testauksen kehittämiselle.

Luvussa viisi käydään läpi Wipron mahdollista ohjelmistotestauksen muuttamista ja sen automatisointia. Luku sisältää mahdollisen työkalun hankinnan yritykseen, jolla voidaan nopeuttaa testauksen rutiinitehtäviä. Luvussa käydään läpi testauksen toimintatapoja, joilla voidaan parantaa Wipron ohjelmistotestausta.

Kuudennessa luvussa tehdään testauksen kehityssuunnitelma ohjelmistotestausta varten, jolla voidaan nopeuttaa ja tehostaa testausta. Luvussa käsitellään Rational Visual Test –työvälineen käyttöönottoa testauksessa. Luku kuusi sisältää myös kehitysideoita testitapausten kirjoittamiseen, testitapausten tulosten tallentamiseen ja katselmointien kehittämiseen testauksessa.

Seitsemännessä luvussa testataan kehityssuunnitelman pohjalta uudet menetelmät ja käyttöönotettu työväline. Testauksen suorittaa kaksi koetestaaja ja tutkimuksen tekijä. Kehityssuunnitelman testauksen jälkeen verrataan testauksien tuloksia vanhojen menetelmien tuloksiin ja arviodaan kehityssuunnitelman hyvyys vanhoihin menetelmiin.

Luku kahdeksan sisältää yhteenvedon tutkimuksesta ja mielipiteitä ohjelmistotestauksen kehittämisen onnistumisesta Wiprossa ja mahdollisen jatkokehityssuunnitelman. Luvussa käydään läpi yleisesti testauksen automatisoinnin hyötyjä testauksessa ja testauksen tärkeyden ohjelmistotuotannossa.

2. Ohjelmiston testaus

2.1. Yleistä testauksesta

Ohjelmistotestaus on erittäin luova ja älyllisesti haastava tehtävä, jolla voidaan antaa lisäarvoa ohjelmistolle (Myers 1979, s.16). Ohjelmiston virheettömyys ja stabiilisuus ovat todennäköisesti positiivisia vaikutuksia loppukäyttäjälle.

Ohjelmiston testaaminen on suuri kokonaisuus, joka koostuu koko ohjelmistotuotteen kehitysprosessin testauksen ohjelmiston määrittelydokumenteista alkaen aina ohjelman käyttöohjeeseen.

Pentti Pohjolaisen (2003, s. 7) mukaan testauksen tärkeyttä ei pidä vähätellä ohjelmointiprosessissa. Mahdollisimman oikein toimivien ohjelmien ja ohjelmistojen tekeminen tehokkaasti ja taloudellisesti on ollut aina ongelma. Järjestelmien ylläpitokustannukset voivat nousta korkeiksi – testauksessa ja jäljityksessä jopa 50 – 80 % kokonaiskustannuksista. Testausmenetelmillä ja apuvälineillä voidaan vähentää testauksen kustannuksia esimerkiksi automatisoimalla testausta.

Testauksen merkitys kasvaa voimakkaasti sitä mukaan kun tuotteet monimutkaistuvat ja komponenttien lukumäärä kasvaa. Erityisesti toiminnallinen testaus, josta nykyisin on automatisoitu 20 – 30 %, tulee olemaan merkittävä automatisoinnin kohde. (Jot Automation, 2001)

2.2 Testauksen määrittely

Testauksen voidaan yleisesti ajatella tarkoittavan vain ohjelmiston oikeellisuuden tutkimista. Testauksen tulos kertoo tällöin, miten hyvin tai huonosti ohjelma vastaa määrittelyä. Jos ohjelmistoa ei voida todistaa oikeaksi millään formaalilla menetelmällä, on ohjelman oikeellisuutta tutkittava ohjelmistoa suorittamalla eli etsimällä ohjelmistosta virheitä. (Paakki 2004, s. 1-2) Testauksen tarkoituksena on osoittaa, että ohjelmisto toimii tai ohjelma ei toimi tietyissä tilanteissa. On hyväksyttävä se tosiasia, että kaikkia virheitä tai vikoja ei voida löytää ohjelmistosta testauksen aikana, koska ohjelmistot voivat olla laajoja (Enqvist et. al. 2001, s. 3).

Ohjelmistotestaus kuuluu olennaisena osana ohjelmistotuotantoprosessiin. Testaamalla pyritään systemaattisesti löytämään tuotteessa olevia virheitä. Lisäksi testauksen tarkoituksena on varmistaa, että kehitetty tuote täyttää sille kehitysprosessin alussa annetut vaatimukset. Tuotteen testaus vie tyypillisesti yhtä kauan aikaa kuin tuotteen kehitys. Testauksen pitäisi olla kattava, mutta täysin kattavan testauksen toteuttaminen on mahdotonta testauksen resurssien vähäisyyden takia.

Testaukselle on sen historian aikana annettu useita erilaisia määritelmiä. Tavanomaisessa puhekielessä termillä testaus tarkoitetaan lähes mitä tahansa kokeilemista. 1) Edward Kit määrittää kirjassaan "Software Testing in the Real World" testauksen seuraavasti:

- Testaus varmistaa, että ohjelma tekee, mitä sen pitääkin tehdä.
- Testaus on prosessi, jossa ohjelmaa tai järjestelmää ajetaan tarkoituksena löytää siitä virheitä.
- Testaus on määrittelyvirheiden ja määrittelystä poikkeavuuksien etsimistä.
- Testaus on mikä tahansa toimenpide, jonka tarkoituksena on arvioida ohjelman tai järjestelmän jotain ominaisuutta tai toimivuutta.
- Testaus mittaa ohjelmiston laadun.
- Testaus on prosessi, jossa arvioidaan ohjelmaa tai järjestelmää.
- Testaus varmistaa järjestelmän täyttävän sille asetetut vaatimukset tai todellisten ja odotettujen tulosten eroavuuksien tunnistaminen.
- Testaus varmistaa, että ohjelma suorittaa määrätyt toiminnot oikein.(Nikkanen, 2003)

2.3. Testauksen tarkoitus

Testauksessa pyritään tekemään mahdollisimman kattavia testitapauksia ja löytämään menetelmiä, joilla testattavassa ohjelmistossa ja sen osassa esiintyvät viat saadaan löydettyä taloudellisesti ja ohjelmiston toiminta luotettavaksi. Tärkeintä testauksessa on valita jokaiseen testitapaukseen sopivat ja mahdollisimman monta virhettä paljastavat työkalut ja menetelmät.

Ohjelmiston toiminnallista oikeellisuutta tarkastetaan syöttämällä sovellukselle syötteitä testin ajon aikana ja siitä syntyneitä vasteita. Muita testauskohteita ovat ohjelmiston luotettavuus, suorituskyky, käytettävyys ja itse sovelluksen stabiilisuus. Ohjelmistotestaus

---

1) Kit, Edward (1995). Software Testing in the Real World. Addison-Wesley Professional.

ei paranna ohjelmaa tai sen laatua, vaan testauksella selvitetään ohjelmiston puutteita. Testaus paljastaa myös toteutus- ja suunnitteluvaiheen aikana syntyneet virheet ja antaa ohjeita ohjelmistotuotteen laadun kehittämiselle. Testaukseen osallistuvat ohjelmoijat, itse testaajat sekä myös ohjelmiston loppukäyttäjät.

Testauksen kolme tärkeintä periaatetta Myersin (1979, s. 16) mukaan ovat:
- Onnistuneessa testitapauksessa havaitaan vielä aiemmin löytymätön virhe.
- Hyvässä testitapauksessa on suuri mahdollisuus havaita vielä löytymätön virhe
- Testaus on prosessi, jolla on tarkoitus löytää virheitä ohjelmaa suorittamalla.

Pressmanin (2000, s. 426-427) mukaan virheen korjauksen aiheuttamat kustannukset kasvavat sitä suuremmmiksi, mitä myöhemmin ohjelmiston kehitysvaiheessa virhe löydetään. Jos ohjelma on jo toimitettu asiakkaalle ennen virheen löytymistä, virheen korjaaminen voi olla monta kertaa kalliimpaa kuin määrittelyvaiheessa. Virheet tulisi löytää mahdollisimman nopeasti ja myös vähäisellä vaivalla. Näiden tavoitteiden saavuttamiseksi testaus pitäisi aina suunnitella huolellisesti ja aloitus pitäisi tehdä mahdollisimman aikaisessa vaiheessa.

Ohjelmistojen suunnittelijat ja kehittäjät testaavat ohjelmiansa yleensä vain toimivuuden osalta. Testitapaukset ovat vain sellaisia, joissa ohjelma toimii ja virheet jäävät helposti näkemättä. Jos virheitä ei löydy ohjelman testausvaiheessa, asiakas ne viimeistään löytää. Siksi tarvitaan erillisiä testaajia, jotka eivät tee ohjelmiston kehitystyötä. Testauksen alkuvaiheessa, kun testataan yksittäisiä moduuleja, pitäisi ohjelmistojen kehittäjien olla testauksessa mukana. Hehän tietävät parhaiten, kuinka ohjelma toimii ja mitä heikkouksia ohjelma voi sisältää. (Pressman 2000, s. 467-468)

Testauksen avulla voidaan mahdollisesti osoittaa, että ohjelmissa on virheitä, mutta niiden virheettömyyttä testauksella ei voida osoittaa. Ohjelman testauksella voidaan kattaa vain murto-osa kaikista mahdollista tilanteista ja reaaliaikajärjestelmien tapauksessa testauksen ”todistusvoima” on vieläkin pienempi.(Haikala ja Märijärvi 2004, s. 286-287)

2.4. Testauksen suunnittelu

Haikalan ja Märijärven (2004, s. 283) mukaan testaukseen liittyvät työvaiheet ovat testauksen suunnittelu, testiympäristön luonti, testin suorittaminen ja tulosten tarkastelu. Näihin testauksen eri

työvaiheisiin kuluu tyypillisesti yli puolet ohjelmistoprojektin resursseista, joten testaukseen kannattaa kiinnittää huomiota.

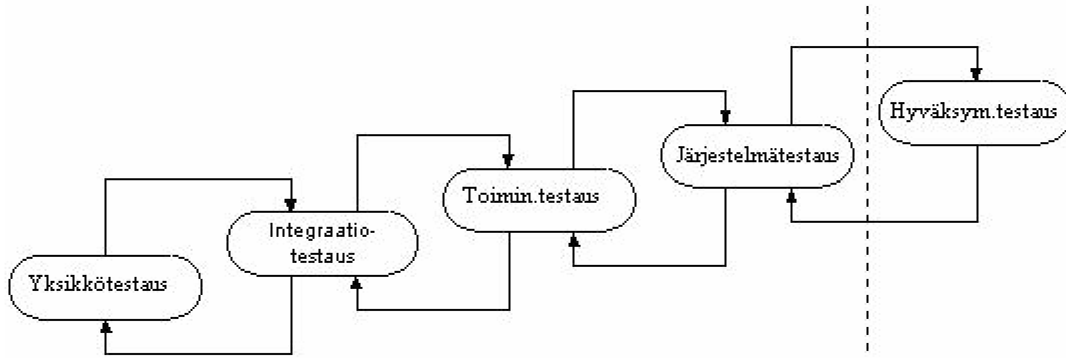
Kun testauksia suunnitellaan, pitää luoda erilaisia testaussuunnitelmia, kuten järjestelmätestaus-, integrointitestaus- ja moduulitestaussuunnitelma. Pienehköissä projekteissa riittää yleensä yksi testaussuunnitelma. Alustava suunnitelma laaditaan määrittelyvaiheessa ja sitä täydennetään myöhemmin suunnitteluvaiheessa. Testaussuunnitelmat voidaan myös sisällyttää projektisuunnitelmaan, toiminnalliseen määrittelyyn ja tekniseen määrittelyyn. Testisuunnitelmasta yleensä selviää mm. mitä testejä tehdään ja milloin, miten ne järjestetään sekä millaisia lopputuloksia odotellaan. Tärkeintä suunnitelmassa on määritellä testien loppumiskriteerit. (Haikala ja Märijärvi 2004, s. 299)

Testitapausten suunnittelussa pyritään laatimaan mahdollisimman tehokkaita ja tarpeellisia testejä, joilla voidaan todeta mahdolliset virheet.
Hyvien testitapausten tunnusmerkkejä Pressmanin (2000, s. 431) mukaan ovat:

- Testillä on suuri todennäköisyys löytää virheitä. Päästäkseen tähän, testaajan täytyy löytää ohjelmasta tilanteita, joissa virheet voivat esiintyä.
- Testitapaus ei ole tarpeeton. Koska aika ja resurssit ovat rajalliset, pitää turhat ja samaa asiaa testaavat tapaukset jättää testaamatta.
- Testin pitää olla paras omassa joukossa. Saman tyyppisten testitapausten joukosta pitää valita se testitapaus, joka kattaa parhaiten koko joukon tavoitteet ja jolla voidaan parhaiten löytää virheet ohjelmasta.
- Testitapauksen ei saa olla liian monimutkainen, mutta ei myöskään liian yksinkertainen. Testattavien asioiden yhdistäminen samaan testitapaukseen voi vaikeuttaa virheiden löytymistä.

2.5. Testauksen vaiheiden jako

Testausprosessi (Kuva 1.) voidaan jakaa pienempiin ja paremmin hallittaviin osiin / vaiheisiin, joita kutsutaan myös testaustasoiksi. Jos testattava kohde ei läpäise jotakin testivaihetta, se palautetaan takaisin tasolle, jolla virhe on tapahtunut tai jolla virheen uskotaan olevan. Tämän jälkeen virheet ja ristiriidat tarvittaessa paikannetaan, korjataan ja kohde testataan uudelleen. (Kautto, 1996)



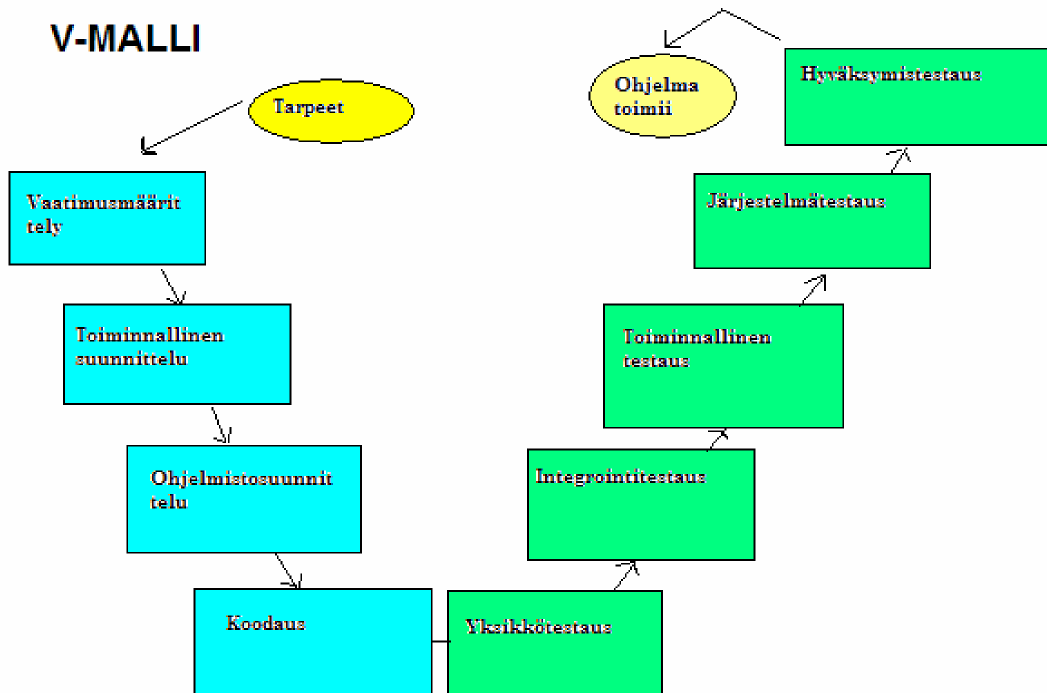
Kuva 1. Testauksen tasot (Muutettu lähteestä Kautto, 1996)

Yleensä kehitettävä ja suunniteltava ohjelmisto on osa suurempaa kokonaisuutta eli tietojärjestelmää, johon se integroidaan. Testauksen tarkoituksena on varmistaa, että ohjelmisto toimii tulevan tietojärjestelmän osana. Testattava ohjelmisto ei saa aiheuttaa ongelmia tai virheellistä toimintaa muissa tietojärjestelmän ohjelmissa.

Vaihejako- eli prosessimalli määrää, miten ohjelmiston koko elinkaari tai ainakin sen toteutusprosessi jaetaan vaiheisiin. Vaihejakomalleja on olemassa useita, joilla kaikilla on omat piirteensä sekä hyvät ja huonot puolensa. Tässä luvussa vaihejakomalleista esitetään yleisin eli vesiputousmalli. (Tersa 2002, s. 15)

Testaus jaetaan eri vaiheisiin V-mallin (Kuva 2) mukaisesti. Testaus etenee V-mallissa yksittäisten moduulien testauksesta aina suurempien kokonaisuuksien testaukseen. Mitä korkeammalle V-mallin testaustasolla ollaan, sen kalliimmaksi virheiden korjaus tulee (Haikala ja Märijärvi 2004, s. 290).

Pressmanin (2000, s. 468-471) mukaan testaukselle määritellään tietyt kriteerit, joiden avulla voidaan ratkaista, koska ohjelma tai sen osa on valmis seuraavaan testausvaiheeseen. Yhtenä kriteerinä voi olla esimerkiksi löydettyjen virheiden määrä. Pressman (2000, s. 428-429) myös ilmaisee, että kaikkien testien pitäisi kohdata aina asiakkaan vaatimukset. Testauksen pitäisi alkaa aina pienistä yksikkötason testeistä kohti suurempia ja laajempia koko järjestelmää koskevia testejä.



Kuva 2. Testauksen V-malli (Muutettu lähteestä Kokkoniemi, 2005)

Tiina Tersan (2002, s. 15) mukaan vesiputousmalli on yksinkertainen ja samalla kuitenkin toimiva. Mallissa liikutaan portaittaisesti vaiheesta toiseen aina projektin alkuideasta elinkaaren loppuun asti. Jokaisen askeleen lopussa tarkastellaan, joko ollaan valmiita seuraavaan vaiheeseen vai tulisiko samaa vaihetta jatkaa ennen siirtymistä.

2.5.1. Moduulitestaus eli yksikkötestaus

Moduulitestauksessa eli yksikkötestauksessa on testattavana yksittäinen moduuli (Enqvist et. al. 2001, s. 5). Moduuli tarkoittaa tietokoneohjelman itsenäistä osaa, jolla on syöttötiedot, tulostiedot ja oma toiminnallinen tehtävä (Wikipedia, 2007). Moduuli koostuu yleensä noin 100-1000 ohjelmarivistä. Moduulin toimintaa verrataan moduulin suunnitteluun ja arkkitehtuurisuunnittelun tuloksiin, tavallisimmin tekniseen määrittelydokumenttiin. Testauksen suorittamiseksi voidaan joutua toteuttamaan testipelejä. Testipeliin voi kuulua ohjelman ympäristöä simuloivia osia. (Enqvist et. al. 2001, s. 6) Moduulitestauksen suorittaa yleensä moduulin toteuttaja (Jaakkola, 2004).

Pressmanin (2000, s. 473-475) mielestä moduulitestauksessa testataan suurempaan kokonaisuuteen kuuluvia yksittäisiä moduuleja erillään. Moduulitestauksessa on tarkoituksena testata kyseisen

moduulin sisäinen toiminta ja varmistaa, että se toimii vaatimusten ja suunnitelmien mukaisesti. Moduulien testaamiseen yleensä tarvitaan testiajureita ja tynkämoduuleita, jotka korvaavat moduulin testaamiseen tarvittavat muut moduulit. Testiajurit ja tynkämoduulit ovat ohjelmakoodeja, joilla emuloidaan muita toteutumattomia moduuleja (Niksula, 1998). Moduulitestauksessa on tarkoituksena löytää seuraavanlaisia vikoja: väärät loogiset operaatiot, vääränlaiset silmukoiden poistumistiet, epäonnistuneet muuttujien vertailut ja virheellinen tietotyyppien vertailu. (Pressman 2000, s. 473-475)

2.5.2. Integrointitestaus

Integrointiprosessissa kootaan yhteen komponentit, joista sitten luodaan yhtenäinen yksittäisiä moduuleita isompi komponentti. Integrointitestauksen tarkoituksena on näyttää, että aiemmin itsenäisesti hyväksytyt komponentit voivat olla yhdistettynä komponenttina virheellisiä tai yhteensopimattomia. Integrointitestauksessa voidaan käyttää sekä lasi- että mustalaatikkotestausta (Määttälä 2006, s. 23). Integrointitestauksessa yhdistellään yhteen moduuleita tai moduuliryhmiä. Painopiste on moduulien välisten rajapintojen toimivuuden tutkimisessa. Testauksen tuloksia verrataan tavallisemmin tekniseen määrittelyyn. Integrointitestaus etenee usein samanaikaisesti moduulitestauksen kanssa ja integrointitestausta onkin usein tarpeetonta tarkastella erillään moduulitestauksesta. (Enqvist et. al. 2001, s. 6)

Pressman (2000, s. 476-479, 481) sanoo teoksessaan, että integrointitestauksessa yritetään löytää ohjelman toiminnan kannalta kriittisiä moduuleita. On erittäin tärkeää, että ne tulee testattua mahdollisimman aikaisessa vaiheessa. Kriittisten moduulien tunnusmerkkejä ovat esimerkiksi suorituskykyvaatimukset, moduulin monimutkaisuus, alttius virheille tai korkean tason hallintatoiminnot. Testaus etenee integrointivaiheessa yleensä joko kokoavasti(bottom-up) tai jäsentävästi (top-down). Kokoavassa testauksessa alemman tason komponentit kootaan yhteen komponentti kerrallaan ja jokaiselle vaiheelle suoritetaan testaus. Jäsentävässä testauksessa testaus aloitetaan korkealta tasolta ja syvennytään tarvittaessa tarkemmille tasoille.

2.5.3. Systeemi- eli järjestelmätestaus

Järjestelmätestauksessa on tarkoituksena testata järjestelmää, joka kuvataan isoksi komponentiksi. Testauksen avulla pyritään löytämään virheet, jotka johtuvat arvaamattomista vuorovaikutuksista alijärjestelmän ja järjestelmäkomponenttien välillä (Määttälä 2006, s. 23). Järjestelmätestauksessa

testataan järjestelmää kokonaisuutena. Järjestelmätestauksella pyritään selvittämään että ohjelmisto toimii oikein ja toteuttaa sille asetetut vaatimukset. Lisäksi testataan erityisesti käyttöliittymän toimintaa ja käytettävyyttä. (Bergström et. al. 2005, s. 3)

Ohjelmistoa tarkastellaan järjestelmätestauksessa loppukäyttäjän näkökulmasta. Järjestelmätestaus suoritetaan mahdollisuuksien mukaan aidossa käyttöympäristössä. (Pressman 2000, s. 483-485)

2.5.4. Hyväksymistestaus

Testausprosessin viimeisenä vaiheena ennen käyttöönottoa on hyväksymistestaus. Hyväksymistestauksessa järjestelmä pyritään testaamaan ohjelmiston oikealla datalla. Näissä testeissä pyritään paljastamaan virheet ja laiminlyönnit järjestelmän vaatimusmäärittelyissä, sillä järjestelmä käyttäytyy todellisella datalla eri tavalla kuin testidatalla (Määttälä 2006, s. 24). Hyväksymistestaus tehdään yleensä järjestelmätestauksen yhteydessä ja sen tarkoituksena on testata, että ohjelma täsmää asiakkaan vaatimuksiin (Haikala ja Märijärvi 2004, s. 290). Hyväksymistestaus suoritetaan asiakkaan kanssa asiakkaalle järjestettävässä ohjelmiston demotilaisuudessa. Jos asiakas haluaa ohjelmistoon muutoksia ja niiden tekoon olisi aikaa, voidaan muutosten toteuttamista harkita. Muissa tapauksissa katsotaan testaus päättyneeksi. (Bergström et. al. 2005, s. 3)

2.5.5. Regressiotestaus

Regressiotestauksella pyritään selvittämään, että muunneltu ohjelma lisäyksen tai muutoksen jälkeen vastaa yhä määrityksiään ja ettei uusia virheitä ole päässyt ohjelmaan. Regressiotestausta voidaan myös käyttää kehitystyössä varmistamaan, etteivät uusien alijärjestelmien lisäykset pääjärjestelmään ole aiheuttaneet tahattomia sivuvaikutuksia. (Määttälä 2006, s. 25)

Regressiotestauksella tarkoitetaan ohjelman uudelleentestaamista muutoksen jälkeen. Regressiotestauksen tavoitteena on paljastaa muutoksen mahdollisesti aiheuttamat virheet ohjelmassa ennen muutosta olleeseen toiminnallisuuteen. Tämä tehdään testaamalla ohjelmaa regressiotestitapauksilla, joista osa on peräisin ohjelman vanhasta testijoukosta ja osa on uusia, jo aiemmin testatun toiminnallisuuden testaamiseen tarkoitettuja regressiotestitapauksia. Uuden toiminnallisuuden testaamista kutsutaan progressiotestaukseksi. Progressiotestausta ei tarvita, jos ohjelmaan ei ole tehty uutta toiminnallisuutta. (Holopainen 2005, s. 8-10) Kun regressiotestausta

suoritetaan useita kertoja korkealta V-mallin testaustasolta, tulee virheiden korjaus kalliimmaksi kuin testaus V-mallin alemmilla tasoilla, koska regressiotestaus vaatii paljon resursseja ja aikaa. Automatisoimalla uudelleentestaus voidaan saada aikaisempaa tehokkaammaksi ja mahdollisesti myös nopeammaksi. (Haikala ja Märijärvi 2004, s. 290)

2.5.6. Käytettävyystestaus

Käytettävyystestauksella pyritään varmistamaan, että käyttäjä pystyy selviämään mahdollisimman hyvin ohjelman toiminnasta. Käytettävyystesteissä yleensä valitaan pieni otos tulevista käyttäjistä ja heidän suoriutumista seurataan valvotussa käytettävyystestejä varten järjestetyssä laboratoriossa. Käytettävyystestaus voidaan myös hoitaa ohjelmistojen käytettävyyden arviointiin erikoistuneiden ammattilaisten avulla (Haikala ja Märijärvi 2004, s. 291). Käytettävyystestaus on yksi tapa tutkia ohjelman käytössä ilmeneviä ongelmia. Testauksessa etsitään käyttäjälle päänvaivaa tuottavia ohjelman vikoja ja puutteita tarkkailemalla käyttäjän toimintaa ohjelman parissa. Kyseessä on testitilanne, jossa käyttäjä tekee hänelle annettuja tehtäviä ja toimii testin järjestäjien antamien ohjeiden mukaisesti ajatellen ääneen tehtäviä tehdessään. Testaus tarkoittaa siis käytön havainnointia irrallaan käyttäjän normaalista käyttöympäristöstä ja käyttötilanteista.

Ongelmakohdat ilmenevät monella tavalla: käyttäjä voi annettua tehtävää ohjelman avulla tehdessään ihmetellä näytöllä näkyviä asioita, tehdä vääriä valintoja tai joutua etsimään toimintoja. Testissä pienetkin ongelmat tulevat ilmi. Testissä paljastuu myös tilanteita, joita käyttäjä ei osaisi ongelmiksi kuvatakaan. Näitä ovat esimerkiksi kiertotiet, joihin käyttäjä turvautuu, kun ohjelman tarjoama suorempi tapa työn tekemiseksi on huonosti esillä eikä käyttäjä huomaa sitä. (Ovaska ja Räihä, 1998)

2.6. Testausmenetelmät

Haikalan ja Märijärven (2004, s. 291) mukaan ohjelmistoa voidaan testata monella eri menetelmällä. Lisäksi eri testausvaiheissa voidaan käyttää erilaisia testausmenetelmiä. Oikean testausmenetelmän valinta riippuu testattavasta ohjelmasta ja aikatauluista. Ohjelmistotestauksessa yleisimmät käytössä olevat menetelmät ovat mustalaatikko-, lasilaatikko- ja harmaalaatikko testaus. Muita ohjelmistotestauksessa käytössä olevia menetelmiä ovat integraatiotestaus, tilastollinen testaus, muuttumattomuustestaus (Pohjolainen 2003, s. 18), virheiden kylväminen (Jaakkola, 2004), mutaatiotestaus (Haikala ja Märijärvi 2004, s. 296), tutkiva testaus (Pyhäjärvi ja Pöyhönen, 2005) ja

äärimmilleen viety testaus (Pohjolainen 2003, s.18) sekä aluetestaus (Koikkalainen, 2000). Myös katselmoinneissa voidaan testata ohjelmistoa.

2.6.1. Mustalaatikkotestaus

Mustalaatikkotestauksessa testitapaukset valitaan testattavan ohjelman vaatimusmäärittelyiden perusteella tutustumatta ohjelman toteutukseen ja yleensä mustalaatikkotestaus suoritetaan V-mallin ylemmillä tasoilla eli hyväksymis-, järjestelmä- ja toiminnallisuutestauksessa (Haikala ja Märijärvi 2004, s. 291-292). Black box-testaus perustuu testattavan systeemin (esim. ohjelma) tai komponentin (esim. funktio) input-output käyttäytymiseen. Black box-testauksessa ei välitetä testattavan kohteen rakenteesta tai sisällöstä, vaan tutkittavana ovat kohteen tulosteet (output) erilaisilla syötearvoilla (input). Testaajalle kohde on siis jokin tuntematon "musta laatikko" (black box). Testattavan kohteen oikeellisuutta tarkastellaan vertaamalla saatuja tulosteita haluttuihin tai odotettuihin tulosteisiin. Testitapaukset johdetaan aina kohteen määrittelyn perusteella. (Enqvist et. al. 2001, s. 8)

Pressman (2000, s. 448) ilmaisee, että black-box testauksessa testataan ohjelman toiminnallisuutta ja vaatimusten perusteella tiedetään, mihin ohjelman tulisi kyetä ja testeillä osoitetaan, että ohjelma siihen kykenee. Mustalaatikkomenetelmällä voidaan löytää erilaisia virheitä, kuten alustus- ja keskeytysvirheiden löytäminen, suorituskyvyssä tai toiminnallisuudessa esiintyvät virheet, virheet tietorakenteissa tai ulkoisen tiedon käsiksi pääsyyn ja väärät tai puuttuvat toiminnat.

2.6.2. Lasilaatikkotestaus

Lasilaatikkotestauksessa(white box) testitapaukset valitaan ohjelman toteutustietojen avulla (Haikala ja Märijärvi 2004, s. 291-292). Lasilaatikkotestauksessa pyritään mahdollisimman kattaviin testituloksiin. Testeissä keskitytään mahdollisimman täydelliseen lausekattavuuteen eli testitapauksien tulisi käydä jokainen järjestelmän sisäisen rakenteen haara läpi (Myers 1979, s. 37-38).

Toisin kuin mustalaatikkotestauksessa ohjelmakoodi on nyt käytettävissä. Lasilaatikkotestaus voidaan jakaa kontrollivirtaan perustuvaan testaukseen, silmukkatestaukseen ja tietovirtaan perustuvaan testaukseen (Paakki 2000, s. 39). Lasilaatikkotestauksessa testitapaukset johdetaan ohjelman sisäisestä rakenteesta ja logiikasta. Tavoitteena on valita testitapaukset siten, että kaikki

kohteen (esimerkiksi ohjelman tai funktion) haarat ja ohjelmapolut tulisi käytyä läpi. Lasilaatikkotestauksessa testidatan valinnalla pyritään siihen, että ohjelma tulee testattua mahdollisimman kattavasti. Testauksen kattavuutta (coverage) voidaan kuitenkin arvioida eri tavoilla, jotka samalla määrittelevät kuinka testidata valitaan (Enqvist, et. al. 2001, s. 10). Pressman (2000, s. 433) esittää, että lasilaatikkomenetelmällä käydään läpi kaikki mahdolliset suorituspolut ja silmukat niiden raja-arvoilla. Sisäisen tietorakenteen oikeellisuuden varmistaminen ja kaikkien loogisten päätösten testaaminen on myös tärkeää lasilaatikkotestauksessa.

2.6.3. Harmaalaatikkotestaus

Harmaalaatikkotestauksessa käytetään hyväksi tietoa ohjelman toteutusperiaatteista. Voidaan hyödyntää esimerkiksi sellaista tietoa, että luvut ohjelmaan luetaan merkki kerrallaan ja muunnetaan samalla 32-bittisiksi kokonaisluvuiksi. Harmaalaatikkotestaus on musta- ja lasilaatikkotestauksen yhdistelmä, jossa tietoa ohjelman toteutuksesta käytetään hyväksi testattaessa komponenttia rajapinnan kautta. Kun tiedetään, kuinka jokin kohta ohjelmasta on toteutettu, voidaan kyseistä kohtaa testata tuon tiedon perusteella (Haikala ja Märijärvi 2004, s. 291-292). Harmaalaatikkotestauksessa ohjelman toteutusperiaatteita käytetään hyväksi testitapausten valinnassa ja sitten testattaessa voidaan käyttää ohjelmiston kriittisiä osia, jolloin tietoisesti valitaan vaikeita tapauksia ja vääriä syötteitä ja näin testataan esimerkiksi ohjelmiston toipumiskykyä virhetilanteista (Karkulehto ja Urpi 2003, s. 2).

2.6.4. Katselmoinnit

Katselmoinnit ovat kokouksia, joissa etsitään lukemalla eri ohjelmistosuunnitelmista ja dokumenteista virheitä. Virheiden etsinnän suorittaa joku muu kuin suunnitelmien tekijä (Kollanus 2004, s. 1). Katselmoinneilla pyritään avustamaan ohjelmiston elinkaaren etenemistä siten, että eteneminen olisi näkyvää ja poistamaan virheet tuotteesta mahdollisimman aikaisessa vaiheessa. Katselmointeja voidaan käyttää minkä tahansa kirjallisen tuotteen laadun varmistamiseen. Tarkoituksena ei ole etsiä ratkaisuja, vaan mahdollisimman paljon virheitä. Katselmointien tulee olla hengeltään positiivisia, rakentavia tilaisuuksia. Tavoitteena on synergiaetujen hakeminen ja kommentointiin kannustaminen. Niissä tarkastellaan tuotetta, ei sen tekijää.

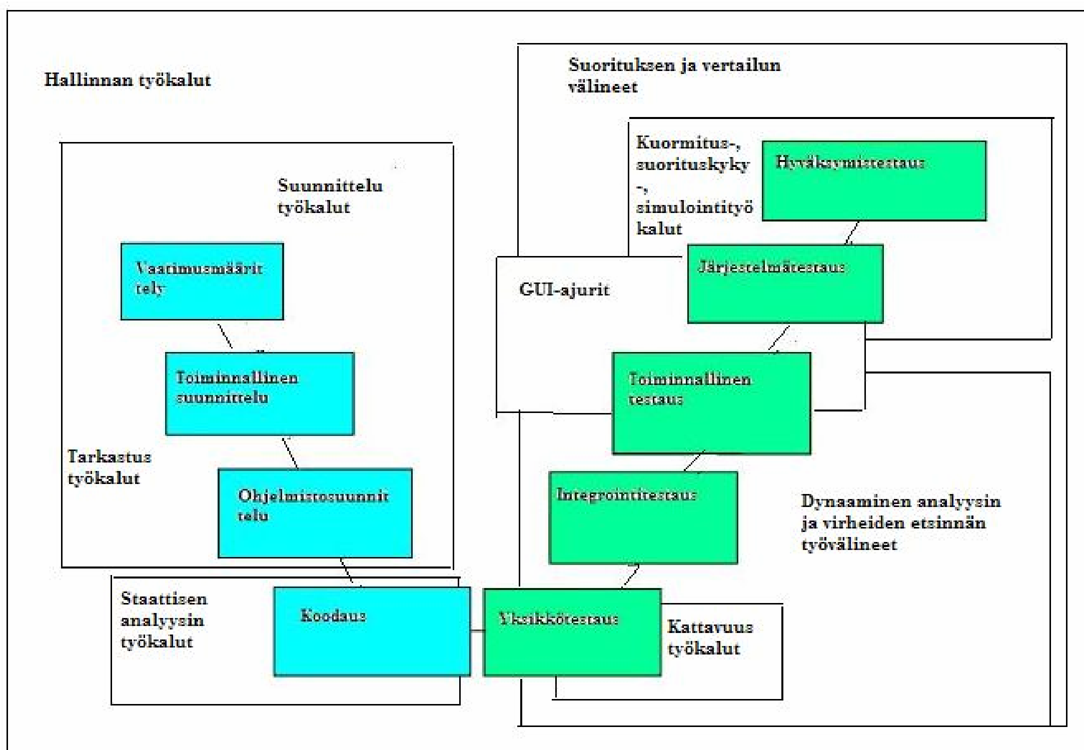
Katselmoinnit kohdistetaan usein vaihetuotteisiin, joita ei voi ohjelman tapaan testata, esimerkiksi tekniseen määrittelyyn. Koodin yhteydessä katselmoinnit eivät korvaa testausta, mutta vähentävät

eri testausvaiheissa löytyvien virheiden määrää ja siten koko projektin työmäärää. Katselmointien tarkoituksena on siis todeta, että jonkin projektin vaihe on päättynyt.

Katselmoinnin hyötynä on se, että katselmointi on testausta tehokkaampi ja halvempi virheiden etsintäkeino. Tämä perustuu siihen, että virhekustannukset kasvavat nopeasti V-mallin vaiheiden funktiona eli mitä myöhemmin virhe löytyy, sen kalliimpi se on korjata. (Nikkanen 2003, s. 25-27)

2.7. Testauksen apuvälineet

Ohjelmakehityksessä käytettävillä testaustyökaluilla tarkoitetaan yleensä työkaluja, joita ohjelmoija ja testaaja voivat käyttää apunaan kehittäessään ohjelmakoodia tai välineitä, joita voi käyttää testauksessa. Työkalun valintaan vaikuttaa muun muassa se, millaisia menetelmiä testauksessa halutaan käyttää ja missä testauksen vaiheessa. (Nikkanen 2003, s. 31)



Kuva 3. Apuvälineiden sijainti eri testauksen vaiheissa (Muutettu lähteestä Pohjolainen, 2003).

Testauksen suorittamiseksi testauksen tulisi olla mahdollisimman pitkälle automatisoitua (Haikala-Märijärvia 2004, s. 297). Testauksen automatisointi on yleensä testitapausten suorittaminen automaattisesti erilaisten työkalujen avulla. Työkalut hoitavat testeissä syötteiden antamisen

testaajien puolesta ja työkalut voivat myös hoitaa tulosten vertaamisen sekä niiden keräämisen (Patton 2001, s. 221-228). Testauksessa käytettävät työkalut voivat olla muun muassa testipetigeneraattorit, ajettavat tekstitiedostot (testiskriptit), emulaattorit, simulaattorit, vertailuohjelmat, kuormageneraattorit, testikattavuustyökalut ja itse testauksen automatisointi (Haikala ja Märijärvi 2004, s. 297). Kuvassa 3 on havainnollistettu eri testausapuvälineet ja niiden sijainti eri kohdissa testausta.

Ohjelmistotuotannon eri tasoilla on käytössä eri testaustyökalut. Kuitenkin eri työkalut ja niiden toiminnot eivät ole niin tarkoin rajattuja, etteivätkö ne olisi käyttökelpoisia jossain muussakin vaiheessa testausta. (Virkanen 2002, s. 36)

Työkaluja käytettäessä onkin aina tunnettava niiden toiminnot ja rajoitteet, jotta tuloksia ei tulkita väärin. Työkalut ovat kuitenkin hyvä lisä testauksessa, sillä ne lisäävät järjestelmällisyyttä, toistettavuutta ja verrattavuutta. (Nikkanen 2003, s. 33)

Testityökalun avulla päästään parempaan testikattavuuteen työkalun mahdollistaessa suuremmat testiaineistot verrattuna testin manuaaliseen suorittamiseen. Oikein suunniteluilla testitapauksilla ja testityökalujen käytön kohdentamisella mahdollistetaan testaukseen panostettujen resurssien tehokkaampi käyttö. Työkalut myös mahdollistavat entistä suurempien testiaineistojen käytön, testien aikaisempaa nopeamman suorittamisen ja saatujen tulosten tarkistamisen. Työkalut tekevät myös vähemmän virheitä kuin testaajat, kun toimitaan suurien testiaineistojen kanssa. (Virkanen 2002, s. 51)

Automatisointiin voidaan käyttää esimerkiksi nauhoita ja toista -työkalua. Testiohjelma suoritetaan yhden kerran manuaalisesti työkalun samalla nauhoittaessa kaikki testauksessa tehdyt toimenpiteet. Nauhoituksen jälkeen testaus onnistuu automaattisesti, kun nauhoitettuja testejä toistetaan. Yökalun käytössä voi tulla ongelmia, jos testit sisältävät hiiren liikkeitä tai napin painalluksia. Liikkeet toistetaan samanlaisina kuin ne on tehty nauhoitushetkellä. Esimerkiksi jos testi alkaa hiukan eri kohdassa näyttöruutua kuin se oli nauhoituksessa, voivat napin painallukset olla virheellisiä. Myös makroja voidaan käyttää nauhoituksen apuna, koska makrot toistavat sitä varten ohjelmoituja toimintoja. Makrojen avulla syötteiden antaminen voidaan automatisoida, jolloin testaajan ei tarvita koko ajan seurata testiajoa. Etuna makrojen käytössä on ylläpidon helppous, jos testattava ohjelma muuttuu. (Patton 2001, s. 228-231)

Haikalan ja Märijärven (2004, s. 299) mukaan ongelmana testaustyökalujen käytössä on niiden vaikutukset järjestelmän toimintaan. Erityisesti sulautetuissa järjestelmissä on tavallista, että kehityslaitteistossa toimiva ohjelma ei toimi kohdelaitteessa tai että testaustyökalun ohjelmaan liittämisen jälkeen ongelmaa ei enää saada näkyviin.

2.7.1. Analysointi/tarkastusapuvälineet

Nämä apuvälineet lukevat lausekielistä ohjelmaa ja tekevät siitä erilaisia analyysejä tai rakenteen kuvauksia ja laskevat erilaisia tunnuslukuja, joita ovat esimerkiksi kompeksisuusmitat (Rajamäki 2004, s. 19).

2.7.2. Testauksen suunnittelun ja hallinnan apuvälineet

Suunnittelun työkalut auttavat määrittämään, millaisia testejä on suoritettava. Ne ovat käytössä vaatimustenmäärittely-, arkkitehtuurisuunnitelu- ja ohjelmistosuunnittelutasolla (Pohjolainen 2003, s. 54). Tähän ryhmään kuuluvat erilaiset testauksen standardit ja ohjeet, testiaineistojen generointityökalut ja projektin hallinnan työkalut. Testiaineistojen generointityökalut ovat käteviä, koska niillä voidaan generoida syöttötietojen kaikki mahdolliset erilaiset kombinaatiot (Rajamäki 2004, s. 19). Tavoitteena on saada minimaalisella määrällä testitapauksia maksimaalinen toiminnallinen kattavuus (Pohjolainen 2003, s. 54). Hallinnan työkalujen tarkoituksena on automatisoida testauksen suunnittelua, analysointia, dokumentointia ja raportointia (Pohjolainen 2003, s. 56). Testauksen hallinnan työkalut ovat kehitetty automatisoimaan testauksen suunnittelua, analysointia, dokumentointia ja raportointia. Markkinoilla on tarjolla ohjelmia ja ohjelmistoja, joilla luodaan oma ympäristökokonaisuus testauksen hallintaan. Tyypilliset hallintaympäristöt helpottavat testauksen organisointia, analysoivat testituloksia, tekevät yhteenvetoraportteja ja lähettävät yhteenvetoja prosessissa mukanaoleville. (Enqvist, et. al. 2001, s. 13) Suunnittelu- ja hallintatyökalukategoriaan luetaan myös virheiden kirjaus- ja seurantavälineet (Nikkanen 2003, s. 32).

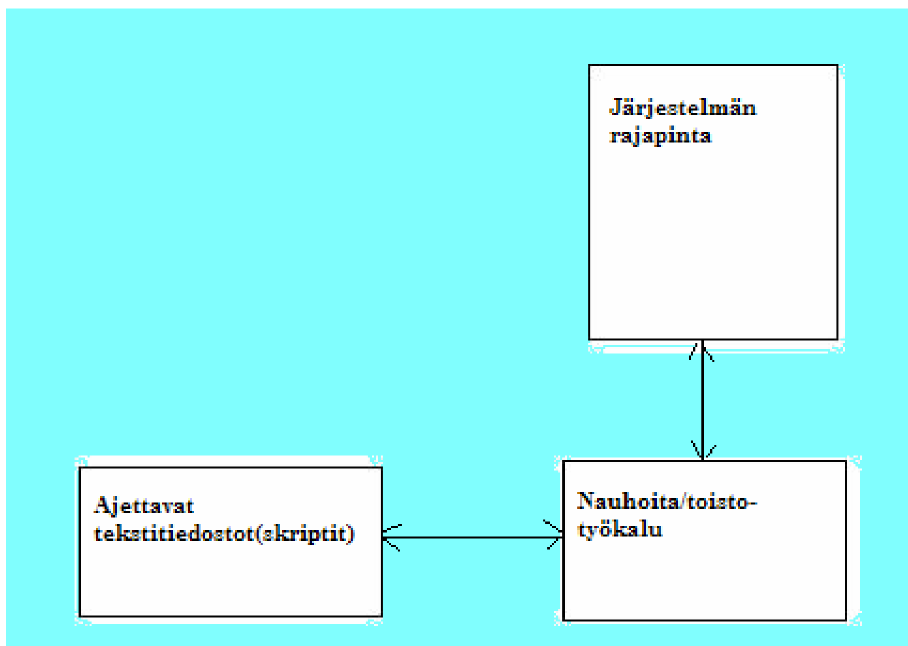
2.7.3. Dynaamiset työkalut testauksessa

Dynaamisia virheenjäljitystyökaluja käytetään yleensä yksikkö- ja integrointitesteissä. Dynaamisella testauksella tarkoitetaan virheiden etsimistä ohjelmasta ohjelmaa tai sen osaa

suorittamalla. Dynaamisessa testauksessa testattavan ohjelmakomponentin ympäristö muodostetaan yleensä testiajureiden ja testitynkien avulla. Testiajurit ja -tyngät voivat olla joko itsenäisiä ohjelmia tai testattavaan ohjelmaan yhdistettyjä erillisiä osia. Testiajurit simuloivat korvaamansa komponenttien antamia kutsuja ja myös pystyä vastaanottamaan kutsumiltaan komponenteilta saamiaan paluuarvoja. Testityngät ovat komponentteja, jotka korvaavat testattavien komponenttien kutsumia komponentteja. (Virkanen 2002, s. 25-41)

Testiympäristön avulla testattavasta komponentista voidaan saada ulos paljon informaatiota testausta suorittaessa. Testattavasta sovelluksesta saatavat tulosteet voidaan tallentaa erillisiin lokitiedostoihin, joista haluttua informaatiota voidaan tarkastella. Testaus suoritetaan joko käsin syöttäen testitapahtumia ja ohjelman tarvitsemia testisyötteitä tai antaen testiajurin lukea annettavat syötteet testattavan järjestelmän ulkopuoliselta tulostuslaitteelta, joka suorittaa tai syöttää testattavalle ohjelmalle sille määritellyt testitapahtumat.(Virkanen 2002, s. 41)

2.7.4. Automatisointityövälineet testauksessa (GUI-ajurit)



Kuva 4. Nauhoitus ja toisto -työkalun periaate (Muutettu lähteestä Stenberg, 2006)

Testauksen automatisointivälineillä tarkoitetaan testauksen suoritus- ja vertailuvälineitä sekä nauhoitus ja toisto –työkaluja. Nauhoitus ja toisto –työkalut suorittavat testin käyttämällä testattavaa sovellusta eli simuloivat sovellukselle manuaalisesti suorittamia toimintoja. Nauhoitus ja toisto -työkalun periaate löytyy kuvasta 4. Testauksen automatisoinnissa testityökalu on testaajan

työtä helpottava apuväline, jolla voidaan nopeuttaa ja vähentää testaajan työtä. (Virkanen 2002, s. 51)

Testien automatisointi on hyödyllistä testaajalle aikaa vievillä ja paljon manuaalista työtä vaativilla osa-alueilla. Automatisoitua testijärjestelmää luotaessa on tärkeää tunnistaa testattavasta sovelluksesta osa-alueet, joihin voidaan soveltaa tarkoitukseen sopivaa testityökalua. (Virkanen 2002, s. 55) Hyvällä automatisoinnin työkalulla on monia samoja piirteitä kuin hyvällä kehitysympäristöllä. Testauksen automatisoinnin täytyy olla ylläpidettävä. Siirryttäessä versioista toiseen on selvittävä mahdollisimman vähillä muutoksilla. Välineen on oltava luotettava. Sen tulee antaa oikeita tuloksia, jotka kohdentuvat tarkasti testattavaan ohjelmaan.(Pohjolainen 2003, s. 55)

Työkalulla suoritettavat testit on kirjoitettu työkalun omalla komentosarjakielellä (testiskriptikieli), joka sisältää yleensä tavallisimmat ohjelmointirakenteet: toistot, haarautumat, funktiokutsut, muuttujat ja muuttujien väliset operaatiot. Testiskriptit ovat sarja käskyjä ja ohjeita testaustyökaluille siitä, mitä testattavalle ohjelmalle tulisi tehdä halutun testin aikaansaamiseksi. (Virkanen 2002, s. 52)

Testiskriptejä pystytään luomaan pelkästään testaustyökalun nauhoitus ja toisto -välineen avulla, mutta myös niitä ohjelmoidaan työkalulle. Testin suorituksen nauhoituksen avulla saadaan kuitenkin usein helpommin ja nopeammin luotua pohja koodille testiskriptin aikaansaamiseksi. Nauhoitus ja toisto -apuvälinettä voidaan käyttää avuksi luotaessa raamit varsinaiselle testiskriptille. Tätä välinettä voidaan verrata tekstieditorin leikkaa/liimaa toimintoon, koska se helpottaa tekstiskriptin tekemistä, mutta itse toiminnan logiikka on lisättävä itse käsin koodaten. (Virkanen 2002, s. 53)

2.7.5. Kuormitus- ja suorituskykytyökalut

Kuormitus- ja suorituskykyä testaavat työkalut auttavat toiminnallisuus-, järjestelmä- ja hyväksymistestausvaiheessa. Nämä työkalut voivat antaa raportteja käyttäjistä, testidatasta, keskimääräisistä vastausajoista yhden istunnon aikana ja vertailla eri istuntojen välisiä tuloksia. (Pohjolainen 2003, s. 55-56) Markkinoilla on työkaluja, joilla voidaan luoda järjestelmään suuria määriä tietoliikennettä tai dataa. Näiden kuormitus- ja rasitustyökalujen avulla voidaan simuloida ohjelman tai järjestelmän käyttäytymistä poikkeuksellisen raskaan kuormituksen aikana.

Simulaattorityökalujen avulla voidaan testata tilanteita, joiden luominen perinteisessä testauksessa olisi vaikeaa, kallista ja mahdotonta. Simulaattoreiden ja testidatageneraattoreiden avulla voidaan nopeasti luoda dataa, jonka avulla on mahdollista testata ohjelman tehtäväkuvauksissa (spesifikaatio) määritellyt syötteiden ylä- ja alarajojen toimintaa sekä simuloida ohjelman käyttäytymistä näiden rajojen ulkopuolella. (Nikkanen 2003, s. 32-33)

2.7.6. Staattisen analyysin työkalut

Staattisen analyysin työkalujen tarkoitus on löytää virheitä ohjelman koodista ilman minkäänlaista ohjelman suoritusta. Yksinkertainen esimerkki staattisen analyysin apuvälineestä on kääntäjä, joka koodin kääntämisen yhteydessä ilmoittaa useista koodista havaituista virheistä. Staattisia analysoijia käytetään yleisimmin ohjelmamittojen laskemiseen ja ohjelman rakenteen analysointiin. Ohjelmamitoista voidaan päätellä ja ennustaa koodin mahdollisia ongelmakohtia sekä analysoida ohjelman monimutkaisuutta.(Kautto, 1996)

2.8. Työvälineen hankinta

Testauksen työkalun valintaan kuuluvat käyttäjän tarpeiden tunnistaminen, välineen arviointimenetelmän luominen, sopivien ehdokkaiden löytäminen ja työkalun valinta ja arviointi siitä, mikä on investoinnin takaisinmaksuaika (Pohjolainen 2003, s. 59). Jos työkalu toteutetaan itse, täytyy työkalun olla tarkoituksenmukainen testauksen tarpeisiin. Työkalulla täytyy olla mahdollista parantaa testattavuutta ohjelmistossa, jota testataan. Dokumentointi ja koulutus eivät todennäköisesti ole kovin hyvin tuettuja itse rakennetussa apuvälineessä. (Pohjolainen 2003, s. 61)

Työkalun ostaminen voidaan jaotella eri vaiheisiin:

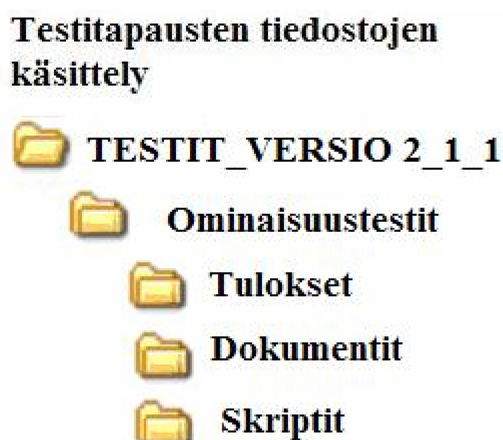
- Tehdään esiselvitys siitä, mitä ominaisuuksia tuotteelta odotetaan.
- Tutkitaan markkinat mahdollisimman laajasti.
- Tarkempien määrittelyjen tekeminen tähän asti saatujen tietojen avulla. Myös lopulliset käyttäjät voivat olla mukana tässä vaiheessa.
- Valitaan ehdolle kaksi tai kolme työkalua tarkempiin tutkimuksiin.
- Lopuksi pyydetään työkalun kokeiluversiot toimittajilta.
- Tuotteen valinnan jälkeen järjestetään koulutus ja käyttöönotto. (Pohjolainen 2003, s. 60)

Kun työkalu on valittu, alkaa todellinen työvälineen käyttö. Vaikka valinta olisikin tehty huolellisesti, ei ole mitään takeita työkalun käyttämisen onnistumisessa. Monet yritykset, jotka ovat menestyksellisesti valinneet ja hankkineet työkalun, eivät ole saaneet mitään hyötyä työkalusta. Työkalun hyödyt ovat voineet jäädä vähäisiksi, koska työkalua ei ole koskaan otettu käyttöön tai työkalu on jäänyt pois käytöstä hyvin pian. (Pohjolainen 2003, s. 61) Työkalun valinnassa pitäisi aina kuulla testaajien mielipidettä ja selvittää todelliset tarpeet testauksessa, että taloudelliset resurssit eivät menisi hukkaan ja työkalun valinta onnistuu.

Testauksen automatisointi ja työkalujen käyttö ovat apuna testauksen laadun lisääntymiseen. Markkinoilla on kuitenkin valtava määrä vaihtelevan laatuisia ja erilaisiin tarpeisiin sopivia testaustyökaluja, joten oikean työkalun tai työkalujen löytäminen saattaa vaikuttaa mahdottomalta tehtävältä. Monesti työkalun hankintapäätös tehdään hätiköiden ja huono valinta kostautuu myöhemmin työkalun jäämisellä käyttämättömäksi. Työkalun valintaprosessissa tulisikin kartoittaa tarpeet ja tavoitteet mahdollisimman systemaattisesti.(Mäkelä 2000, s. 7)

2.9. Testauksen tulosten ja lokien käsittely

Stenbergin (2006, s. 36) mukaan pitää testaajien yhdessä päättää eri tiedostojen nimeämiskäytännöt ja tallennuspaikat, koska nämä molemmat keinot auttavat luonnollisesti tiedostojen uudelleenkäyttöä ja tukevat regressiotestausta sekä ovat myös apuna testauksen kehitystyössä. Selvä hakemistorakenne (kuva 5) helpottaa testaustoimintaa ja nopeuttaa sitä. Yhteisillä tiedostojen nimeämiskäytännöillä ja tallennuspaikoilla pystytään tunnistamaan helpommin testeissä käytetyt ajettavat tekstitiedostot (skriptit). Ehkä tärkein apu hyvän tiedostohierarkian luonnissa on, että yhteiset säännöt tiedostojen käsittelyssä rajoittaa eri tiedostojen turhaa monistumista ja luo mahdollisuuden jonkinlaiseen versionhallintaan sekä testeissä käytetyt tiedostot ovat kaikki verkossa kaikkien saatavilla yhteisellä verkkolevyllä mahdollistaen varmuuskopioinnin. Yhteiset tiedostojenkäsittelysäännöt voivat myös vähentää dokumentoinnin tarvetta testauksen eri tasoilla.



Kuva 5. Testien ja testitulosten käsittely (Muokattu lähteestä Stenberg, 2006)

Automatisoinnista seuraa myös erilaisten tiedostojen määrän raju kasvu, koska ennen testausta ja sen jälkeen voi syntyä erilaista dataa, skriptejä, odotettuja tuloksia, vertailuja ja muita dokumentteja. Ohjelmistokehityksessä ja testauksessa useat ongelmat alkavat, kun suunnittelijat ja testaajat omivat työnsä, eivätkä dokumentoi tai tallenna niitä versionhallintaan. Tarvitaan siis yhteiset pelisäännöt ja yhteinen tiedostojen tallennuspaikka. (Stenberg 2006, s. 35)

Testitiedostojen tallentamisessa pitää ottaa huomioon mahdollinen uudelleen käyttö. Jokaiselle tiedostolle määritellään oma paikkansa ja myös tiedostojen on oltava omalla paikallaan. Hakemistorakenne ja nimeämiskäytännöt tuovat näkyvyyttä ja ne luovat selkeyttä tietojen tallentamiseen. Tunnistetaan myös yhteisiä osia eli on olemassa yleiskäyttöisiä skriptejä, testikirjastoja ja käyttötapauksien pilkkominen pienempiin osiin.(Stenberg 2006, s. 24)

2.10. Testausympäristön ylläpidettävyys

Stenbergin (2006, s. 22-23) mukaan testausympäristön ylläpidettävyys on tärkeää. Testattavaan järjestelmään voi tulla muutoksia projektin aikana tai eri julkaisujen välillä. Muutosten määrä voi vaihdella, mutta tähän pitää testauksessa varautua. Monet asiat vaikuttavat ylläpidettävyyteen, kuten testitapausten lukumäärä, testitapauksissa käytetyn datan määrä, käytetyn datan formaatti, testaukseen käytettävä aika, testitapausten testattavuus, testien keskinäiset riippuvuudet, nimeämiskäytännöt, testien monimutkaisuus ja dokumentointi. Automatisoitujen testien ylläpidossa on tärkeää, että muutokset minimoidaan. Ylläpidossa pitää ottaa huomioon, että muutettavien tiedostojen määrä on pieni, päällekkäisiä testejä ei ole ja tiedostojen ja komentojonojen (skriptit)

turha monistuminen pitää estää. Ylläpito vaatii suunnittelua eli testitapaukset pitäisi käydä säännöllisesti läpi, jotta voidaan poistaa turhat ja päällekkäiset testit (Stenberg 2006, s. 4).

On tärkeää optimoida testausympäristöä, koska pieni määrä skriptejä on helpompi ylläpitää ja testijoukon säännöllinen läpikäynti on myös tärkeää. Tämä mahdollistaa myös sen, että uusien testien kehittämiseen menee aikaisempaa vähemmän aikaa ja erilaisia testiskenaarioita voidaan luoda helposti yhdistelemällä ja muokkaamalla valmiita skriptejä. Optimointi vaikuttaa myös dataan ja vertailuihin. (Stenberg 2006, s. 25)

Dokumentoinnissa on tärkeää, että käytetään yhtenäisiä tiedostomalleja ja -pohjia. Tämä edistää uudelleenkäyttöä ja tiedostoihin ei kannata liittää liikaa dokumentointia, koska tiedostoja voidaan hajottaa pieniin osiin. (Stenberg 2006, s. 28)

3. Testauksen automatisointi

3.1. Testauksen automatisoinnin hyötyjä

Testauksen automatisoinnintia voidaan kutsua myös tietokoneavusteiseksi testaamiseksi. Automatisointi on voi olla tietoinen sijoitus, jonka tuotot tulevat vasta jälkeenpäin. Aluksi automatisointi vaatii paljon panostusta, jotta hyötyjen maksimoiminen olisi mahdollista. Automatisointi voi vaatia 5-10 kertaa suuremman työmäärän manuaaliseen testaukseen verrattuna. Työkalun ja arkkitehtuurin rakentaminen vie aikaa. (Stenberg 2006, s. 2)

Automatisoinnin yhteydessä testaajan rooli muuttuu siten, että testaajasta tulee myös ohjelmoija. Automatisointi vaikuttaa myös töiden organisointiin, koska automatisointi voi luoda uusia työtehtäviä. Testien ylläpito ja uudelleenkäyttö ovat ratkaisevassa asemassa automatisoinnin menestyksessä tai epäonnistumisessa. Automatisoinnissa tarvitaan myös eri testitietojen, lokien ja tulosten hallintaa testausympäristössä. (Stenberg 2006, s. 11)

Testauksen automatisoinnilla saadaan testitapausten tuottaminen ja suoritus manuaalisia menetelmiä nopeammaksi. Testitapausten tuottamisen automatisointiin on ainakin kahdenlaisia lähestymistapoja. Puoliautomaattisessa tuottamisessa testitapaukset johdetaan testausohjelmointikielistä. Täysin automaattinen testitapausten tuottaminen perustuu jonkin algoritmin mukaiseen testattavan tila-avaruuden läpikäyntiin. Tässäkin tapauksessa saatetaan tarvita ihmisen apua rajoittamaan tila-avaruuden kokoa. (Prosessori,1998)

Testitapausten suorittamisen automatisointi perustuu usein jonkinlaisen komentotiedoston käyttöön. Kun ohjelmistoa päivitetään, tulee suuren testiaineiston päivittäminen erittäin työlääksi. Lisäksi suuren testiaineiston seurauksena saadaan myös erittäin suuri määrä testituloksia. Automaattisilla tilatestereillä saadaan testiaineiston ylläpito-ongelmaa pienemmäksi, mutta testitulosten analysointi jää usein edelleen testaajan harteille. Tilatesteri on työkalu, joka käy ohjelmistokomponentin kaikki tilat läpi. (Prosessori,1998)

Kun järjestelmä testataan ilman mitään automatisointia eli täysin manuaalisesti, on inhimillisiä virheitä ja puutteita aina mukana. Kaikkea ei ehdi testata ja toisaalta kaikkea oleellista ei aina huomata. Automatisoinnille on siis olemassa tilaus. Testausapuvälineet tukevat niin manuaalista

kuin automaattista testaamista. Niillä voidaan nopeuttaa ja tehostaa testaamista. Testauksen automatisointi edellyttää yritykseltä investointeja niin välineisiin kuin osaamiseen. Organisaatio voi aloittaa testauksen kehittämisen pienillä panoksilla ja kehittää toimintaansa lisäten automatisointia vähitellen. Tietojen ja taitojen lisääntyessä saadaan automatisoinnista ja muista testausapuvälineistä enemmän hyötyä ja testaaminen tehostuu. Ohjelmatoimittajat järjestävät koulutusta myös räätälöidysti yrityksen omaan ympäristöön. (Valtonen 2000, s. 45)

Kun mietitään testauksen automatisointia, on tärkeätä miettiä yleensäkin automatisoimisen soveltuvuutta testauksessa. Testauksen automatisointi vaatii paljon lisäkustannuksia normaalin sovelluksen rakentamisen lisäksi. Testauksen automatisointia ei kannata pitää itsestäänselvyytenä, vaan on tarkkaan mietittävä milloin automatisointi kannattaa. Jos testejä ei tarvitse toistaa, eivät automatisoinnin lisäkustannukset ole perusteltuja. Mutta jos testaustoimintoja automatisoidaan, on automatisointi parasta aloittaa rutiinitoiminnoista eli testauksen suorittamisesta ja saatujen tulosten vertailemisesta odotettuihin tuloksiin. (Määttälä 2005, s. 32)

Pattonin (2001, s. 220-221) mukaan automatisoinnilla saavutetaan esimerkiksi seuraavia hyötyjä:

- Testaus nopeutuu, koska automatisoinnilla voidaan lyhyessä ajassa tehdä enemmän testitapauksia kuin manuaalisesti.
- Suoritettu testi on luotettava – inhimillisiä erehdyksiä ei tule. Automatisoinnilla voidaan testit suorittaa joka kerta samalla tavalla
- Testien toistettavuus – saman testin voi suorittaa automatisoinnilla niin monta kertaa kuin haluaa
- Testauksen tehokkuus paranee. Kun ajetaan testitapauksia manuaalisesti, ei voi tehdä mitään muuta. Automatisoinnilla voidaan saada lisää aikaa testaussuunnittelulle ja uusien testien suunnittelulle.

Testaajat kokevat automaattisen testaamisen päähyödyn siinä, että testaamisen suoritus nopeutuu ja sovelluksen uudelleentestauksessa manuaalisen testaamisen määrä vähenee nauhoitettujen testien ansiosta. Lisäksi automatisoinnin tärkeä hyöty on siinä, että testaaminen tulee manuaalista testausta turvallisemmaksi ja kattavammaksi. Samassa ajassa automatisoidut testit saavat aikaan enemmän

tuloksia kuin manuaalisesti suoritetut testit. Toisaalta automaattitesti löytää virheet varmemmin kuin henkilö. Kaikkea ei toki kannata eikä voikaan automatisoida. Sellaiset rutiinit automatisoidaan, joita on tarve ajaa usein ja joiden sisältö pysyy melko vakaana. Kertaluonteisia testejä ei siis kannata automatisoida. (Valtonen 2000, s. 45)

Muuttuneet työtavat ohjelmistotuotannossa ja automatisoitu testaaminen parantavat ohjelmistojen laatua. Ohjelmistojen ominaisuuksien ja usein epärealististen aikataulutavoitteiden sijaan nyt korostetaan testaamista. Automatisoitujen menetelmien avulla testaamisen volyymi kasvaa. Testiautomaatiota kehittämällä voidaan osittain myös estää manuaalista testausta vaativien työvaiheiden siirtämistä halpojen työvoimakustannusten maihin. (Karvonen, 2006) Järjestelmien monimutkaistuessa mallipohjainen testaus ja automatisointi ovat ainoa keino saada tehokkuutta ja tuottavuutta testaukseen (Lahti, 2004).

Tulevaisuudessa voidaan luoda älykkäitä testausjärjestelmiä, joissa järjestelmä tietää, mitä testata. Spesifikaatiopohjaisella testauksella haetaan ajoitukseen, järjestykseen ja rinnakkaisuuteen liittyviä virheitä, joiden läpikäyminen käsin on työlästä ja monilta osin mahdotonta ilman jonkinlaista työkalua. (HETKY, 2002) Spesifikaatiopohjainen testaaminen tarkoittaa testitapausten kehittämistä ohjelmiston vaatimusmäärittelystä (Harju ja Koskela 2003, s. 29).Uudenlaiset automaatiot nostavat osaltaan abstraktiotasoa testeissä ja helpottavat muutosten tekemistä ja ylläpitoa. Tämä ei korvaa aiempia tapoja testata, vaan täydentää niitä vastaten haasteisiin, joihin olemassa olevalla teknologialla ei voida vastata. (HETKY, 2002)

Käytännössä ohjelmistojärjestelmien monimutkaisuus ja järjestelmiin kohdistuvat odotukset ovat saavuttaneet tason, jolla ohjelmistojen testaaminen nykyisillä tavoilla ja menetelmillä ei onnistu tarpeeksi hyvin. Tehokkaalla suunnittelulla voidaan kohdistaa käytössä olevat resurssit tärkeimpiin asioihin, mutta testauksessa tapahtuva toistokin on jo osoittanut, että automatisointi on tarpeen. On kuitenkin oleellista ymmärtää oma tekemisensä, ennen kuin ryntää suin päin työkalujen kimppuun, sillä niistä saadut hyödyt riippuvat omien toimintatapojen strukturoinnista.(HETKY, 2002)

Tärkein tekijä onnistuneen testauksen läpiviennille on järjestelmällisten, määrämuotoisten menetelmien käyttö. Testauksen järjestelmällisyyden kasvattaminen on eräs olennainen tehtävä, jolla ohjelmistoprojektia ja kehitettävän järjestelmän ja tuotteiden laatua saadaan parannettua. Lisäksi on tärkeää kerätä tietoa ja mitata testaukseen liittyviä suureita, kuten testaukseen käytettyä aikaa ja löydettyjen virheiden määrää. (Nikkanen 2003, s. 83)

Erityisesti regressiotestuksen automatisoinilla voidaan saada paljon etuja. Koska regressiotestauksessa toistetaan valmiita testitapauksia, niiden automatisoitu suorittaminen säästää aikaa ja resursseja. Esimerkiksi testauksen suunnitteluun ja manuaalisten testien suorittamiseen jää aikaisempaa enemmän aikaa, jos suuri osa regressiotestausta on automatisoitu. (Patton 2001, s. 220)

Tietokoneet, jotka olisivat yöt toimettomina, saadaan automatisoinnin avulla hyötykäyttöön. Siten testaajat voivat keskittyä uusien testitapausten suunnittelemiseen ja automaattisia testejä harvemmin suoritettavien manuaalisten testien suorittamiseen. Näin saadaan kokonaisuutena käytettävissä olevat resurssit paljon parempaan käyttöön. Myös testitapausten uudelleenkäyttö tehostaa resurssien käyttöä, sillä valmiita automatisoituja testejä voidaan hyödyntää kaikilla testaustasoilla. Samalla näistä testitapauksista tulee laadultaan aikaisempaa parempia, koska niitä voivat käyttää useat testaajat samanaikaisesti ja testitapausten suunnitteluun voidaan käyttää entistä enemmän aikaa. (Patton 2001, s. 234-238)

Testauksen automatisointi voi parantaa ohjelmistojen laatua suoranaisesti lisäämällä testaajien aikaa perehtyä testaamaan toimintoja, joita ei voida automatisoida. Samalla resurssien määrällä, joka käytetään manuaaliseen testaamiseen, voidaan lisätä testauksen määrää ja kattavuutta automatisoidun testausjärjestelmän avulla. (Virkanen 2002, s. 36) Korkean kattavuuden lisäksi on tärkeää myös testitapausten ylläpidettävyys, jolloin testitapaukset ovat helpommin muokattavissa uusiin tarpeisiin. (Määttälä 2005, s. 32)

Hyvä automatisoitu testitapaus on Stenbergin mielestä sellainen, että sillä todennäköisesti voidaan löytää vika ohjelmasta ja eristää testin avulla löydetty vika. Automatisoitu testitapaus testaa ainoastaan tiettyä selkeää kohdetta ja eikä ole kokonaan samanlainen muiden testien kanssa. Se ei saa olla liian yksinkertainen tai monimutkainen. (Stenberg 2006, s. 33)

3.2. Automatisoinnin kompastuskiviä

Kaikkia testejä ei kannata koskaan automatisoida. Jo pelkästään turhien ylläpidettävien testauskoodien määrä kasvaa kohtuuttoman suureksi, jos kaikki testitapaukset ovat automatisoitu. Osa testeistä suoritetaan vain hyvin harvoin tai niitä ei yksinkertaisesti voi automatisoida. Siksi on hyvin tärkeää ajatella, mitkä testit kannattaa automatisoida ja mitkä tehdään manuaalisesti.

Seuraavia testejä ei kannata automatisoida:

- Testi suoritetaan vai kertaalleen tai hyvin harvoin. Tällaisen testin automatisoinnista ei saada yleensä ainakaan ajallisia säästöjä.
- Kun etukäteen on tiedossa, että ohjelman seuraavassa versiossa jokin osa ohjelman toiminnallisuutta muuttuu hyvin paljon. Jos esimerkiksi ohjelman käyttöliittymä vaihtuu kokonaan, ei käyttöliittymän testejä kannata automatisoida.
- Testiin sisältyy jokin vain ihmiselle helposti näkyvä tarkistus. Tietokoneen on yleensä mahdotonta suorittaa automaattisesti esimerkiksi ohjelman ulkoasun testaus, kuten värien sopivuuden tarkistaminen.
- Testiin kuuluu jokin fyysistä toimintaa vaativa vaihe, kuten levykkeen vaihtaminen. (Patton 2001, s. 239)

Testauksen automatisoinnin ongelmista syytetään usein työkalua, mutta usein todellinen syy testauksen automatisoinnin epäonnistumiseen on suunnitelmallisuuden ja arkkitehtuurin puute (Stenberg 2006, s. 4).

Yleisempiä kompastuskiviä automatisoidessa ovat:

- Testauksen automatisointi laitetaan osaamiseltaan väärien ihmisten huoleksi.
- Ajetaan liian isoja testitapauksia.
- Pidetään automaattista testausta samanarvoisena manuaalisen testauksen kanssa.
- Työkalun antamat raportit ovat huonoja.
- Automatisoinnin ohjeet puuttuvat.
- Testaustyökaluissa ei ole tarvittavaa toiminnallisuutta tai ne eivät toimi hyvin aidossa testausympäristössä.
- Järjestelmän testattavuus on huono.
- Testauksen kattavuudesta ei ole tietoa.
- Automatisoinnin tuominen testauksen suhteen kypsymättömän organisaation.
- Testattavaan ohjelmaan tulee paljon muutoksia juuri ennen testausta.
- Automatisoitu testaus vaatii liian paljon työtä ja resursseja.
- Organisaatiolla on automatisointiin ylisuuret odotukset.
- Johdon todellinen tuki puuttuu automatisoidun testauksen käyttöönotolle. (Stenberg 2006, s. 78-80)

4. Ohjelmistotestausprosessin kehittäminen

4.1. Lähtökohdat kehittämiselle

Ohjelmistotestausprosessi koostuu joukosta toimintoja, joiden onnistuneen toteutuksen tuloksena syntyy haluttu lopputulos. Ohjelmistotestausprosessia voidaan parantaa monin eri tavoin. Testauksen parannuskohteita voivat olla testausprosessin hallintatapa, käytetyt menetelmät tai käytetyt työkalut. Muutosten toteutus täytyy suunnitella huolellisesti, koska huonosti toteutettu muutos voi aiheuttaa muutosvastarintaa. (Nikkanen 2003, s. 75)

Testausprosessin lähtökohdat kehittämiselle ovat seuraavanlaisia:

- Prosessin kehittäminen vaatii johdon tuen.
- Kaikkien prosessin osallistuvien pitää sitoutua parantamiseen.
- Selkeät tavoitteet ja testauksen nykytila
- Kehittäminen on jatkuvaa ja se sisältää oppimista.
- Tehtyjä muutoksia tulee ylläpitää ja valvoa.
- Kehittäminen vaatii investointeja. (Nikkanen 2003, s. 75)

4.2. Käytännön toimenpiteet kehitettäessä testausta

Testauksen kehittäminen käytännössä etenee seuraavasti:

- Tunnistetaan testausprosessin tila ja ymmärretään se, kuinka sitä voidaan parantaa.
- Tehdään päätökset kehittämisideoista ja tehdään niistä dokumentit.
- Toteutetaan mahdolliset kehittämistoimenpiteet. (Nikkanen 2003, s. 77)

Nikkasen (2003, s. 77) havaintojen mukaan hallittuun testausprosessin voidaan päästä seuraavilla toimenpiteillä:

- Katselmointien käytöönotto kaikilla tasoilla aina koodin tarkastuksesta käyttöohjeiden tarkastukseen.
- Testaussuunnitelmien tehostaminen.
- Versionhallinnan järjestäminen.
- Testauksen suunnittelu ja aikataulujen laadinta.
- Vakiintuneiden testausmenetelmien käyttöönotto.

- Testaustyökalujen käyttöönotto.
- Eri mittareiden määrittely ja käyttöönotto testausprosessissa.
- Tiedonkulun lisääminen ja viestinnän helpottaminen.

Kaikkea ei voi muuttaa yhdessä yössä, joten aluksi tulee valita muutama kehitysidea. Testauksen kehittäminen täytyy suunnitella ja toteuttaa huolellisesti. Myös yrityksen johdon tulee sitoutua muutokseen. Yrityksen kaikille osapuolille tulee selvittää nykyiset ongelmakohdat testauksessa ja syyt testausprosessin kehittämiseen sekä saatavat hyödyt testauksen kehittämisestä. Pelkkä tieto siitä, miten ohjelmatuotannon eri osa-alueita tulee hoitaa, ei riitä, vaan eri prosesseja tulee parantaa aktiivisesti. (Nikkanen 2003, s. 82)

5. Wipron ohjelmistotestauksen kehittäminen ja automatisointi

Tarkoituksena oli kehittää Wipron testausta aikaisempaa nopeammaksi, nopeuttaa tai automatisoida rutiineja sekä muuttaa testaukseen liittyviä toimintatapoja. Tässä kappaleessa käydään asioita, joita korjaamalla voidaan nopeuttaa testaamista ja tulosten analysointia Wiprossa.

5.1 Toimintatavat

Testitapauksia tehdessä havaitsin, että testikonfiguraatio ja perusrutiinit ovat samankaltaisia ja testitapaukset eroavat toisistaan perustoiminnallisuudeltaan vähän. Yksi todennäköinen muutos testitapausten teossa on tehdä yksi isompi testitapaus testikantaan, joka sisältää useita testitapauksia. Testitapaukset kuvataan ns. ranskalaisilla viivoilla ja testistä kirjataan vain olennainen. Isossa testitapauksessa testauksen perusrutiinit ja perustoiminnallisuudet ovat samanlaiset jokaisen testin kohdalla.

Toinen heikkous, jonka olen havainnut testauksessa, on makrojen ja erilaisten komentojonojen (skriptien) sekä testien tulosten sijainti. Ne ovat nykyään aika sekaisin eri verkkolevyillä ja eri hakemistoissa. Ne pitäisi laittaa oikeisiin hakemistoihin ja mahdollisesti samalle verkkolevylle. Eri tiedostojen sijoittelu nopeuttaisi testausta ja tulosten analysointia. Myös makrojen ja komentojonotiedostojen (skriptien) päivittäminen on jäänyt kokonaan tai niiden viimeisimpiä versioita ei löydy heti tai ne sijaitsevat jonkun henkilökohtaisessa hakemistossa. Pitäisi löytyä vastuuhenkilö, joka vastaa niiden päivittämisestä ja joka etsii makrojen ja skriptien viimeisimmät versiot. Lisäksi kaikilla pitää olla oikeudet niihin hakemistoihin, joissa makrot ja komentojonotiedostot (skriptit) sijaitsevat.

5.2 Rational Visual Tester

Wiprossa otetaan käyttöön uusi työväline, joka voi auttaa testausta ja joka nopeuttaa testauksen rutiinitehtävien tekemistä. Työväline on Rational Visual Test 6.5. Sen avulla voidaan automatisoida toiminnallista testausta Windows-ympäristössä ja sen sovelluksissa. Rational Visual Test 6.5 -työvälineellä voidaan luoda paljon massatietoa testausympäristöä varten. Tietoja ei tarvitse enää luoda manuaalisella tavalla, joka hidasta ja työlästä. Rational Visual Test –työvälineellä voidaan

luoda uudelleenkäytettäviä, ylläpidettäviä ja laajennettavia komentojonotiedostoja (skriptejä) (Rational Visual Test, 2007)

Rational Visual Test 6.5 on automaattinen toiminnallinen testausväline, joka auttaa kehittäjiä ja testaajia nopeasti luomaan testitapauksia Windows-sovelluksiin, jotka on luotu millä tahansa kehitysvälineellä. Rational Visual Test -väline on yhdistetty Microsoft Developer Studion kanssa sekä työpöytäkehitysympäristön kanssa ja Rational Visual Test -välineellä on myös yhteistyötä Microsoft Visual C++ kanssa. (VersionTracker, 2007) Rational Visual Test on yksi laajimmin levinneitä ja käytettyjä graafisen käyttöliittymän(GUI) automaatiovälineitä, joita käytetään Windows- sovelluksissa. Aluperin Microsoft julkaisi tämän vuonna 1992, mutta myi sen Rational Softwarelle vuonna 1996 (Automationjunkies, 2007). Ohjelmiston kehittäjät ja testaajat voivat tehdä Rational Visual Test –sovelluksen avulla testauksesta tuottavaa ja tehokasta (VersionTracker, 2007).

Tällainen työväline sopii hyvin regressiotestakseen, jos järjestelmä on vakaa eikä siihen tule paljon muutoksia. Graafisen käyttöliittymän automaatiovälineet pystyvät tunnistamaan nopeasti käyttöliittymän eri kohteet. Tämä työväline voi löytää virheitä, jos järjestelmä ei ole riittävän kypsällä tasolla. (Stenberg 2006, s. 53)

Ongelmana tällaisen työvälineen käytössä ovat esimerkiksi seuraavat asiat:

- Yksi järjestelmään tehty muutos voi vaikuttaa monen komentojonotiedoston (skriptin) muuttamiseen ylläpitovaiheessa.
- Tämä työväline vaatii usein ohjelmointia nauhoituksen jälkeen.
- Komentojonotiedostojen (skriptien) luominen on helppoa eli niitä voi tehdä liikaa.
- Graafisen käyttöliittymän automaatiovälineillä ei voi tehdä kovin monimutkaisia testejä. (Stenberg 2006, s. 54)

6. Ohjelmistotestauksen kehityssuunnitelma

Tässä kappaleessa käyn läpi, kuinka Wipron testausta kehitettiin ja tehtiin testauksen kehityssuunnitelma, jonka avulla voidaan suorittaa testitapaukset tehokkaasti Wiprossa nyt ja tulevaisuudessa. Seuraavassa luvussa 7 käydään läpi ja testataan testauksen kehityssuunnitelman eri menetelmiä. Kehitysmenetelmien onnistumista selvitetään kahden testaajan haastatteluilla ja vanhojen ja uusien menetelmien hyvyyden vertailuilla.

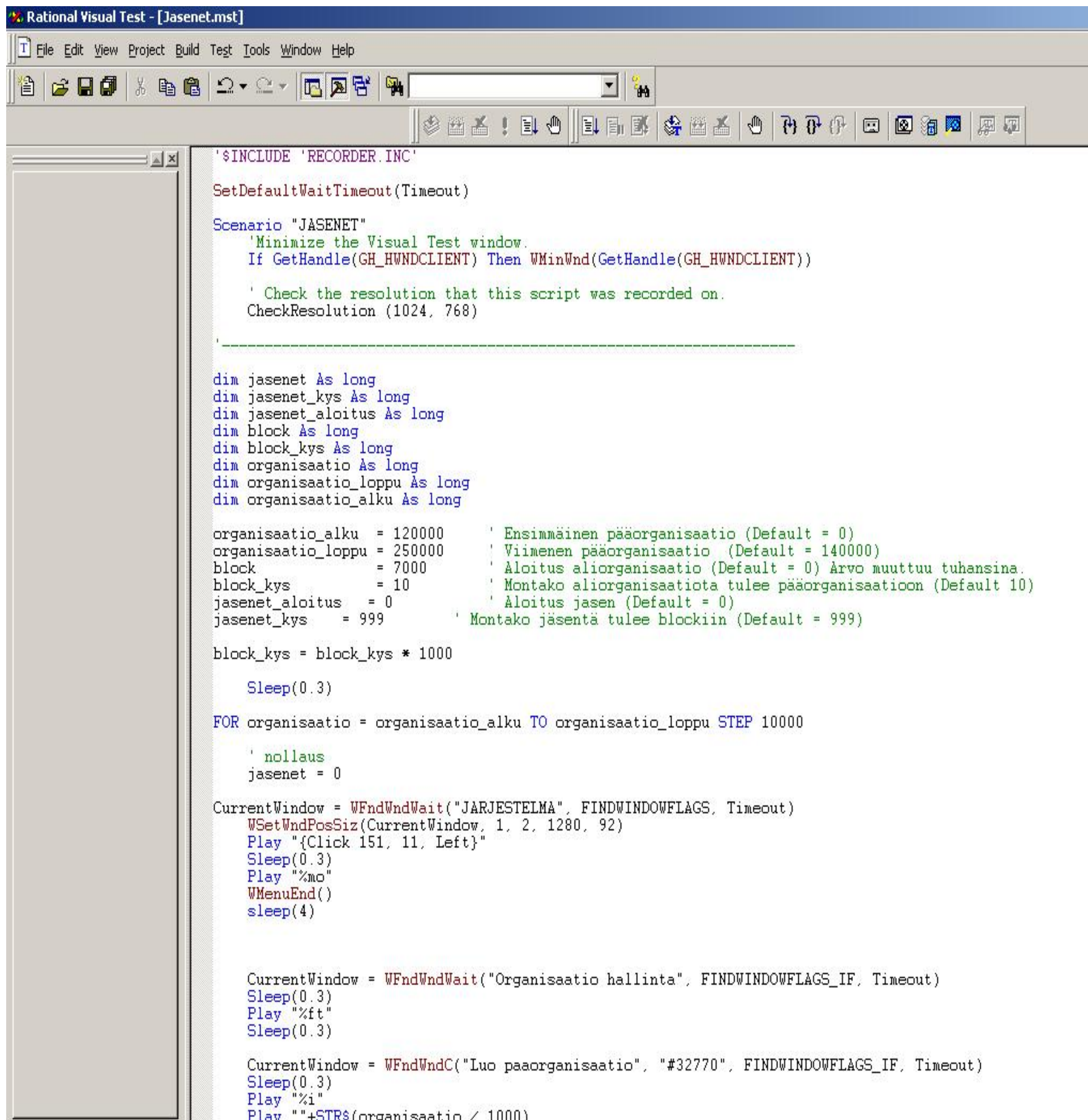
Kehityssuunnitelma on jaettu kolmeen eri kohtaan, joita tullaan kehittämään testauksessa. Ensimmäisenä kohtana on testaukseen liittyvien testaussuunnitelmien ja testitapauksien kehittäminen ja katselmoinnin kehitysideat. Toisena kehityskohteena on testaustulosten ja testimakrojen tallentamiseen liittyvät asiat ja keinot. Kolmantena kehittämiskohteena on uuden työvälineen käyttöönotto ja sen hyödyntäminen testauksessa ja testausympäristön luonnissa.

6.1. Rational Visual Test –työväline käyttöönotto

Kehityssuunnitelmassa otetaan käyttöön uusi Rational Visual Test –työväline, jonka avulla luodaan paljon tietoa testausympäristöön kuormitustestaukseen Windows-ympäristössä. Esimerkiksi Rational Visual Test -työvälineen makrolla (Liite 1 ja kuva 6) luodaan paljon jäseniä testausympäristöön. Jäseniä tarvitaan testitapausten tekemiseen. Liite 1:ssa on kuvattuna koko jäsenien luontimakro, jonka loin näitä testejä ja testausrutiinien nopeutamista varten. Kuvassa 6 on alkunäkymä jäsenien luontimakrosta. Samat tiedot voidaan lisätä myös manaalisesti testausympäristöön, mutta tämä keino on hidasta ja työlästä isojen määrien lisäyksessä. Tämän työvälineen avulla voidaan nopeuttaa testausta ja säästynyt aika voidaan käyttää muihin testausrutiineihin. Lisäksi testien määrää ja kattavuutta voidaan lisätä.

Aluksi tällä ohjelmalla nauhoitetaan makropohja jäsenien luontia varten. Nauhoitus sisältää kaikki tietokoneen hiiren liikkeet ja klikkaukset sekä näppäimistön painallukset. Makrotiedostoston pohjassa käydään läpi yhden jäsenen kaikki tilanteet ja tiedot kohta kohdalta. Ohjelma nauhoittaa kaikki jäsenen luontia varten tarvittavat tiedot eli luodaan ensiksi raamit itse testimakrolle. Seuraavaksi lisätään itse manuaalisesti makroon jäsenten määrä, aloituskohta eli ensimmäisen jäsenen numero ja niiden sijoituspaikka testausympäristössä (organisaatio) sekä muut vaadittavat

loogiset toiminnot. Lopuksi tallennetaan jäsenien luontimakro uudestaan mahdollisesti uudella nimellä. Makro on nyt valmis ajettavaksi RUN -komennolla.

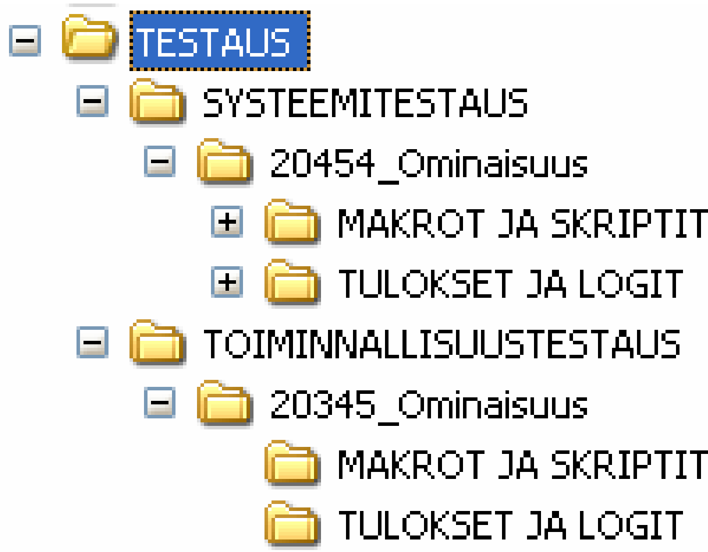


Kuva 6. Näkymä jäsenien luontimakrosta.

6.2. Testien tulosten tallentaminen

Testitapausten tulokset ja lokit sekä niissä käytetyt makrot ja komentojonotiedostot (skriptit) tallennetaan samalle palvelimelle, jota testaajat ja muut ohjelmistokehittäjät käyttävät. Ne sijoitetaan selvästi tarkoituksenmukaisesti nimettyihin alihakemistoihin. Toiminnalisuustestien tulokset sijoitetaan omaan hakemistoon ominaisuuden mukaisella numerolla varustettuna esimerkiksi ”20345_ominaisuuden nimi” ja järjestelmätestien tulokset omiin hakemistoihin

ominaisuuden mukaisella numerolla varustettuna esimerkiksi ”50345_ominaisuuden nimi”. Näiden kahden ominaisuus-hakemistojen alapuolelle tulee vielä eri alihakemistot testeissä käytetyille komentojonotiedostoille (skripteille) ja makroille sekä testien tuloksille oma alihakemisto. Tällaisen rakenteen (Kuva 7) avulla on helppo löytää testien tulokset ja mahdollisesti uudelleen käytettävät skriptit ja makrot, joita käytetään regressiotestauksessa.



Kuva 7. Tulosten, lokien, makrojen ja skriptien sijoittaminen verkkolevylle.

Kaikkien testaajien ja koodaajien pitää tallentaa tuotoksensa ja testien tulokset yhteiselle verkkolevylle, jossa jokaiselle tiedostolle on määrätty oma paikkansa ja eri tiedostojen pitää myös olla omalla paikallaan. Hakemistorakenne ja nimeämiskäytännöt tuovat selvyyttä ja näkyvyyttä testien tulosten sisällöstä. Kaikkien pitää yhdessä päättää eri tiedostojen nimeämiskäytännöt ja tallennuspaikat. Tämä auttaa luonnollisesti eri testitiedostojen uudelleenkäyttöä ja antaa tukea myös regressiotestaukseen. Tämä on senkin takia tärkeää, että testituloksia, komentojonotiedostoja (skriptejä) ja makroja voidaan helposti ja nopeasti löytää ja käyttää tietyistä hakemistoista samalta verkkolevyltä.

Kaikki kansioiden käyttäjät päivittävät aina hakemistojen tietoja, jos tulee jotain uutta ja muutettavaa tietoa sekä lisäävät uusia lokeja johonkin hakemiston kansioon. Verkkolevyn jokaiselle hakemistolle valitaan vastuuhenkilö tai henkilöt, jotka sitten päivittävät muuttuvia tietoja ja ylläpitävät näitä hakemistoja. Eri testaushakemistoille ylläpitämiseen valitaan testaushenkilöt ylläpitämään muuttuvia tietoja. Esimerkiksi se testaaja, joka testaa lähes koko ajan kuormitustestauksia, voi päivittää makro ja skripti -tietoja kuormitushakemistossa.

Tässä testien tallennuksen kehityskohteessa suunnittelijat ja testaajat eivät enää omi töitänsä ja tuloksiansa itselleen omiin hakemistoihin, vaan kaikki todellakin tallennetaan samaan paikkaan omiin nimellä määriteltyihin hakemistoihin ja alihakemistoihin ja kaikki noudattavat yhteisiä pelisääntöjä. Näin voidaan välttää mahdolliset päällekkäiset testit, skriptit ja makrot. Lisäksi eri tiedostojen ylläpito helpottuu.

6.3. Testitapausten kirjoittaminen ja katselmoinnit

Kirjoitettaessa testitapauksia toiminnallisuustestitapaukset tehdään tarkasti. Tällöin määritellään tarkasti testattavan ominaisuuden testiympäristö ja siinä käytettävät laitteet sekä itse testi käydään tarkasti läpi kohta kohdalta. Jokainen kohta sisältää mahdolliset testikomennot ja mitä komentojonotiedostoja (skriptejä) pitää käyttää testissä.

Taas saman ominaisuuden järjestelmätason testitapaukset tehdään paljon ylimalkaisemmin ja lähinnä ranskalaisin viivoin kuvaamalla vain mitä tehdään sekä mahdollisen testitapauksen tuloksen. Tässä testausvaiheessa keskitytään ja käytetään aikaa itse ongelmatapauksiin ja mahdollisiin testauksen kriittisiin kohtiin, jolloin saadaan paremmin viat pois testattavasta ominaisuudesta ja pystytään löytämään paremmin testitapauksia, joilla voidaan niin sanotusti rikkoa testattavaa järjestelmää sekä testauskattavuus on paljon parempi, koska kuitenkin testaussuunnitelmien ja testitapausten tekemiseen sekä suunnitteluun on määritelty ja suunniteltu rajallisesti aikaa.

Ennen varsinaista katselmontia järjestetään testaajien kesken oma niin sanottu esikatselmointi, jossa käydään testattavaa ominaisuutta läpi ja mietitään mahdollisia puuttuvia testattavia vikatilanteita testaajien näkökulmasta. Lisäksi tässä esikatselmoinnissa korjataan myös kirjoitusvirheet ja erilaiset muotoseikat. Esikatselmoinnin jälkeen muokataan testaussuunnitelmaa ja testitapauksia korjausten mukaisesti. Sitten 2-3 päivän päästä pidetään varsinainen katselmointi, jossa on mukana myös katselmoitavan ominaisuuden toteuttavat henkilöt eli koodaajat, arkkitehtuurisuunnitelmien tekijä, ominaisuudesta vastaava, projektin vetäjä sekä ominaisuuden testaajat. Tämän parannuskeinon avulla jää vähemmän korjattavaa katselmointiin ja voidaan keskittyä pelkästään ominaisuuden testaamiseen liittyviin asioihin ja ongelmatilanteisiin. Lisäksi katselmoinnissa saadaan ohjelmistoprosessin eri tekijöiden näkökulma mahdollisiin vikatilanteisiin ja ominaisuuden testattavuuteen.

7. Mittaukset ja arvioinnit

Tässä kappaleessa on määritelty joitain testejä ja testauksen liittyviä toimintoja, jotka tehtiin vanhalla tavalla ja uuden kehityssuunnitelman mukaisesti ja testien avulla tehtiin mittaukset ja arvioinnit. Lisäksi verrattiin vanhoja ja uusia menetelmiä keskenään ja tarkasteltiin uuden suunnitelman mahdollista parantavaa vaikutusta ja nopeutta testauksessa. Kehittämiskohteiden arvioit, testaukset sekä vertailut suorittivat kaksi yrityksen testaajaa ja minä itse.

Kaikilla näillä kehitysideoilla testauksessa pyrin todistamaan, että pienilläkin muutoksilla voitiin saada paljon aikasäästöä testausprosessissa ja nopeuttaa sitä. Näillä parannuksilla olisi tarkoitus aloittaa yrityksen testauksen kehitystyö ja jatkaa sitä vähitellen etsien uusia kehityskohteita testauksen kehittämiselle.

7.1. Työvälineen käyttö

Uuden työvälineen käyttö ja sen tehokkuus testattiin siten, että uudella Rational Visual Test –ohjelmalla luotiin aluksi jäseniä 200 kappaletta testejä varten. Tämä sovellus käytti liitteessä A:ssä olevaa valmista jäsenien luontimakroa, jonka olen tehnyt tätä testiä varten. Seuraavaksi kaksi testausinsinööriä loi 20 kappaletta jäseniä manuaalisesti lisäten yhden jäsenen kerrallaan. 20 jäsenen luomiseen kulunut aika kerrottiin 10:llä, niin saatiin noin 200 jäsenen tekemiseen kulunut aika. Näin voitiin verrata keskenään uuden käyttöönotettavan automaattisen työvälineen ja testaajien käyttämää aikaa jäsenien manuaalisesti luomisessa.

Yhden jäsenen tekemiseen testausympäristöön kesti tällä uudella työvälineellä noin kaksi sekuntia, kun taas normaalisti testaajalta jäsenen tekemiseen meni aikaa noin 50 - 53 sekuntia jäsenen perusominaisuuksilla. Jos laitetaan lisäominaisuuksia jäsenelle, voi yhden jäsenen tekeminen kestää yli 1.5 minuuttiakin. Makron ajaminen uudella työvälineellä kesti noin 400 sekuntia eli 6 minuuttia ja 40 sekuntia. Testaaja 1:llä meni aikaa 20 jäsenen tekemiseen 1011 sekunttia eli 16 minuuttia 51 sekuntia ja taas testaaja 2 teki manuaalisesti jäsenet ajassa 998 sekuntia eli 16 minuuttia 38 sekuntia. Nuo arvot kerrottiin 10, niin saadaan 200 jäsenen tekemiseen manuaalisesti kulunut aika. Testaajilta kului aikaa 200 jäsenen tekemiseen 2 tuntia 48 minuuttia 30 sekuntia ja 2 tuntia 46 minuuttia 20 sekuntia. Kahden sadan jäsenen lisääminen kestää käsin manuaalisesti testaajalta yli kolmasosan työpäivästä (7,5 tuntia) eli yli 2,5 tuntia. Vertailutulokset löytyvät taulukosta 1.

Tekijä	Luonnin kesto 20 jäsentä	200 jäsentä
Rational Visual Test -sovellus	-	6 min. 40 sek.
Testaaja 1	16 min. 51 sek.	168 min. 30 sek
Testaaja 2	16 min. 38 sek.	166 min. 20 sek.

Taulukko 1. Automaatiotyövälineen ja manuaalisen tekemisen ero ajallisesti

Tietojen luominen testausympäristöön vie aikaa ja huomiota muulta testaukseen liittyvältä. Aikasäästö on suuri ja lähes puolityöpäivän aikana voi tehdä kaikkea muuta testaukseen liittyvää: Esimerkiksi ehtii tekemään muut testausympäristön vaatimat asiat ja testaamaan joitakin testitapauksia.

Lopuksi haastattelin testaajia ja kyselin heidän mielipiteitä työvälineen tehokkuudesta ja sen hyödyllisyydestä. Haastattelut kestivät yhteensä noin kymmenen minuuttia. Haastatteluita ei nauhoitettu, vaan ainoastaan kirjoitin muistiinpanot paperille testaajien haastatteluista. Toinen testaaja oli ollut töissä Wiprolla vajaan vuoden ja taas toinen 7 vuotta. Toisen testaajan työkokemusvuosista huolimatta haastatteluiden vastaukset olivat samanlaisia.

Testaajat olivat samaa mieltä kanssani sovelluksen hyödyllisyydestä testausympäristön luonnissa. Heidän vastauksistaan ilmeni, että eri testausympäristön vaatimia tekijöitä ja asioita voidaan luoda nopeasti ja helposti uuden työvälineen makrojen avulla ja työvälineen käytöllä säästynyt aika voidaan käyttää muuhun testaukseen liittyviin asioihin. Testaaja 1 kertoi testin jälkeen, että kädet väsyivät jäseniä luodessa ja sellaisia ongelmia ei tule työvälineen tehdessä. Hän totesi myös sovelluksen tekevän testausympäristön vaatimia asioita nopeasti ilman mitään virheitä. Testaaja 1 olisi käyttänyt työvälineen avulla säästyneen ajan esimerkiksi testausympäristön parametrien asettamiseen ja muihin testauksen esivalmisteluihin.Testaaja 2 oli sitä mieltä, että uusi työväline teki nopeasti jäsenet testausympäristöön ja tästä työvälineestä tulee olemaan vielä hyötyä testauksessa. Hän olisi käyttänyt työvälineen avulla säästyneen ajan testitapausten tulostamiseen, uusien testaussuunnitelmien tekemiseen ja testausympäristön valmisteluun.

7.2 Testausten tulokset ja lokit

Testitapausten tuloksien ja lokien sekä niissä käytetyt makrojen ja komentojonojen (skriptien) tallennuksessa samalle palvelimelle käytettiin yhtä toiminnallisuustestitapausta.

Normaalisti vanhalla tavalla tehden lokit, tulokset ja siinä käytetyt makrot ja komentojonotiedostot (skriptit) jäävät helposti vain tietokoneen Temp-hakemistoon tai johonkin omaan hakemistoon eri verkkolevyille piiloon muilta testaajilta. Esimerkiksi regressiotestauksessa, kun tarvittaisiin uudelleen samoja makroja tai komentojonotiedostoja (skriptejä), ei niitä löydy mistään tai ne löytyvät monen tunnin etsimisen jälkeen. Myös uusien ja vanhojen tuloksien vertailuissa on tärkeää löytää lokit ja tulokset nopeasti.

Aluksi testaajat tekevät testien tulosten, makrojen ja komentojonotiedostojen (skriptien) tallennuksen vanhalla totutulla tavalla. Tiedostot tallennettiin testi-nimellä eli esimerkiksi tulokset tallennettiin testi1 ja makro-tiedosto makro1. Testaaja 1 teki testitapauksen ja tallensi tulokset sekä makrot ja komentojonotiedostot (skriptit) omaan määrittelemäänsä hakemistoonsa vanhojen tapojen mukaisesti. Sen jälkeen etsin niitä tiedostoja verkkolevyltä. Löysin testitapauksen tulokset, makrot ja komentojonotiedostot (skriptit) testaaja 1:n omalta hakemistosta muiden tiedostojen seasta. Oli aika hankalaa löytää toisen testaajan testaamaa testitapauksen tuotoksia muiden joukosta, kun ei ollut mitään selvää sääntöä niiden tallentamisesta. Testaaja 2:n tuotokset löytyivät helpommin testaajan omasta hakemistosta uudesta luodusta TESTI-hakemistosta, mutta siellä ei ollut lainkaan testissä käytettyjä makroja ja komentojonotiedostoja (skriptejä). Ne löytyivätkin tietokoneen TEMP-hakemistosta. Testaajat etsivät myös toistensa tuotoksia ja myös heillä tiedostojen löytäminen kesti kauan.

Uuden kehityssuunnitelman mukaisesti testaajat tallensivat tulokset ja lokit etukäteen annettujen ohjeiden mukaan (kuva 8). Molemmat testaajat testasivat testitapauksen ja lopuksi testissä syntyneet tulokset ja testissä käytetyt makrot ja komentojonotiedostot (skriptit) tallennettiin verkkolevylle ohjeiden mukaan nimettyihin hakemistoihin.

Annettujen ohjeiden mukaisesti kahden koetestaajan piti tallentaa testauksen ajamisen jälkeen tulokset ja lokit omaan alihakemistoon sekä testissä käytetyt makrot ja komentojonotiedostot (skriptit) myös omaan alihakemistoon. Alihakemistot sijaitsivat testaajien verkkolevyllä ja siellä TESTAUS-hakemistossa. Testaajien hakemistojen nimeämiskäytäntö tässä testissä oli seuraavanlainen:

Testaaja 1: TESTI_1-hakemisto ja alihakemistot **TULOKSET_1** ja **MAKROT_SKRIPTIT_1.**

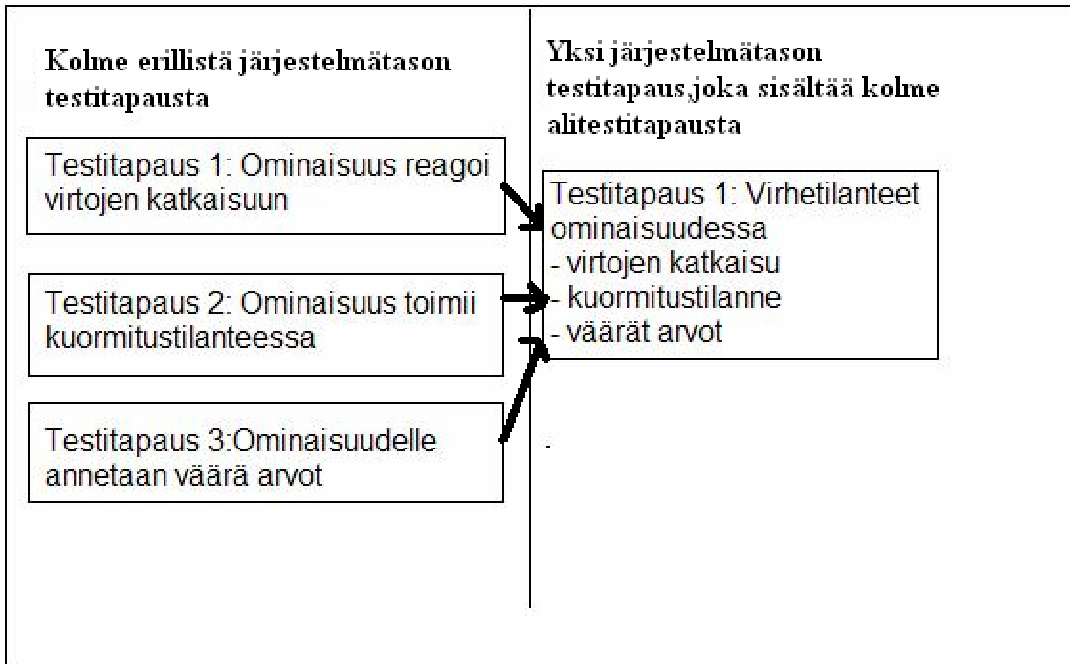
Testaaja 2: TESTI_2-hakemisto ja alihakemistot **TULOKSET_2** ja **MAKROT_SKRIPTIT_2**

Etsin uudelleen testitapauksien tuloksia, makroja ja komentojonotiedostoja (skriptejä) testaajien testitapausten ajamisen jälkeen. Nyt nämä tiedostot löytyivätkin helposti ja oikeista hakemistoista ennalta annettujen ohjeiden mukaisesti. Näiden yhteisten pelisääntöjen noudattaminen helpotti testausten tuotosten löytymistä. Myös testaajat etsivät toistensa testitapauksen tuloksia ja testissä käytettyjä välineitä. He löysivät myös testien tulokset nopeasti.

Haastattelin testien jälkeen testaajia ohjeiden toimivuudesta ja niiden mahdollisesta käyttöönotosta. Haastattelut kestivät noin viisi minuuttia ja vastaukset kirjoitin ylös paperille. Testaaja 1:n mielestä tuotokset oli helppo löytää monien tiedostojen ja hakemistojen joukosta sekä niiden etsimiseen kulunut aikasäästö oli merkittävä verrattuna vanhoihin tallennustapoihin. Testaaja 1 piti myös ohjeita hyödyllisenä ja hänen mielestä tällä tavalla olisi pitänyt aikaisemminkin menetellä testauksen tuotosten tallentamisessa. Testaaja 2 piti tätä hyvänä keinona löytää tulokset ja tiedot nopeasti oikeasta hakemistosta ja testeissä käytetyt makrot ja komentojonotiedostot (skriptit) löytyivät nopeasti testitapauksen uudelleentestausta varten. Hän mainitsi myös tämän keinon nopeuttavan itse testaustilanteissa toimimista, koska tulokset ja testaukseen liittyvät tiedostot oli helppo löytää kiireellisissä testaustilanteissa.

7.3. Testitapausten kirjoittaminen

Tässä kappaleessa on tarkoituksena tehdä järjestelmätason testitapauksia uudella tavalla ja arvioda niiden paremmuutta vanhaan tekemiseen. Uudella tavalla testitapaukset kuvataan ranskalaisin viivoin. Uuden ja vanhan tavan ero voidaan nähdä kuvassa 8. Kuvan vasemmalla puolella on vanhalla tavalla tehdyt järjestelmätason testitapaukset ja kuvan oikealla puolella ovat uudella tavalla tehdyt testitapaukset. Uudessa testitapauksessa on yhdistetty kolme testitapausta yhdeksi testiksi. Testasin itse tämän uuden ja vanhan menetelmän ja haastattelin kahta testaaja menetelmien testauksien jälkeen.



Kuva 8. Järjestelmätason testitapausten tekemisen ero vanhalla (vas.) ja uudella (oik.) tavalla.

Järjestelmätason testitapaukset tein ensiksi normaalin vanhan kaavan mukaisesti, jossa yhteen testitapaukseen lisäsin kaikki testitapauksessa tarvittavat yksityiskohdat. Testitapaus sisälsi testin etenemisen kohta kohdalta, suoritettavat toiminnot ja odotetut lopputulokset. Tein seuraavaksi samat testitapaukset yhdistämällä yhteen testitapaukseen. Testitapaukset selitettiin ranskalaisilla viivoilla. Yhdellä ranskalaisella viivalla (1-2 riviä tekstiä) selitettiin hyvin yleisesti, mitä testissä tehdään ja mikä on testin mahdollinen lopputulos. Yhdessä järjestelmätason testitapauksessa voi olla useita ranskalaisia viivoja, jotka testaavat samanlaisia asioita.

Kun testitapaukset tehtiin tällä uudella tavalla, testien suunnittelussa jäi paljon aikaa keskittyä ongelmatilanteisiin. Uudessa menetelmässä selitetään ainoastaan testi lyhyesti, testin lopputulos ja yhteen testitapaukseen tulee useita samantyylisiä testejä. Uudella tavalla tehtynä aikaa kului yhden järjestelmätason testitapauksen tekemiseen 11 minuuttia. Testitapaus sisälsi kolme eri testiä ranskalaisilla viivoilla tehtynä. Vanhalla tavalla tehtynä kolmen erillisen testitapauksen tekeminen kesti noin 77 minuuttia. Tällä uudella tavalla voidaan käyttää yli tunti enemmän aikaa ongelmakohtien miettimiseen.

Menetelmien testauksien jälkeen haastattelin kahta Wipron testaajaa. Kysyin heidän mielipiteitä uuden menetelmän toimivuudesta. Testaajien haastattelu kesti noin kymmenen minuuttia. Haastattelun tulokset kirjasin paperille. He eivät olleet seuranneet testien tekemistä, mutta

molemmat testaajat vertailivat uudella ja vanhalla tavalla tehtyjä testitapauksia haastatteluiden aikana. Molemmat testaajat olivat samaa mieltä siitä, että uudella tavalla tehtynä voi jäädä enemmän aikaa ongelmakohtien etsimiseen. Testaaja 1 oli sitä mieltä, että mahdolliset viat löytyvät paremmin jo testauksessa ennen kuin sovellukset ja ohjelmistot viedään asiakkaalle. Testaaja 2 piti tätä uutta keinoa hyvänä, jos vain toiminnallisuustestitapaukset tehdään perusteellisesti yksityiskohdat huomioon ottaen. Testaaja 2 mainitsi myös, että testauksen suunnittelu voi tehostua ja myös uudella tavalla voidaan luoda kattavia testitapauksia.

7.4. Esikatselmointi

Esikatselmoinnissa ideana oli löytää kirjoitusvirheet ja muut pienet muotovirheet ennen varsinaista testaussuunnitelman ja testitapausten katselmointia. Itse katselmoinnissa käydään läpi testattavan ominaisuuden ongelmakohtia, vaikeita tapauksia ja ohjelmiston toipumiskykyä virhetilanteista.

Tämän esikatselmointikohdan testasin itse omalla vastuulla olevalla ominaisuudella. Tein ensiksi testaussuunnitelman valmiiksi. Sen jälkeen järjestin esikatselmointitilaisuuden testaajien kesken. Tilaisuudessa testaajat oppivat samalla uudesta ominaisuudesta ja kommentoivat samalla virheistä ja puutteista, joita suunnitelmasta löytyi. He myös ehdottivat joitain lisätestitapauksia, jotka voisivat olla hyödyllisiä ominaisuuden testauksessa. Esikatselmointi kesti noin 40 minuuttia.

Seuraavaksi järjestettiin varsinainen katselmointi kolmen päivän päästä, johon osallistui myös muita ominaisuuden tekemiseen liittyviä henkilöitä. Osallistujat olivat etukäteen tutustuneet katselmoitavaan materiaaliin. Katselmointi suoritettiin paljon nopeampaa kuin yleensä ja lähinnä siellä käytiin läpi vain ominaisuuden mahdollisia ongelmakohtia ja kuormitustilanteita sekä lisäksi ehdotettiin joitain kriittisiä tilanteita ominaisuudessa testauksessa. Normaalisti ominaisuuden katselmointi on kestänyt 2-3 tuntia, mutta nyt selvittiin 45 minuutissa ja sekin aika käytettiin hyödyllisesti ongelmakohtia miettien.

Esikatselmoinnin haastattelut tein kahdelle testaajalle, jotka olivat mukana esikatselmoinnissa. Kysyin heidän mielipiteitä esikatselmoinnin toimivuudesta. Haastattelut tein nopeasti töiden ohessa ja testaajien kommentit kirjoitin paperille. Testaajien haastattelut kestivät noin kymmenen minuuttia. Testaaja 1 oli sitä mieltä, että tällä uudella katselmointimenetelmällä voidaan keskittyä paremmin testattavan ominaisuuden ongelmatilanteisiin itse katselmoinnissa ja samalla kirjoitusvirheiden korjaaminen siirtyi esikatselmointiin. Testaaja 2 mainitsi haastattelussa, että

esikatselmoinnissa oppi tuntemaan uuden ominaisuuden ja puutteelliset testitapaukset saatiin korjattua. Hän tarkoitti puutteellisella testitapauksella sitä, että testaus ei ollut tarpeeksi kattava ja testitapaukset eivät sisältäneet kaikkea informaatiota, mitä testien pitäisi sisältää. Testaaja 2 mainitsi myös, että nyt katselmointi on todella hyödyllinen keino löytää mahdollisia vikatilanteita. Myös oman kokemukseni mukaan katselmoinnit pitäisi aina järjestää tällä tavalla.

8. Yhteenveto

Tutkimustyössäni perehdyin ohjelmistotestauksen menetelmiin ja testausprosessin liittyviin toimintoihin ja kuinka niitä voitaisiin kehittää yrityksessäni Wiprossa. Tutkielman tavoitteena oli kartoittaa kirjallisuuden, omien kokemuksien ja tutkimuksen pohjalta Wipron testauksen kehittämistä ja sen automatisointia sekä sen pohjalta luoda uusi testauksen kehityssuunnitelma. Tutkimuksen lähestymistapa oli käytännönläheinen deduktiivinen päättely. Päättely nojasi aikaisempaa teoriaan, jota tutkielmassa käytiin läpi. Päättelyn keinoin verrattiin tehtyä tutkimusta olemassa oleviin teorioihin ja niiden avulla pyrittin luomaan testauksen kehityssuunnitelma.

On erittäin tärkeää nähdä, että pystyy hahmottamaan ohjelmistotestauksen kokonaisvaltaisesti. Testauksen tulisi alkaa jo ennen kuin yksikään ohjelmoija on tehnyt riviäkään koodia tehnyt ja jatkua ohjelmiston elinkaaren loppuun saakka. Testaus on tärkeää myös siksi, että virheet voidaan poistaa mahdollisimman aikaisessa vaiheessa. Virhekustannukset saadaan myös vähenemään testauksen avulla.

Keskityin työssäni vain erilaisten testauksen ja automatisoinnin osa-alueiden sekä valmiin testaustyökalun tutkimiseen, koska kokonaisen testausympäristön toteuttaminen automatisoituna olisi ollut tämän pro gradun laajuuden ja aikataulun asettamissa rajoissa liian suuri projekti. Tässä tutkimuksessani pyrin kehittämään eri testausmenetelmiä, jotta Wipron testaus paranisi ja tehostuisi.

Testauksen automatisointi on työläs prosessi, joka vaatii onnistuakseen koko ohjelmistonkehitysorganisaation määrätietoista toimintaa ja tukea. Jotta automatisoinnista olisi hyötyä yritykselle, testauksen automatisointiin tulisi panostaa huolellisesti. Aloittamalla pienistä osa-alueista voidaan testausprosessin automatisointiastetta kuitenkin vähitellen nostaa, kunnes se on saavuttanut halutun automatisoinnin tason.

Automatisoimalla testausta voidaan saavuttaa useita hyötyjä. Testaukseen käytettyä aikaa ja resursseja pystytään säästämään. Testaus nopeutuu, koska automatisoinnilla voidaan lyhyessä ajassa tehdä enemmän testitapauksia kuin manuaalisesti. Automatisoinnilla projektin läpimenoaika lyhenee. Automatisoinnilla voidaan poistaa inhimillisten virheiden mahdollisuudet testitapauksia suorittaessa. Testauksen tehokkuus paranee automatisoinnin avulla. Kun tehdään testitapauksia manuaalisesti, ei voi tehdä mitään muuta.

Osoitin käytännössä, mitä puutteita Wipron testausprosessissa oli ja kuinka sitä voitiin pienilläkin parannuksilla parantaa ja kehittää. Lisäksi selvitin tutkimuksen avulla, mitä toimenpiteitä Wipron testauksen kehittämiseen kuului. Ohjelmistotuotantoprosessin ja testausprosessin parantaminen on tärkeää parannettaessa ohjelmistotuotteiden laatua sekä projektien kykyä pysyä budjetissa ja aikatauluissa. Tärkein tekijä onnistuneen testauksen läpiviennille on järjestelmällisten ja määrämuotoisten menetelmien käyttö. Lisäksi on tärkeää kerätä tietoa ja mitata testaukseen liittyviä suureita, kuten testaukseen käytettyä aikaa ja löydettyjen virheiden määrää.

Tämän tutkimustyön tulokset voivat auttaa myös muita organisaatioita tehostamaan testausprosessiaan sekä tunnistamaan mahdollisia ongelmia ja kehityskohteita testauksessa. Yrityksen johdon pitäisi ymmärtää, että testauksen kehittäminen tärkeää taloudellisesti ja testauksen pienilläkin muutoksilla voidaan saavuttaa tehokkuutta testausprosessissa. Ohjelmistomarkkinoilla laatu- ja tehokkuusvaatimukset ovat aiempaa suuremmat, joten testausta kehittämällä voidaan päästä näihin vaatimuksiin.

Viiteluettelo

Painetut:

Haikala, Ilkka, Märijärvi, Jukka. 2004. Ohjelmistotuotanto.Talentum, Helsinki.

Myers, Glenford J. 1979. The Art of software Testing. John Wileys & Sons, Yhdysvallat.

Paakki, J. 2000. Ohjelmistojen testaus. Lecture notes, University of Helsinki Department of Computer Science.

Patton, Ron. 2001. Software Testing. Sams Publishing, Yhdysvallat.

Pressman, Roger S. 2000. Software Engineering – A Practioner's Approach. McGraw-Hill Publishing Company, Englanti.

Internet-lähteet:

Automationjunkies (2007). ”Automationjunkies”.
http://www.automationjunkies.com/tools/vt/index.shtml. Haettu 28.2.2007

Bergström, Juhani, Von Etter, Peter, Känsälä, Teppo, Lyytinen, Olli, Penttinen, Jessika, Waris, Mikko (2005). ”Testaussuunnitelma”.
http://www.cs.helsinki.fi/group/orava/docs/testaussuunnitelma.pdf. Haettu 22.11.2006

Enqvist, Antti, Krats, Antti, Lahti, Kai, Luukkonen, Mikko, Maanselkä, Asmo,Yli-Honkola, Juho (2001). ”Ohjelmiston testaus”.
http://www.cc.jyu.fi/~akrats/harkat/ohjtuo/Ohjelmiston_testaus.htm. Haettu 10.11.2006

Harju, Hannu, Koskela, Mika (2003). ”Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi”.VTT tiedote. Otamedia Oy, Espoo.
http://www.vtt.fi/inf/pdf/tiedotteet/2003/T2193.pdf. Haettu 28.5.2007

HETKY (2002). ”Teema – Ohjelmistobisneksen ajankohtaiset toimintamallit”. http://www.hetky.fi/Hetky0302.pdf. Haettu 13.10.2006

Holopainen, Juha (2005).”REGRESSIOTESTAUS JA TESTIEN VALINTATEKNIIKAT”. http://www.uku.fi/tike/his/avointa/julkaisut/Regressiotestaus_ja_testien_valintatekniikat.pdf. Haettu 28.11.2006

Jaakkola, Hannu (2004). “15. Testaus”. http://www.pori.tut.fi/~hj/for-guests/public/ost/15-testaus.PDF. Haettu 1.9.2006

Jot Automation (2001). ”JOT Automation Group vuosikertomus 2000”. http://helecon3.hse.fi/FI/yrityspalvelin/pdf/2000/Felektrobit2000.pdf. Haettu 22.10.2006

Kankaanpää, Timo (1998). “Testaus ohjelmiston kehitysprosessissa”. http://www.cc.puv.fi/~tka/kurssit/Tietojarjestelmien_suunnittelu/Testaus.doc. Haettu 30.8.2006

Karkulehto, Jukka, Urpi, Tuukka (2003).”Testitapausten suunnittelu”. http://www.virtuaaliyliopisto.fi/liikkuvuus/suunnitteluvaihe/LH_Testitapausten_suunnittelu_versio_1.0.doc. Haettu 28.11.2006

Karvonen, Tuomas (2006). ”Testaus korostuu softatuotannossa”. http://www.digitoday.fi/page.php?page_id=9&news_id=200610227. Haettu 11.10.2006

Kautto, Tuomas (1996).”Ohjelmistotestaus ja siinä käytettävät työkalut”. http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus/. Haettu 2.12.2006

Koikkalainen, Pasi (2000). ”Aluetestaus”. http://erin.mit.jyu.fi/pako/kurssit/ot2000/l12/lect12/node29.html. Haettu 1.9.2006

Kokkoniemi, Jouni (2005). ”Ohjelmistojen testauksesta”. http://www.tol.oulu.fi/~jiikoo/Testaus2005/testaus2005_1.pdf. Haettu 30.8.2006

Kollanus, Sami (2004). ”Katselmoinnit”. http://www.cs.jyu.fi/~kolli/JOT04/materiaali/katselmoinnit.pdf. Haettu 30.8.2006

Lahti, Jarmo (2004). “Suomi löytämässä vientiraon testiohjelmista”.
http://www.digitoday.fi/page.php?page_id=9&news_id=200410940. Haettu 13.10.2006

Mäkelä, sini (2000). ”Testaustyökalut”. http://www.cs.helsinki.fi/u/laine/otv/testaustyokalut.pdf. Haettu 27.2.2007

Määttälä, Jukka (2005). “AUTOMATISOITU REGRESSIOTESTAUS SULAUTETUN JÄRJESTELMÄN KEHITYSTYÖSSÄ”. Pro gradu -tutkielma. Jyväskylän yliopisto. Tietotekniikan laitos.
http://rf.chydenius.fi/dokumentit/GraduJukkaMaattala.pdf . Haettu 1.9.2006

Nikkanen, Eero (2003). ”Ohjelmistotestausprosessin kehittäminen”. Insinöörityö. Evtek.
http://pww.evitech.fi/courses/mm2002/insinoorityot/vmp99s/nikkanen/Nikkanen_Eero_Testausprosessin_kehittaminen.pdf. Haettu 20.10.2006

Niksula (1998). ”Testaussuunnitelma”. http://www.soberit.hut.fi/tik-76.115/97-98/palautukset/groups/Burritos/lu/dokumentit/testaussuunnitelma.html. Haettu 21.5.2007

Ovaska, Saila, Räihä, Kari-Jouko (1998). ”Käytettävyystestaus”.
http://www.pcuf.fi/sytyke/lehti/kirj/st19984/13.pdf. Haettu 28.11.2006

Paakki, Sirpa (2004). ”Testauksen historiaa”.
http://www.cs.helsinki.fi/u/kerola/tkhist/k2004/alustukset/testaus/TestHist.pdf. Haettu 4.9.2006

Pohjolainen, Pentti (2003). “Ohjelmiston testauksen automatisointi”. Pro gradu -tutkielma. Kuopion yliopisto. Tietojenkäsittelytieteen laitos.
http://www.cs.uku.fi/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf. Haettu 1.9.2006

Prosessori (1998). ”Testausko vaikeaa?”.
http://www.prosessori.fi/es98/testaus.htm . Haettu 22.10.2006

Pyhäjärvi, Maarit, Pöyhönen, Erkki (2005). "Testaus ohjelmistokehityksen osana". http://www.cs.jyu.fi/~sakkinen/testaus/aineisto/2_TestausOhjelmistokehityksenOsana_v1.ppt. Haettu 1.9.2006

Rajamäki, Jussi (2004). "Ohjelmistojen testaus". http://www.edu.stadia.fi/~luuma/testaus/materiaali/rajamaki-moniste.doc . Haettu 14.2.2007

Rational Visual Test (2007). " Rational Visual Test ". http://www.embeddedstar.com/software/content/r/embedded330.html. Haettu 20.1.2007

Stenberg, Arto (2006). "Testauksen automatisointi". http://www.pori.tut.fi/~stenberg/index_files/automatisointi.pdf. Haettu 18.1.2007

Tekes (2003). "Ohjelmistotuotteet SPIN 2000-2003 –teknologiaohjelma". http://www.tekes.fi/julkaisut/SPIN_loppuraportti.pdf. Haettu 9.9.2006

Tersa, Tiina (2002). "TESTAUSMENETELMIEN KÄYTTÖ SOVELLUKSEN SYSTEEMITESTAUSVAIHEESSA". Pro gradu –tutkielma. Jyväskylän yliopisto. Tietotekniikan laitos. http://www.mit.jyu.fi/progradut/toteutettuja/systeemitestaus/Tiina_Tersa.pdf. Haettu 1.10.2006

Valtonen, Mari-Anna (2000). "JÄRJESTELMÄTESTAUS". Opinnäytetyö. Hämeen Amk. Tietojenkäsittelyn koulutusohjelma. http://trade.hamk.fi/osma/Tekstimateriaalit/Jarjestelmatestaus_MAV.doc. Haettu 22.10.2006

VersionTracker (2007). "Rational Visual Test - 6.5". http://www.versiontracker.com/dyn/moreinfo/win/16790. Haettu 20.1.2006

Virkanen, Hannu (2002). "Ohjelmistojen testaus ja virheenjäljitys". Pro gradu –tutkielma. Kuopion yliopisto. Tietojenkäsittelytieteen laitos. http://www.cs.uku.fi/tutkimus/Teho/graduvirkanen.pdf. Haettu 5.2.2007

Wikipedia (2007) ”Moduuli”.
http://fi.wikipedia.org/wiki/Moduuli_(ohjelmointi). Haettu 21.5.2007

Liitteet

Liite 1

```
'$INCLUDE 'RECORDER.INC'

SetDefaultWaitTimeout(Timeout)

Scenario "JASENET"
	'Minimize the Visual Test window.
	If GetHandle(GH_HWNDCLIENT) Then WMinWnd(GetHandle(GH_HWNDCLIENT))

	' Check the resolution that this script was recorded on.
	CheckResolution (1024, 768)

'-----------------------------------------------------------------

dim jasenet As long
dim jasenet_kys As long
dim jasenet_aloitus As long
dim block As long
dim block_kys As long
dim organisaatio As long
dim organisaatio_loppu As long
dim organisaatio_alku As long

organisaatio_alku  = 120000		' Ensimmäinen pääorganisaatio (Default = 0)
organisaatio_loppu = 250000		' Viimenen pääorganisaatio		(Default = 140000)
block			= 7000		' Aloitus aliorganisaatio (Default = 0)		Arvo muuttuu tuhansina.
block_kys		= 10			' Montako aliorganisaatiota tulee pääorganisaatioon (Default 10)
jasenet_aloitus   = 0			' Aloitus jasen (Default = 0)
jasenet_kys		= 999		' Montako jäsentä tulee blockiin (Default = 999)

block_kys = block_kys * 1000

	Sleep(0.3)

FOR organisaatio = organisaatio_alku TO organisaatio_loppu STEP 10000

	' nollaus
	jasenet = 0

		CurrentWindow = WFndWndWait("JARJESTELMA", FINDWINDOWFLAGS, Timeout)
	WSetWndPosSiz(CurrentWindow, 1, 2, 1280, 92)
	Play "{Click 151, 11, Left}"
	Sleep(0.3)
	Play "%mo"
	WMenuEnd()
	sleep(4)



	CurrentWindow = WFndWndWait("ORGANISAATIO HALLINTA", FINDWINDOWFLAGS_IF, Timeout)
	Sleep(0.3)
	Play "%ft"
	Sleep(0.3)
```

```
    CurrentWindow = WFndWndC("Create Top-Level Organisation Block", "#32770",
FINDWINDOWFLAGS_IF, Timeout)
    Sleep(0.3)
    Play "%i"
    Play ""+STR$(organisaatio / 1000)
    Play "%o"


    IF (organisaatio < 100000) THEN

    Play "TET "+STR$(organisaatio / 10000) ' Aakkosjärjestys top levelissä

    ELSE

    Play "TET"+STR$(organisaatio / 10000)

    END IF

    Play "%r"

    Play "%r"
    Play "{DOWN}"
    'Play "{DOWN}"
    'Play "{DOWN}"
    Sleep(0.3)
    Play "{ENTER}"
    Sleep(0.3)


    Play "{ENTER}"
    Sleep(0.3)
    Play "{ENTER}"
    Sleep(0.3)
    Play "{TAB}{TAB}{TAB}{TAB}{ENTER}"
    Sleep(0.3)
    Play "%vr"

   while block < block_kys

    CurrentWindow = WFndWndWait("ORGANISAATIO HALLINTA", FINDWINDOWFLAGS_IF, Timeout)
    Sleep(0.3)

    Play "{END}"
    Sleep(0.3)
    Play "%fs"
    Sleep(0.3)

    CurrentWindow = WFndWndC("LUO ALIORGANISAATIO ", "#32770", FINDWINDOWFLAGS_IF,
    Timeout)
    Sleep(0.3)
    Play "%i"
    Play ""+STR$(block / 1000)
    Play "%o"
    Play "TET"+STR$(organisaatio + block)

        Play "%r"
    Play "{DOWN}"
    'Play "{DOWN}"
    'Play "{DOWN}"
    Sleep(0.3)
```

```
Play "{ENTER}"
Sleep(0.3)

'Play "{TAB}{TAB}{TAB}{TAB}{RIGHT}"
'Play "%a"
'Play "17221 10010"
'Play "{TAB}"
'Play "17221 10010"
'Play "{TAB}{ENTER}"


Play "{ENTER}"
Sleep(0.3)
Play "{ENTER}"
Sleep(0.3)
Play "{TAB}{TAB}{TAB}{TAB}{ENTER}"
Sleep(0.3)
Play "%({F4})"
Sleep(0.3)


CurrentWindow = WFndWndWait("JARJESTELMA", FINDWINDOWFLAGS, Timeout)
Sleep(0.3)
Play "%mr"
Sleep(0.3)


CurrentWindow = WFndWndWait("JASEN HALLINTA", FINDWINDOWFLAGS_IF, Timeout)
Sleep(0.3)
Play "%fo"
Sleep(0.3)

CurrentWindow = WFndWndC("AVAA ORGANISAATIO ", "#32770", FINDWINDOWFLAGS_IF,
Timeout)
Sleep(0.3)



            Play "{END}"
            Sleep(0.3)
            Play "{RIGHT}"
            Sleep(1)
            Play "{END}"

Sleep(0.3)
Play "{ENTER}"
Sleep(0.3)

CurrentWindow = WFndWndWait("JASEN HALLINTA", FINDWINDOWFLAGS_IF, Timeout)
Sleep(3)
Play "%fr"
Sleep(5)
CurrentWindow = WFndWndC("LUO JASEN", "#32770", FINDWINDOWFLAGS_IF, Timeout)
Sleep(0.3)
Play "%i"
Play ""+STR$(organisaatio + block + jasenet_aloitus)
Play "%m"
Play "MS-"+STR$(organisaatio + block + jasenet_aloitus)
Play "%t"
Sleep(0.3)
```

```
CurrentWindow = WFndWndC("VALITSE ORGANISAATIO", "#32770", FINDWINDOWFLAGS_IF,
Timeout)
Sleep(2)


                Play "{END}"
                Sleep(0.3)
                Play "{RIGHT}"
                Sleep(1)
                Play "{END}"
                Sleep(0.3)

Play "{ENTER}"
Sleep(3)
CurrentWindow = WFndWndC("LUO JASEN", "#32770", FINDWINDOWFLAGS_IF, Timeout)
Sleep(0.3)
Play "{TAB}{TAB}{TAB}{RIGHT}"
Play "{TAB}{DOWN}{DOWN}{DOWN}"
Play "%g"
Sleep(0.3)
Play "{DOWN}{ENTER}"
Sleep(0.3)
Play "{TAB}{TAB}{TAB}{TAB}{TAB}{TAB}{TAB}{TAB}{RIGHT}"
Sleep(0.3)
Play "{TAB}{TAB}{TAB} {TAB}{TAB}{TAB}{TAB}{TAB}"
Play "{ENTER}"
Sleep(0.3)
Play "{ENTER}"
Sleep(0.3)
Play "{TAB}{TAB}{TAB}{TAB}{TAB}{TAB}{TAB}{TAB}"
Sleep(0.3)
Play "{ENTER}"
Sleep(1)
CurrentWindow = WFndWndWait("JASEN HALLINTA", FINDWINDOWFLAGS, Timeout)
Sleep(0.3)
Play "%({F4})"
Sleep(0.3)

CurrentWindow = WFndWndWait("JARJESTELMA", FINDWINDOWFLAGS, Timeout)
Sleep(0.3)
Play "%mr"
Sleep(0.3)

CurrentWindow = WFndWndWait("JASEN HALLINTA", FINDWINDOWFLAGS_IF, Timeout)
Sleep(0.3)
Play "%fo"
Sleep(0.3)

CurrentWindow = WFndWndC("AVAA ORGANISAATIO ", "#32770", FINDWINDOWFLAGS_IF,
Timeout)
Sleep(0.3)

                Play "{END}"
                Sleep(0.3)
                Play "{RIGHT}"
                Sleep(1)
                Play "{END}"
Sleep(0.3)
Play "{ENTER}"
Sleep(0.3)
```

```
        ' jasenet luuppi!!!

                jasenet_aloitus = jasenet_aloitus + 1

                FOR jasenet = jasenet_aloitus TO jasenet_kys STEP 1

                CurrentWindow = WFndWndWait("JASEN HALLINTA", FINDWINDOWFLAGS_IF, Timeout)

                        Play "^(u)"
                        Sleep(0.4)
        CurrentWindow = WFndWndC("LUO JASEN", "#32770", FINDWINDOWFLAGS_IF, Timeout)

                                Play "{TAB}"
                                Play ""+STR$(jasenet + organisaatio + block)
                                Play "{TAB}{TAB}"
                                Play "MMS-"+STR$(jasenet + organisaatio + block)
                                Play "{TAB}"
                                Play "{ENTER}"

                        Sleep(0.05)
                        'Play "{ENTER}"
                        'Play "{TAB}{TAB}{TAB}{TAB}" ' Varmistus jos jasen on jo olemassa
                        'Play "{ENTER}"
                        'Play "{UP}"

                next jasenet

        jasenet_aloitus = 0

        Play "{TAB}{TAB}{TAB}{TAB}{ENTER}"


        ' Sulkee Organisaatio blockin

        CurrentWindow = WFndWndWait("JASEN HALLINTA", FINDWINDOWFLAGS, timeout)
        Sleep(0.3)
        Play "%({F4})"

        Play "%mo"

        block = block +   1000

wend

block = 0

next organisaatio

End Scenario
```