Lyudmila Starovoitenko

Toiminnallisuus- ja regressiotestauksen organisointi

Metropolia Ammattikorkeakoulu
Insinööri AMK
Tietotekniikka
Insinöörityö
7.3.2011

Metropolia

Tiivistelmä

Tekijä Otsikko Sivumäärä Aika	Lyudmila Starovoitenko Toiminnallisuus- ja regressiotestauksen organisointi 42 sivua + 2 liitettä 7.3.2011
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaajat	testaustiimin vetäjä Terhi Pirttimäki yliopettaja Auvo Häkkinen

Tässä insinöörityössä tutkittiin manuaalisen ja automatisoidun toiminnallisuus- ja regressiotestauksen organisointia. Insinöörityö pohjautuu kahdeksan kuukauden työkokemukseen toiminnallisuus- ja regressiotestauksen suunnittelussa ja suorituksessa isossa projektissa. Sen lisäksi insinöörityössä analysoitiin testaukseen liittyvää kirjallisuutta, jonka avulla saatiin tietoa automatisoinnin ongelmista ja viimeisen sukupolven työkaluista.

Insinöörityössä selvitettiin vaatimukset testausprosessin kypsyydelle, jotta automatisoinnin käyttöönotto onnistuisi. Testauksen automatisoinnin kehityshistoria esiteltiin. Testausprosessin muutttaminen mallipohjaisten automatisointityökalujen käytön seurauksena kuvattiin erikseen.

Insinöörityön päätteeksi saatiin analysoitua manuaalisen ja automatisoidun organisointitapojen hyödyt ja puutteet. Insinöörityössä todettiin, ettei tällä hetkellä ole työkalua, jonka avulla pystyy automatisoimaan kokonaan toiminnallisuus- ja regressiotestausta. Automatisointityökalut vaativat paljon manuaalista työtä skriptien luomisessa, niiden suorituksen tulosten analysoinnissa ja ylläpidossa. Testauksen johdon on päätettävä jokaisen projektin ja toteutettavan sovelluksen kohdalla automatisoinnin aste. Testausprosessin kypsyyden kasvaessa automatisointia voi lisätä.

Avainsanat	regressiotestaus, toiminnallisuustestaus, testauksen organisointi, automatisoitu testauksen suunnittelu, manuaalinen testaus

Metropolia

Abstract

Author Title Number of Pages Date	Lyudmila Starovoitenko Functional and regression test organization 42 pages +2 appendices 7 March 2011
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructors	Terhi Pirttimäki, Test Team Lead Auvo Häkkinen, Principal lecturer

The purpose of this graduate study was to analyze manual and automated approaches to organise functional and regression testing. The study is based on eight months of working experience in a large software development project executing and designing functional and regression testing.

The set of the requirements for the testing process maturity is gathered for testing team to successfully automate functional and regression testing. Problems with automated testing tools were studied by analysing literature regarding modern testing. The last generation automation tools are described through a review of the Conformiq Tool Suite™ testing tool.

The study suggests a partial automation of the testing process. The level of the automation should be chosen by testing management based on the team's experience in automation and the suitability of the application for automation. The testing automation tools depending on their complexity may demand a large amount of the manual work in writing test scripts or creating an abstract model of the application, analyzing results of the automated runs and updating test scripts. There is still no such a tool which automates everything so the test management should be prepared to estimate the effort needed in both automation and manual testing and tasks that follows the decision to automate testing.

Keywords	regression testing, functional testing, automated test design, test process organization, manual testing

Sisällys

1 Johdanto

Tämän insinöörityön aiheena on toiminnallisuus- ja regressiotestaus. Toiminnallisuus-testaus on testauksen tyyppi, jossa testataan tuotteen ominaisuuksia ja toimintoja toiminnallisia määrittelyjä ja vaatimuksia vasten [Hewlett-Packard 2007: 1]. Toiminnallisuustestauksen avulla löydetään yleensä suurin osa virheistä, joita ei tunnisteta moduuli- ja integraatiotestauksen aikana. Nämä virheet ovat poikkeuksia tuotteen toiminnallisista määrittelyistä ja / tai tuotteen toimintaa häiritseviä tai estäviä vikoja.

Regressiotestaus on testauksen vaihe, jossa tarkistetaan sovelluksen eheyttä tehtyjen muutosten jälkeen. Regressiotestausta suoritetaan koko sovellukselle, kun tuotteeseen tuodaan uusia ominaisuuksia tai moduuleja. Regressiotestauksen tarkoituksena on löytää virheet, jotka syntyvät vanhoissa järjestelmätestauksen läpäisseissä tuotteen osissa. [Holopainen 2005: 9.]

Testaus toimii keskeisenä prosessina ohjelmistotuotannon laadunvalvonnassa. Testauksen strategia, organisointitavat ja työkalut vaikuttavat radikaalisesti sekä testauksen onnistumiseen että koko tuotteen laatuun. Projektin budjetti ja aikataulu rajoittavat ja määrittelevät testauksen laajuuden. Toisaalta asiakas voi vaatia testauksen ja projektin jatkamista tietyn tason virheiden eliminointiin asti. Tässä tapauksessa projektin aikataulu riippuu suoraan muun muassa testauksen tehokkuudesta. Tärkeintä on löytää vakavat virheet mahdollisimman varhaisissa testauksen tai laadunvalvonnan katselmointivaiheissa. Virheiden aiheuttamat kulut kasvavat nopeasti myöhemmissä vaiheissa [Cauldwell 2008: 57].

Testauksen tuottama lisäarvo on testauksesta tuotteesta saatu tieto, joka voidaan käyttää tuotteen kehittämisessä, tuotetta koskevassa päätöksenteossa sekä testaus- ja kehitysprosessien parantamisessa. Projektinhallinnan kannalta tärkeitä tietoja ovat esimerkiksi onnistuneesti ajettujen testitapausten määrä, testauksen kattavuus, virheiden määrä ja tyypit, virheiden korjauksen nopeus ja virheiden alkuperä [Hass 2008: 80].

Tämän insinöörityön tavoitteena on perehtyä toiminnallisuus- ja regressiotestausvaiheisiin ja verrata manuaalista ja automatisoitua lähestymistapaa niiden organisoinnissa. Työssä etsitään esivaatimuksia näiden vaiheiden automatisoinnille, joiden avulla voittaisin päättää, milloin testausta voidaan automatisoida ja mitkä testitapaukset tai sovelluksen toiminnallisuusalueet kannattaa automatisoida.

Insinöörityö alkaa yleiskatsauksella yleiseen testausprosessiin, ja sen jälkeen keskitytään toiminnallisuus- ja regressiotestaukseen. Luvussa 2 näille testauksen vaiheille annetaan lyhyt kuvaus ja selitetään näihin liittyviä organisoinnin yleisimpiä ongelmia. Luvussa 3 kuvataan organisoinnin parantamista testauksen kypsyysmallin (engl. Testing Maturity Model, TMM) avulla. Testauksen organisoinnin tasot selitetään ja jatkossa niihin viitataan käsiteltäessä käytännön organisoinnin kysymyksiä.

Luvuissa 4 ja 5 kuvataan toiminnallisuus- ja regressiotestauksen manuaalinen organisointi ja siihen liittyviä tehtäviä. Näiden lukujen sisältö perustuu paitsi kirjallisuudesta saatuun tietoon myös työkokemukseen järjestelmä- ja regressiotestauksen suorituksessa isossa projektissa. Projektin aikana työtehtäviin kuuluivat sekä toiminnallisuus- ja regressiotestauksen suunnittelu ja suorittaminen että raportointi. Insinöörityössä organisointitapoihin lasketaan testien suunnittelu ja suoritustapa, tulosten analysointi sekä työkalujen valinta. Luvussa 5 kuvattu regressiotestauksen organisointi nojautuu kirjoittajan työkokemukseen, jonka perusteella esitellään regressiotestauksen kierrosten käyttäminen ennen tuotteen siirtämistä hyväksymistestaukseen.

Luvussa 6 kuvataan testauksen automatisoinnin kehitystä ja esitellään viimeisen sukupolven mallipohjaiset testaustyökalut, joiden avulla testitapausten suunnittelusta tulee automatisoitu prosessi. Automaattinen suunnittelu on testauksen automatisoinnin korkein aste: se on uusi ja innovatiivinen tapa organisoida testausta. Automaattista suunnittelua arvioidaan tutustumalla Conformiq Tool Suite™ -työkaluun.

Luvussa 7 analysoidaan manuaalisen ja automatisoidun tapojen hyödyt ja puutteet. Testaustavan kelpoisuutta verrataan seuraavien kriteerien avulla

- toiminnallisten vaatimusten kattavuus testauksessa

- tuotteen valmiustaso hyväksymistestaukseen
- tarvittavien henkilöstöresurssien määrä
- uudelleenkäytettävyys ja muutosten hallinta.

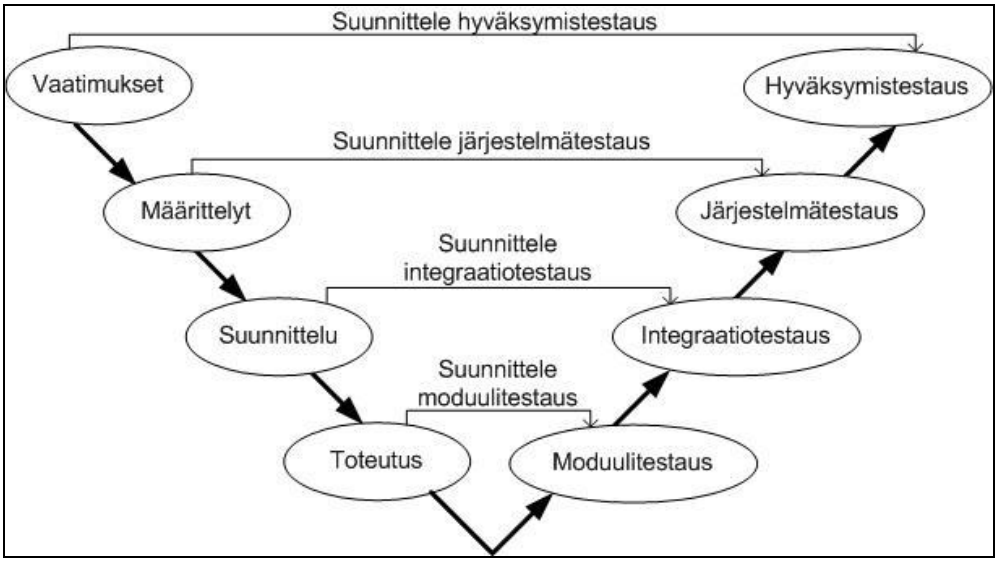
Analyysin tuloksena saadaan tietoon näiden tapojen edut ja haitat sekä ohjeistus siihen, minkä tyyppisissä projekteissa näiden menetelmien käyttö on kannattavaa. Insinöörityön lopuksi tehdään johtopäätökset ja kerrotaan sen soveltuvuudesta testauksen organisoinnissa.

2 Testausprosessi ja sen organisointi

Tässä luvussa kuvataan testausprosessia V-mallin avulla. Testauksen vaiheista käsitellään yksityiskohtaisemmin toiminnallisuus- ja regressiotestaus, jonka jälkeen esitetään näiden vaiheiden tyypillisimpiä organisointiongelmia.

2.1 Testauksen V-malli

Testausprosessi koostuu seuraavista testaustoiminnoista: suunnittelusta, toteutuksesta, suorittamisesta ja virheiden hallinnasta [Lewis 2005: 301]. Testausprosessi toteutetaan vaiheittain. Testausprosessin vaiheet voidaan kuvata eri tavoin: tässä insinöörityössä on käytetty V-mallia. V-malli esittää, miten eri testauksen vaiheet etenevät ja miten ne yhdistetään ohjelmistotuotannossa. Kuvassa 1 on esimerkki yhdestä V-mallin kuvaustavasta.



Kuva 1. V-malli [Watkins 2001: 41]

Kuvan vasen puoli edustaa vesiputousmallin ohjelmistokehityksen vaiheita, ja oikea puoli näyttää niitä vastaavia testausvaiheita. V-mallin mukaan testauksen vaiheet suunnitellaan, kun vastaava kehitysvaihe on saatu valmiiksi. Testauksensuunnittelu tehdään kehitysvaiheen dokumentaation perusteella. Testaus aloitetaan vastaavan toteutusvaiheen valmistuttua: moduulitestaus seuraa moduulin toteutusta jne. Moduulitestaus on yksittäisten moduulien tai yksiköiden testausta, ja integraatiotestaus on yhdistettyjen moduulien testausta. Hyväksymistestaus on sovelluksen testausta loppukäyttäjien toimesta. Vaikka V-malli alun perin tarkoitettiin vesiputousmalliprojekteille [Lewis 2005: 55], sen käyttö on mahdollista myös ketterissä projekteissa. Silloin voidaan ajatella, että jokaisella iteraatiolla on oma mini-V-malli. Tutkitussa projektissa V-mallia käytettiin tällä tavalla.

Testausvaiheiden lisäksi laadunvalvontaan kuuluvat katselmoinnit (engl. inspections tai peer review), joiden aikana koodia, dokumentaatiota tai muuta lopputuotetta käydään läpi. Testauksessa niitä käytetään testitapausten laadun ja jopa testaustehtävien suorittamisen parantamiseen. Tutkimukset todistavat, että katselmoinnit voivat olla tehokkaampia kuin testaus, jonka aikana testattavaa sovellusta käytetään suorittamalla testitapauksia. Katselmointien aikana löydetään enemmän virheitä ja ne vähentävät tehokkaasti virheiden poistamiskustannuksia. [Lewis 2005: 65.]

Testausprosessin organisointi on testauksen eri osien järjestämistä toimivaksi kokonaisuudeksi. Testaus jäsennetään tässä insinöörityössä jakamalla prosessi seuraaviin vaiheisiin:

- testitapausten suunnitteluun
- testitapausten suorittamiseen
- testaustulosten analysointiin
- virheiden jäljittämiseen ja
- testauksen lopettamiseen.

Näin ollen testauksen organisointi kattaa näiden vaiheiden järjestämisen ja yhdistämisen. Testausprosessin organisointia arvioidaan tässä insinöörityössä testauksen kypsyysmallin avulla (eng. Testing Maturity Model, TMM). Malli selittää, mitä toimintoja on toteutettava jokaisessa vaiheessa, jotta testauksen organisoinnin kypsyys kasvaisi.

2.2 Toiminnallisuus- ja regressiotestaus

Toiminnallisuustestaus on osa järjestelmätestausta. Järjestelmätestauksessa etsitään poikkeukset sovelluksen toiminnan ja käyttäjän antamien vaatimusten välillä. Toiminnallisuustestaus perustuu black box -testausperiaatteeseen, jonka mukaan testauksen aikana keskitytään siihen, miten sovellus toimii, eikä siihen, miten se oli koodattu.

Lewis luettelee toiminnallisuustestauksen lisäksi muita järjestelmätestauksen tyyppejä. Suorituskykytestauksen aikana tarkistetaan suorituskykyvaatimusten noudattaminen, vasteajat, transaktioiden nopeus ja muut aika-herkät toiminnallisuusalueet. Turvallisuustestauksessa testataan muun muassa datan käyttöoikeuksien mukainen käyttö. Kuormitustestauksessa tarkistetaan sovelluksen kyky käsitellä isoja datamääriä. Stressitestauksen aikana tarkistetaan järjestelmän käyttäytyminen stressitilassa, jolloin kuorma ylittää maksimirajoitukset. [Lewis 2005: 168.]

Toiminnallisuustestaus suunnitellaan toiminnallisen määrittelyjen pohjalta. Suunnittelun tuloksena saadaan testitapaukset joko tekstimuodossa tai nauhoitettujen skriptien muodossa. Sen jälkeen testitapaukset suoritetaan joko manuaalisesti tai automaattisesti annetussa aikavälissä. Saadut tulokset analysoidaan tarkastelemalla etukäteen sovittujen metriikoiden eli testauksen mittareiden avulla. Analyysitulokset raportoidaan projektin johdolle graafisesti ja kirjallisesti.

Testauksen aikana löydetyt virheet raportoidaan ja sen jälkeen niiden elinkaari jäljitetään. Testaustiimi päättää, valitaanko virhe korjattavaksi ja voi asettaa sille korjausprioriteetin. Korjauksen jälkeen virheen löytänyt testaaja tarkistaa korjauksen ja päättää, onnistuiko korjaaminen. Lopuksi testaaja uudelleentestaa virheeseen liittyvää testitapausta tai sovelluksen aluetta.

Tätä sykliä toistetaan niin kauan, kunnes testauksen lopettamiskriteerit täyttyvät. Lopettamiskriteereistä on aina sovittava ennen testauksen alkua; muuten testausta voitaisiin jatkaa loputtomasti. Toiminnallisuustestauksen loputtua toiminnallisuusalue siirretään regressiotestaukseen. Kuvattua prosessia käytettiin tutkitussa projektissa. Esimerkki kirjallisuudesta löytyvästä toiminnallisuustestauksen prosessista on esitetty liitteen 2 kuvassa 1.

Regressiotestauksen suorittamisen tarkoitus on varmistaa sovelluksen eheys korjausten ja muutosten jälkeen. Regressiotestaus on jatkuva prosessi, ja se poikkeaa toiminnallisuustestauksesta. Regressiotestauksen testitapaukset voidaan valita esimerkiksi toiminnallisuustestauksen testitapauksista ja/tai testikokonaisuuteen suunnitellaan omia testitapauksia. Sen jälkeen testauksen suoritus on samantapaista kuin toiminnallisuustestauksessa. Testaustulosten analysointivaiheessa on tärkeää selvittää, mistä löydetyt virheet johtuvat. Syitä ovat esimerkiksi aikaisemmin raportoidut virheet tai uudet, sovelluksen rikkomisesta kehitysprosessissa kertovat virheet. Kun regressiotestauksen lopettamiskriteerit täyttyvät, testauksen näkökulmasta sovellus voidaan siirtää asiakkaalle hyväksymistestaukseen.

2.3 Toiminnallisuus- ja regressiotestauksen organisoinnin ongelmat

Testaus on sovelluskehityksen kriittinen vaihe. Se on yksi haastavimmista ja kalleimmista kehityksen aktiviteeteista [Burnstein ym. 1996: 581]. Onnistunut testaus nostaa sovellustuotteen laatua.

Toiminnallisuus- ja regressiotestauksen organisoinnin aikana voidaan havaita erityyppisiä ongelmia, joista tässä mainitaan muutama. Testauksen suunnitteluvaiheessa puutteellisesti kootut testijoukot, jotka ovat kerättyjä yhteen tiettyä tarkoitusta varten joukkoja testitapauksia [Holopainen 2005: 8], väärentävät kattavuuden arviointia ja jättävät testausprosessin puutteelliseksi. Testauksen johdon tekemät arviointivirheet voivat vaarantaa koko projektin aikataulun. Testaus sovelluskehitysprojektin vaiheena on riippuvainen muista vaiheista, esimerkiksi suunnittelu- ja kehitysvaiheesta. Sovelluksen uusien korjausversioiden saatavuus voi vaihdella ja testaustiimillä on oltava muita tehtäviä kuin korjausten tarkistaminen. Odotusaikojen huomiointi kuuluu testausaikataulun suunnitteluun. Uusien tekniikoiden ja työkalujen käyttöönotto vaatii oikeaa koulutustarpeiden arviointia ja jatkuvaa monitorointia. Toinen tyypillinen vakava ongelma on testaustulosten vähäinen analysointi.

Testaus toimii koko sovelluskehitysprojektin monitorointityökaluna. Sen tuottamat tiedot pitää analysoida ja niiden avulla pitää laskea valittujen metriikoiden arvot. Testaustulokset eivät ole vain yksittäisiä virheraportteja ja testitapausten ajotuloksia. Virheraporttien määrä, vakavuus ja sovellusaluejakaumat voivat tuottaa arvokasta

tietoa sovelluksen tilasta. Ongelmien pitää olla tiedossa sekä projektin johdolla että kaikilla projektin tiimeillä. Tiedottaminen sovelluksen tilasta ei ole riittävä, jos siitä tietävät vain testaustiimin jäsenet. Ongelmien ja arviointivirheiden välttämiseksi testausprosessin pitää olla hyvin organisoitu ja määritelty.

Testauksen organisointia ja kypsyyttä ei kuitenkaan ole riittävän syvällisesti käsitelty sovelluskehityksen kypsyysmalleissa (esim. CMMI). Kehitysprosessiin ei sisällytetä testauksen parhaita käytäntöjä, eikä prosessissa käsitellä testauksen ongelmia. Kypsä sovelluskehitysprosessi ei pysty yksinään takaamaan sovelluksen laatua, vaan tarvitaan hyvin määriteltyä testausprosessia. Testausprosessin organisointimallin tulee sisältää arviointi-, kehitys- ja monitorointitekniikat. [Burnstein 2003: 264.]

Testaukselle on olemassa useita organisointi- ja kypsyysmalleja, esimerkiksi testaushallinnan malli (engl. Test Management Approach, TMAp), testausprosessin parantamisen malli (engl. Test Process Improvement, TPI) ja testauksen kypsyysmalli (engl. Testing Maturity Model, TMM). Nämä mallit pohjautuvat alan standardeihin ja parhaisiin käytäntöihin ja varmistavat testauksen ja lopputuotteen korkeaa laatutasoa. [Jacobs ym. 2000: 24.]

3 Testausprosessin organisoinnin parantaminen

Toimiva ja tehokas testausprosessi on edellytys menestyksekkäälle testaukselle. Testausprosessi voidaan organisoida monella eri tavalla. Hyvin alkeelliset testausprosessit sopivat yleensä vain tietylle projektille, ja niiden uudelleenkäyttö on lähes mahdotonta. Tällöin on vaikeaa seurata testauksen käytäntöjen ja tekniikoiden tehokkuutta. Kypsä testausprosessi on päinvastoin hyvin standardoitu ja joustava. Tohtori Marc C. Paulkin (Carnegie Mellon Yliopisto) mukaan kypsä testausprosessi on hallittu, mitattu, monitoroitu ja tehokas [Burnstein ym. 1996: 582]. Jotta testaustiimi pystyisi parantamaan testausprosessiaan, heidän tulee tunnistaa testauksen nykytilanne, evaluoida testausprosessin tavoitteita ja toimintoja ja tarvittaessa parantaa sitä lisäämällä prosessiin puuttuvia käytäntöjä.

3.1 Testauksen johtaminen

Klassisen johtamismallin mukaan sovelluskehitysprojektin johtaminen sisältää suunnittelua, organisointia, henkilöstön valintaa, ohjausta ja evaluointia [Burnstein ym. 1996: 10]. Vaikka sovelluskehityksen ja testausprosessin johtamismallit ovat muuttuneet, nämä johtamisen funktiot ja niiden toiminnot ovat yleispäteviä myös nykyään. Tässä insinöörityössä käsitellään jatkossa vain testausprosessin organisointia.

Thaylorin mukaan organisointi on työn järjestämistä ja tiedon välittämistä tavoitteiden saattamiseksi sekä vastuun ja toimivallan jakamista näiden tavoitteiden saavuttamiseksi [Thayer 1990: 16]. Organisointi jaetaan toimintoihin ja toiminnot jaetaan tehtäviin. Esimerkki organisointiin sisältyvästä toiminnosta on testaustiimin tehtävien määrittely ja ryhmittely. Tämä toiminto jaetaan esimerkiksi seuraaviin tehtäviin:

- testaustehtävien määrittely
- testaustehtävien työkuorman arviointi
- testaustehtävien ryhmittäminen ja jako.

Jokainen tehtävä sisältää käytäntöjä, joiden valinta riippuu testaustiimin osaamistasosta ja organisaation laadunvalvonnan järjestelmästä. Lopulta testaustiimi dokumentoi kaikki valitut toiminnot ja käytännöt. Tätä dokumenttia yleistämällä saadaan testausprosessin määrittely.

Johtamismalleja ja alan parhaita käytäntöjä soveltamalla testaustiimi voi kehittää tehokkaan testausprosessin, joka vastaa tiimin osaamistasoa ja testauksen laadun vaatimuksia. Kypsä testausprosessi sisältää joukon tarkasti määriteltyjä testauksen toimintamalleja, kattaa koko testauksen elinkaaren ja määrittää testauksen suunnitteluprosessia. Testausprosessin määrittelyyn kuuluvat myös työkalujen valinta ja laadunvalvonta. Testaustiimin on kuitenkin vaikeaa määritellä koko testausprosessia itse. Yleensä testausprosessia määriteltäessä käytetään valmiita suosituksia, organisointi- ja kypsyysmalleja.

3.2 Testing Maturity Model (TMM)

Testauksen kypsyysmalli (engl. Testing Maturity Model, TMM) kehitettiin kuvaamaan testausprosessin kypsyystasoja ja keinoja parantaa testauksen organisointia. Malli esiteltiin vuonna 1996. Sen jälkeen sitä on kehitetty ja sen rinnalle on noussut muita malleja esim. Puolustusvoimien testauksen kypsyysmalli (engl. Ministry of National Defense – Testing Maturity Model, MND-TMM) ja testauksen organisoinnin kypsyysmalli (engl. Organizational Testing Management Maturity Model, OTM3). TMM-mallin tarkoitus on täydentää CMMI (Configuration Maturity Model Improved) -mallia testauksen kohdalla ja luoda määriteltyjä tasoja ja toimintoja testauksen prosessin kehittämistä varten. TMM on tarkoitettu testauksen johdolle, jonka avulla he voivat arvioida testausprosessin kypsyyttä, kehittää sitä ja saavuttaa seuraavat kypsyystasot. TMM:n jokaisella tasolla annetaan suositeltuja toimintatapoja ja malli, jonka avulla voidaan arvioida yrityksen testausprosessin tasoa. TMM suosittelee myös testaustyökaluja ja V-mallin laajentamista. [Burnstein ym. 1996: 581.]

TMM sisältää viisi testausorganisaation kypsyystasoa. TMM-mallin rakenne on esitetty liitteen 1 kuvassa 1. Jokaisella tasolla on omat päätavoitteensa ja alatavoitteensa. Jokaisella tavoitteella on joukko toimintoja, tehtäviä ja vastuita. Testaustiimin on saavutettava kaikki kunkin tason pää- ja alatavoitteet, jotta sen testausprosessia voidaan pitää tehokkaana. Jotta testausorganisaatio osaisi tunnistaa, millä kypsyystasolla se on nykyhetkellä, TMM-malliin on sisällytetty arviointimalli. Arviointimallin avulla testaustiimi pystyy kehittämään tehokkaampaa testausprosessia ja tehostaa olemassa olevia käytäntöjä.

Ensimmäinen TMM-mallin taso on aloitteleva taso. Sen mukaan testaus alkutasolla on kaoottista ja epämääräistä. Testausta ei voida erottaa virheiden paikallistamisesta koodissa (debuggauksesta). Tämä taso ei sisällä mitään varsinaisia tavoitteita, koska testausprosessia ei ole määritelty.

Seuraava taso selventää testausvaiheen määrittelyä. Tällä tasolla testaus on määritelty erillisenä prosessina, joka seuraa sovelluksen kehitystä. Tämän tason tavoitteet ovat

1. testauksen ja debuggauksen tavoitteiden kehittäminen
2. testauksen suunnitteluprosessin kehittäminen

3. testauksen perusmenetelmien ja tekniikoiden vakiinnuttaminen.

Samaa testausprosessia voidaan hyödyntää ja käyttää uudelleen eri projekteissa. Testausprosessin toimintoja tällä kypsyystasolla ovat esimerkiksi

- testitapausten suunnittelu
- testauksen dokumentointi
- eri testaustekniikkojen käyttöönotto.

Kolmas TMM-mallin kypsyystaso määrittää integrointia eri tasoilla. Tällä kypsyystasolla testaustiimin tehtävä on lisätä laadunvalvontaa testausprosessiin. Tason tärkeimmät tavoitteet ovat testauksen integrointi sovelluskehityksen elinkaareen sekä testausprosessin hallinta ja jatkuva monitorointi. Testauksen suunnittelu pyritään aloittamaan samaan aikaan kuin sovelluskehitys.

Neljäs TMM-taso lisää testaukseen hallittavuutta ja mitattavuutta. Tämän tason päätavoitteet ovat testaustiimin osaamistason jatkuva kehittäminen ja testausprosessin evaluointimallin kehittäminen. Testausprosessi tällä tasolla auttaa kehittämään sovellusta, joka on luotettava, hallittava ja turvallinen.

TMM-mallin viimeinen taso määrittää testausprosessin jatkuvaa parantamista ja virheiden ehkäisyä. Tehokas sovelluskehityksen ja testauksen laadunvalvonta on tämän tason päätavoite.

4 Toiminnallisuustestauksen organisointi manuaalisesti

Tässä luvussa kuvataan, mitä toimintoja kuuluu toiminnallisuustestaukseen, jos sitä päätetään tehdä manuaalisesti. Sen lisäksi kuvataan, miten toiminnallisuustestauksessa saadut tulokset analysoidaan. TMM-mallia sovelletaan eri tehtävien toteuttamisessa. Testauksen jakaminen vaiheisiin ja mm. toiminnallisuustestauksen erittely muista vaiheista kuuluu TMM:n tason 2 suunnittelutavoitteeseen.

4.1 Toiminnallisuustestauksen testitapausten suunnittelu ja suorittaminen

Toiminnallisuustestauksen suunnittelu alkaa, kun asiakas on hyväksynyt toiminnallisen määrittelyn. Toiminnallisuustestaus on black box –testausta [Lewis 2005: 29]. Tämä

tarkoittaa sitä, ettei testejä suunniteltaessa oteta huomioon tapaa, jolla sovellus on toteutettu, vaan tarkistetaan, vastaako sovellus asiakkaan vaatimuksia.
Manuaalisessa organisointitavassa suunnittelu tehdään perinteisesti: testaajat analysoivat määrittelyt, poimivat testattavat alueet ja suunnittelevat testitapaukset. Näin valmistettujen testitapausten kattavuuden ja laadun arviointi on täysin subjektiivinen ja voi vaihdella riippuen siitä, kuinka kokenut testaaja on tehnyt suunnittelutyön. Testitapausten laadun vaihtelua voidaan välttää seuraamalla TMM-mallia. Vaihtelua estetään määrittelemällä suunnitteluprosessi (taso 2), perustamalla testaustiimi (taso 3), kouluttamalla testaajia ja katselmoimalla testitapauksia (taso 4). Testaajalla on oltava tiedot testauksen periaatteista, prosesseista, metriikoista, standardeista, suunnitelmista, työkaluista ja metodologiasta. Hänen pitää noudattaa niitä testaustehtäviä suorittaessa. [Burnstein 2003: 3.]

Testitapaukset ovat sanallisesti kuvattuja käyttäjän toimintoja. Sanalliset testitapaukset kirjoitetaan joko dokumenttiin tai tallennetaan työkalun avulla. Sanallisten testiskriptien tarkkuus voi vaihdella. Testidata voi olla annettu esimerkkimuodossa tai tarkoilla arvoilla. Tarkistettavat tiedot voidaan kuvata testitapauksessa tai viitata määrittely-dokumentteihin, josta testaajan on tarkistettava nykyarvoja.

Tämän työn pohjaksi tutkitussa esimerkkiprojektissa testitapaukset kirjoitettiin ja ajettiin Rational® Manual Tester -työkalun avulla. Manual Tester mahdollistaa testitapausten hallinnan, koska kaikki testitapaukset ovat samassa paikassa ja käyttäjät ajavat testitapaukset riippumatta toisistaan. Testitapausten muutokset tehdään yhteen paikkaan, ja ne ovat näkyvissä kaikille käyttäjille heti tallentamisen jälkeen.

Manuaalinen testitapausten suunnittelu ja ylläpito vaatii huomattavan määrän työtunteja. Testitapausten ylläpitäminen tarkoittaa niiden korjaamista sovelluksen määrittelyiden muututtua. Asiakkaan muutospyynnöt jäljitetään niiden ilmoittamis-hetkestä. Muutospyynnöt analysoidaan ja päätetään, otetaanko ne mukaan. Jos muutospyyntö hyväksytään, sen jälkeen määrittelyt ja toteutus analysoidaan ja tarvittaessa muutetaan. Testaajan on seurattava tätä prosessia ja tehtävä muutokset kaikkiin sellaisiin testitapauksiin, joita hyväksytyt muutokset koskevat, ja sitten uudelleen testattava kaikki päivitetyt testitapaukset. Pienetkin unohdetut muutokset saattavat aiheuttaa vääriä hälytyksiä, silloin testitapaukset ajetaan epäonnistuneesti ja

pahimmassa tapauksessa jopa korjataan vastoin uusia määrittelyjä. TMM-mallin seuraaminen suunnitteluvaiheessa tuottaa tuloksena huolellisesti suunnitellun testausprosessin ja edellä mainitut ongelmat voidaan välttää.

Manuaalisella suunnittelulla on hyötypuolensa:

- Manuaalisia testejä on suhteellisen helppo tehdä.
- Manuaaliset testitapaukset ovat useimmiten ymmärrettävässä muodossa sekä projektin tiimille että asiakkaalle.
- Asiakkaat voivat osallistua aktiivisesti testien suunnitteluun ja tuottaa loppukäyttäjän näkemystä testaukseen.
- Kokenut testaaja voi jo suunnitteluvaiheessa vaikuttaa laatuun panostamalla testauksen ongelmallisimpiin kohtiin ja esimerkiksi priorisoinnin avulla vähentämällä testitapausten määrää alimman prioriteetin alueissa.

Testaus voidaan aloittaa, kun testitapaukset ovat valmiita, ja toteutustiimi luovuttaa testattavan sovellusversion. Testauksen suorittaminen tuottaa testituloksen (esim. testitapaus ajettu onnistuneesti, epäonnistuneesti tai ei ole ajettavissa) ja mahdollisesti virheraportin. Testauksessa löydetyt virheet voidaan jakaa kahteen ryhmään: muutospyyntöihin ja vikoihin. Muutospyyntöjä ovat useimmiten asiakkaan pyynnöt muuttaa sovelluksen toiminnallisuutta tai ominaisuuden määrittelyä ja toteutusta. Viat ovat joko poikkeamia määrittelyistä tai sovelluksen toimintaa häiritseviä tai rikkovia virheitä.

Jos testitapaukset on tehty sanallisessa muodossa, niin testaajat itse suorittavat kaikki testitapaukset manuaalisesti ja raportoivat tulokset. Rational® Manual Tester -työkalussa testitapaukset ajetaan antamalla jokaiselle testiaskeleelle ja lopputulokselle arvo onnistunut, epäonnistunut tai tulokseton, ja raportoidaan testitapaukseen liittyvät virheet. Testitulokset tallennetaan, ja ne ovat raportoitavissa eri muodoissa. Myös testitapauskohtaiset raportit ovat mahdollisia. Rational® Manual Tester -työkalua käytettiin insinöörityössä tutkitussa projektissa toisen IBM:n työkalun Rational® ClearQuest kanssa. IBM Rational® ClearQuest mahdollistaa muun muassa virheiden (vikojen ja muutospyyntöjen) ja testaustulosten raportoinnin ja virheiden elinkaaren seurannan. ClearQuest tukee kyselyiden luomista ja ajamista graafisella

käyttäjäystävällisellä tavalla. Molemmista työkaluista raportoidut virheet saa tallennettua samaan paikkaan.

Manuaalisesti suoritetut testitapaukset voivat joko tuottaa lisäarvoa tuloksiin tai huolimattomasti suorittuna vääristellä niitä. Tässä on tärkeää kontrollointi ja testaajien koulutus. TMM-tasolla 3 on kaksi tavoitetta, jotka vaativat näiden ongelmien ratkaisemista. Ne ovat testausorganisaation muodostaminen ja testausprosessin monitorointi ja kontrollointi. Testausprosessin läpinäkyvyys on tärkeää. Silloin voidaan tarkkailla testaussuunnitelman noudattamista ja korjata ajoissa poikkeukset, jolloin voidaan varmistaa testauksen tavoitteiden saavuttaminen. Kontrollointi ja monitorointi toteutetaan seuraamalla standardeja ja testaussuunnitelmaa, analysoimalla testausraportit, laskemalla metriikkoja ja arvioimalla testausprosessin kulkua ja tehokkuutta. [Burnstein ym. 1996: 585.]

Manuaalisella suorituksella on myös huonoja puolia. Monotoniset toistot saattavat väsyttää testaajia, jolloin heidän huomiointikykynsä laskee. Manuaalisen testauksen kattavuus on pienempi kuin automatisoidun sen hitauden takia. Samoilla resursseilla toteutettu automaattisten skriptien suoritus tuottaa isomman kattavuusasteen. Manuaalisen suorituksen objektivisuutta voidaan parantaa antamalla samat testitapaukset ajettavaksi eri testaajille, jolloin eri testaajat löytävät eri virheet.

4.2 Testaustulosten analysointi

Testauksen tulokset on analysoitava, tai niiden arvo katoaa. Projektin johto tarvitsee tuloksia raportoituna saadakseen arvioitua projektin tilanteen ja mahdolliset korjaustarpeet projektisuunnitelmaan. Erikseen seurataan esim. testitapausten ajotuloksia ja raportoituja virheitä. Jokaisessa projektissa valitaan joukko metriikoita, joita lasketaan ja seurataan. TMM:n tasolla 5 testaustulosten analysoinnin avulla estetään uusien virheiden esiintymistä. Tästä huolehtii oma tiimi, jonka tehtävänä on analysoida testaustiimin raportit ja niiden avulla selvittää tyypilliset virheet ja niiden alkuperät. Virheiden estämiseen luodaan omat toiminnot, joilla parannetaan määrittely-, kehitys- ja testaustiimin työnlaatua.

Yksi tärkeimmistä toiminnallisuustestauksen metriikoista on kattavuus. Kattavuutta voi määrittää laskemalla testattujen toiminnallisuuksien, käyttötapausten tai skenaarioiden

(sekä peruskulku että poikkeustilanteet) osuutta. Kattavuutta lasketaan kaavalla 1 [Kan 2002: 302]:

$$kattavuus = \frac{Vt}{V} \quad (1)$$

Vt on testattujen vaatimusten määrä

V on kaikkien vaatimusten määrä.

Kaavassa 1 käytetty yksikkö voi olla toiminnallisuusalue, käyttötapaus, skenaario tai testitapaus, mutta täsmällisin on testausvaatimusten käyttö. Silloin testaussuunnitelmassa on oltava analysoitu ja listattu kaikki vaatimukset testaustasolla. Se tarkoittaa, että asiakkaan vaatimukset pilkotaan yksinkertaisiin testausvaatimuksiin, joista testauksen aikana saadaan selkeä testaustulos. Esimerkkinä voi olla etusivun lataaminen. Vaikka se voi toistua kymmenissä asiakkaan alkuperäisissä vaatimusskenaarioissa ja testitapauksissa, testausvaatimusten listassa se on vain yksi rivi. Jokainen testitapaus voidaan sen jälkeen esittää testausvaatimusten yhdistelmänä. Kun testausvaatimuslista on valmis, kattavuus lasketaan jokaisen testikerroksen jälkeen edellä mainitun kaavan avulla.

On olemassa muitakin tapoja seurata kattavuutta, esimerkiksi ylläpitämällä jäljitettävyyttä. Jäljitettävyyttä on kahta tyyppiä: eteenpäin jäljitettävyys ja taaksepäin jäljitettävyys. Taaksepäin jäljitettävyys tarkoittaa koodin jäljittämistä esivaatimuksiin. Näin todetaan, että toteutettiin vain vaaditut toiminnallisuudet eikä sovelluksessa ole ylimääräisiä osuuksia. Tätä jäljitettävyyttä käytetään toteutustyön seurannassa. Silloin resursseja käytetään vain sellaisiin tehtäviin, joille löytyy jäljitettävä asiakkaan vaatimus.

Eteenpäin jäljitettävyyttä seurataan toiminnallisuustestauksessa – miten vaatimukset ovat toteutettu ohjelmistotuotannon jokaisessa vaiheessa vaatimusten analysoinnista testaukseen. Näin testauksen ja projektin johto seuraa toiminnallisuuksien tilat ja arvioida niiden valmistumisaikataulua.

Jäljitettävyysmatriisi (engl. requirements traceability matrix) on esimerkki tyypillisestä muodosta jäljitettävyyden seurannasta. Matriisi osoittaa, että kaikki vaatimukset on

toteutettu ja testattu. Matriisia päivitetään jokaisessa ohjelmistokehitysvaiheessa. [Lewis 2005: 351.] Taulukossa 1 on esimerkki matriisista.

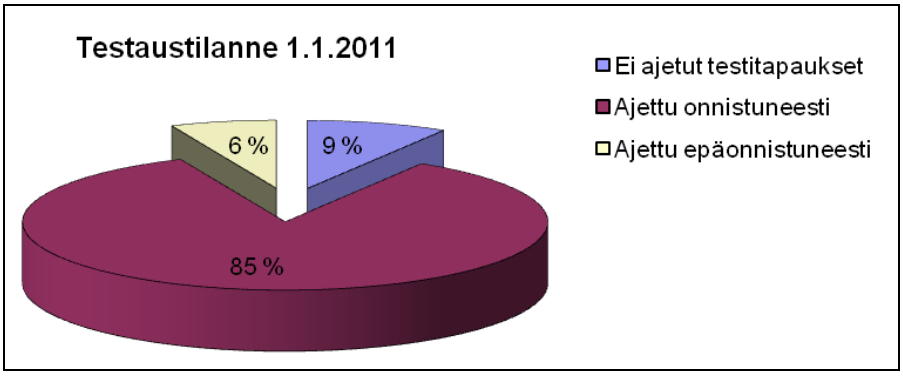
Taulukko 1. Jäljitettävyysmatriisi [Lewis 2005: 352]

Asiakkaan vaatimukset	Määrittely	Suunnitteludokumentaatio	Tekninen määrittely	Moduulitestaus	Integraatiotestaus	Järjestelmätestaus	Hyväksymistestaus
V 2.4	KT 1.2	N 2.5.	TK 2.4	M 2.3	IT 2	TT 3	HT 11

Matriisi sisältää kaikki toiminnallisuuden yksiköt ja testitapaukset. Lyhenteet viittaavat dokumentteihin tai lopputuotteisiin. Esimerkiksi lyhenne V2.4. viittaa Vaatimukseen 2.4 ja KT1.2 Käyttötapaukseen 1.2.

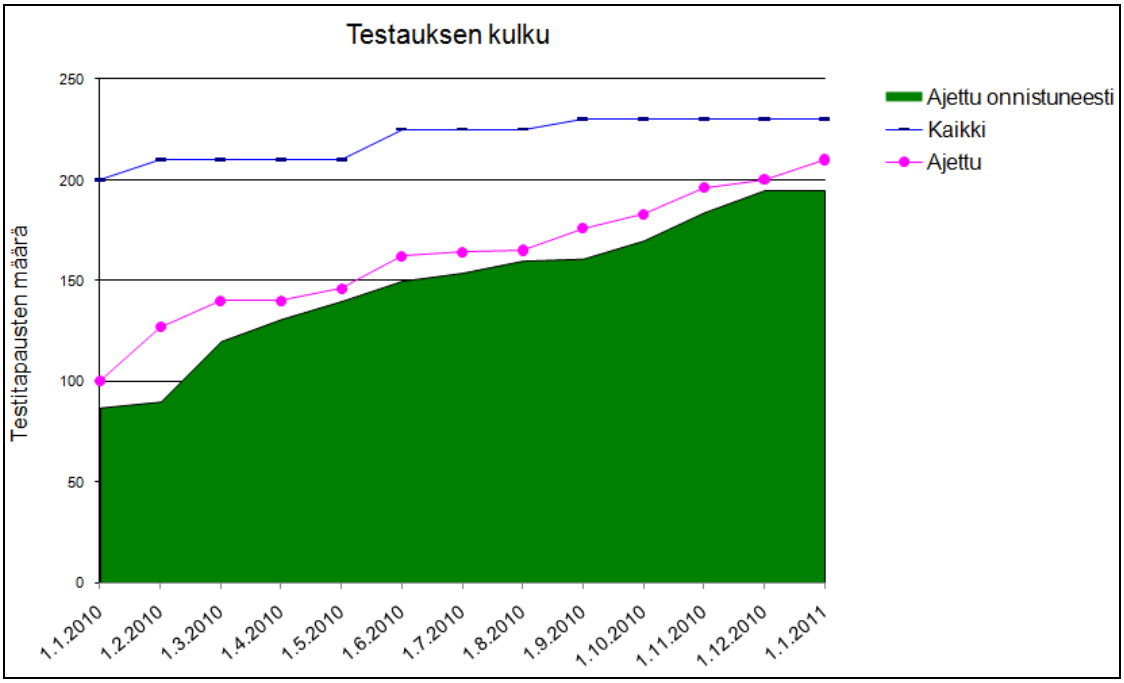
Matriisin avulla tarkistetaan, että kaikki toiminnallisuuden alueet ovat testattu jossain testitapauksessa. Matriisin avulla voidaan myös tarkkailla toiminnallisuusyksiköiden valmiutta regressiotestaukseen ja myöhemmin hyväksymistestaukseen. Jäljitettävyys on testauksen perustekniikka ja sen käyttö on yksi kypsyyden tavoitteista jo TMM-mallin tasolla 2.

Jos testien ja raportoitujen virheiden määrä on iso, silloin testaustilannetta seurataan erityyppisillä raporteilla. Kuvissa 2–4 esitetään esimerkkejä testitapausten testaustilanteen raporteista.



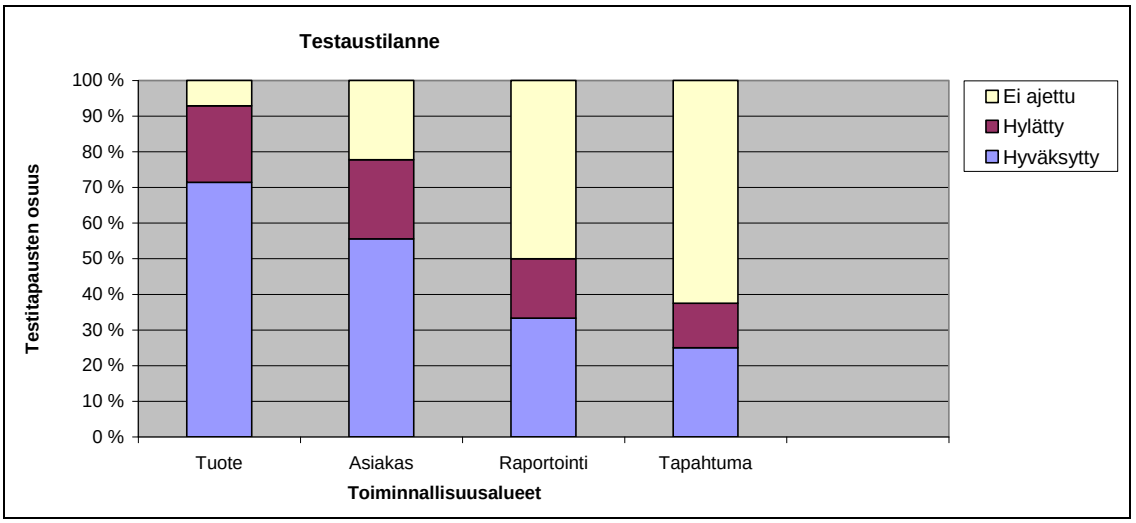
Kuva 2. Raportti testauksen ajotuloksista

Kuvan 2 kaltaisissa raportissa nähdään sen hetken testaustilanne: kuinka paljon testitapauksia on testattu yhteensä ja kuinka moni niistä on testattu onnistuneesti. Kuvassa 3 samantyyppinen tieto on esitetty trendikäyrinä. Trendikäyrä kuvaa, miten testaustilanne on muuttunut vuoden aikana.



Kuva 3. Raportti testauksen kulusta [Kan 2002: 272]

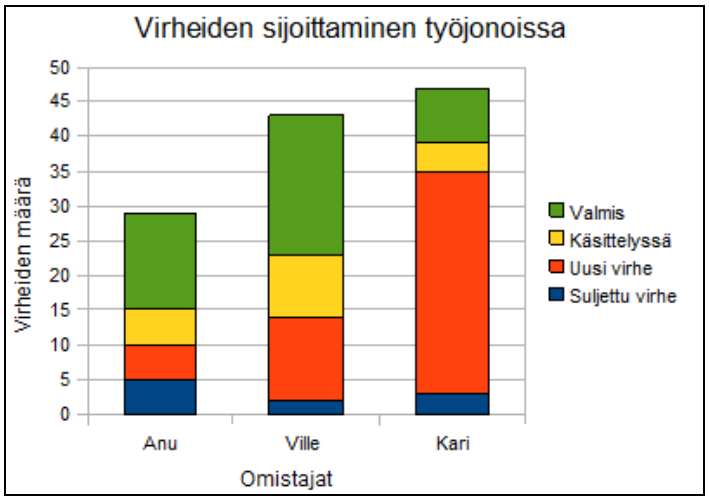
Kuvassa 4 näytetään raportointia toiminnallisuusalueittain. Raportista käy ilmi, miten testaustilanne edistyy eri toiminnallisuusalueissa. Raportin avulla sekä huomataan milloin toiminnallisuusalueet saadaan valmiiksi että nähdään, mitkä toiminnallisuudet osoittautuivat olevan monimutkaisempia kuin niitä arvioitiin.



Kuva 4. Raportti testaustilanteesta toiminnallisuusalueittain

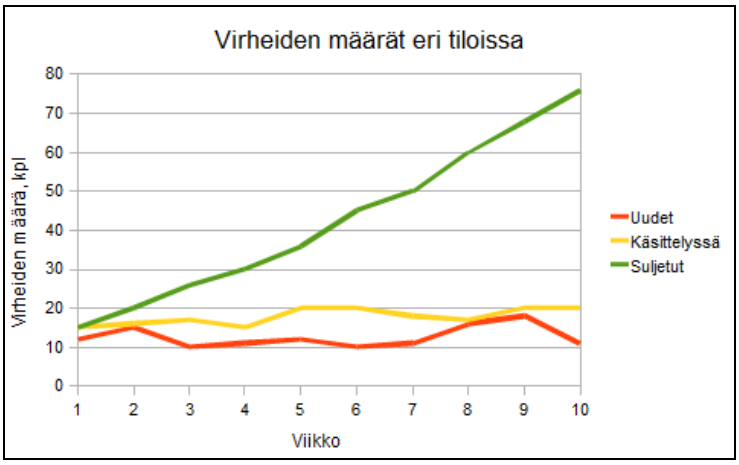
Kun testien suorittamistulokset tallennetaan esimerkiksi Excel-tiedostoon, tällaisten kuvioiden tekeminen vie muutamia minuutteja. Mutta se edellyttää testaustilanteen säännöllistä seurantaa.

Testauksen tuloksena saadaan myös virheraportteja, joita on seurattava ja analysoitava. Virheiden raportointi ja analysointi on yhtä tärkeää kuin testaustilanteen seuraaminen. Rational® ClearQuest -työkalu panostaa eniten virheiden jäljittämiseen ja niiden analysointiin. Rational® ClearQuest -työkalussa testaaja pystyy tekemään omia raportteja Velho-työkalun avulla. Raportille voi valita muodoksi Excel-, Word-, HTML- tai cvs-dokumentin. Raporttien avulla haetaan virheiden listoja, joiden avulla virheet voi ryhmitellä. Graafisia kaavioita on valittavissa 3 tyyppiä: jakaumat, trendikäyrät ja historiakäyrät. Jakaumat seuraavat virheitä esim. projektin työntekijöiden työjonoissa. Kuvassa 5 on esimerkki siitä.



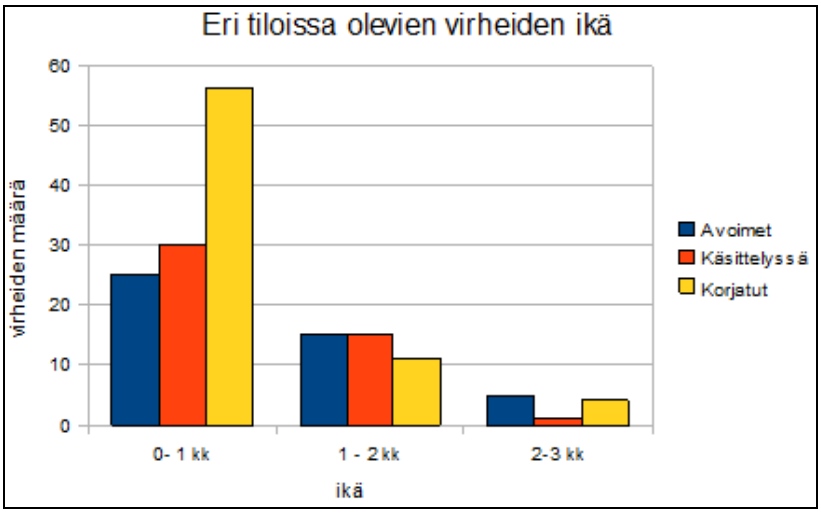
Kuva 5 Virheet työjonoissa [White 2000: 268]

Kuva 5 auttaa jakamaan työmääriä tasaisemmin ja reagoimaan ajoissa tilanteisiin, joissa työntekijät on ylikuormitettu. Kuvassa 6 esitetty trendikäyrä näyttää tarkkailtavien virhetyyppien määrät.



Kuva 6 Virheiden määrät eri tiloissa [White 2000: 269]

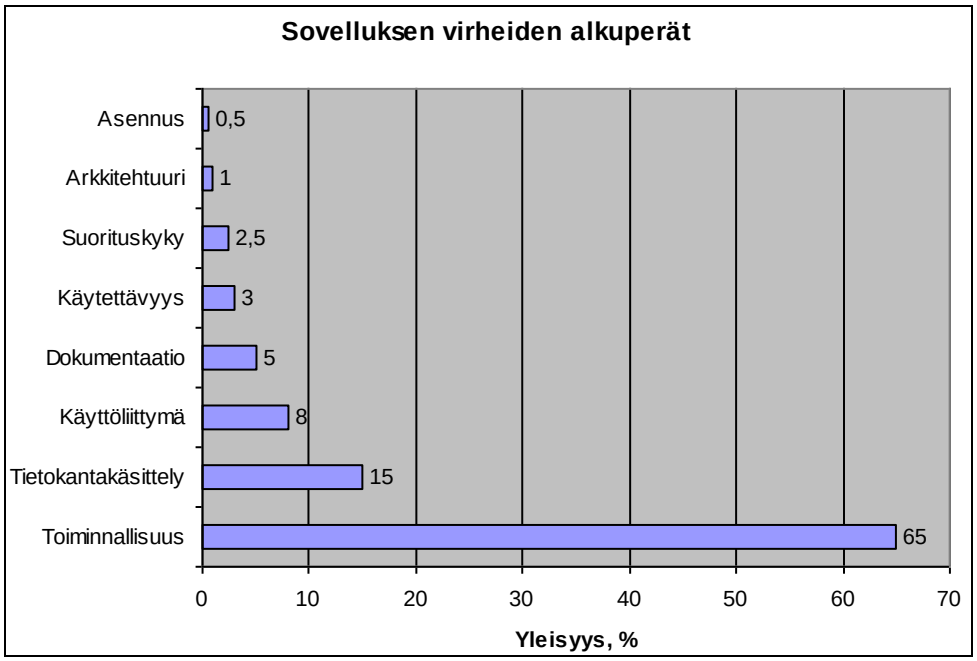
Tämän trendikäyrän avulla voi arvioida eri sovelluksen versioiden ja tuotteen kokonaislaatua. Jos kehitystiimi ei muutu paljon, silloin näiden käyrien avulla voi ennustaa tulevien virheiden määrä. Viimeinen esitetty tyyppi seuraa eri tiloissa olevien virheiden ikä (kuva 7).



Kuva 7 Virheiden ikäjakauma [White 2000: 270]

Kuvasta 7 selviää, mitkä virheraportit ovat olleet pitkään käsittelemättä ja mitkä on palautettu väärin tehdyn korjauksen takia. Näin virheraportit voidaan priorisoida ja seurata kaikkien korjattavaksi valittujen virheiden sulkeminen.

Yksi laadunvalvontatyökaluista on Pareto-jakauma (kuva 8).



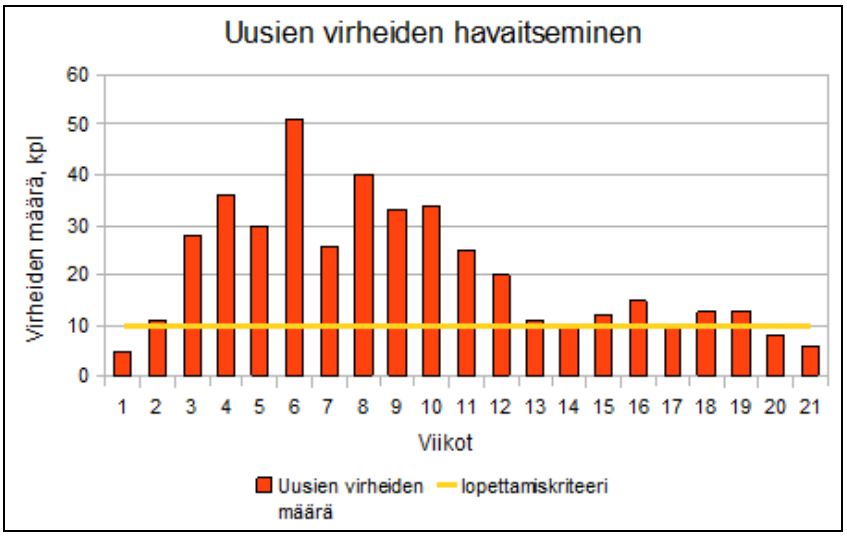
Kuva 8 Pareto-jakauma

Pareto-jakauma esittää graafisesti periaatteen, jonka mukaan 20 % virheiden aiheuttajista tuottaa 80 % virheistä. Näin korjaamalla aiheuttajia eikä yksittäisiä virheitä saavutetaan eliminoitua 80 % virheistä. Pareto-jakauman avulla korjausresurssit keskitytään tärkeisiin aiheuttajien poistamisen ja kehitysprosessia parannetaan ehkäisemällä yleisimmät virhetilanteet.

4.3 Toiminnallisuustestauksen lopettaminen

Copeland luettelee kirjassaan viisi peruskriteeriä testauksen lopettamiseen. Niistä kolme ensimmäistä sopivat tuotteen siirtämispäätöksen seuraavaan testauksen vaiheeseen. Ensimmäinen kriteeri on sovitun kattavuusasteen saavuttaminen. Projektin suunnitelmassa on oltava sovittu kattavuuden arvo, joka on saavutettava testauksessa. Tähän voidaan lisätä, että kattavuudessa kannattaa ottaa huomioon myös läpäisseiden testien osuus kaikista ajetuista testeistä.

Toinen kriteeri on se, että uusien virheiden havaitseminen jää alle määritellyn rajan. Esimerkki on esitelty kuvassa 9. Jos lopettamiskriteeriksi on valittu uusien virheiden löytäminen alle 10 uutta virhettä viikossa, testauksen voi lopettaa kuvan mukaisesti viikon 20 jälkeen.



Kuva 9 Uusien virheiden havaitseminen [Copeland 2004: 253]

Kolmas kriteeri liittyy sovellukseen hintaan. Jos uusien virheiden löytäminen maksaa enemmän kuin niiden potentiaalinen vahinko, testauksen voi lopettaa. Kaikkein halvimmat ovat mahdollisimman aikaisin löydetyt virheet (tietenkin virheiden

löytäminen katselmoinnin aikana ja korjaaminen jo suunnittelu- ja määrittelyvaiheessa on vielä halvempaa). Kun testaus jatkuu, niin virheiden löytämishintakin kasvaa. Jos sovellus ei ole kriittinen, niin lopettamiskriteeriksi voi hyväksyä sellaisen tilanteen, kun uusien virheiden hinta ylittää virheiden potentiaalisen vahingon. [Copeland 2004: 250–255.]

Toiminnallisuustestaus on vain yksi osa järjestelmätestausta, ja sen takia pelkkä toiminnallisuustestauksen onnistuminen ei tarkoita automaattisesti siirtoa hyväksymistestaukseen. Sovellus luovutetaan hyväksymistestaukseen vasta silloin, kun kaikkien järjestelmätestauksen tyyppien katsotaan olevan riittävästi ajettu. Hyväksymistestausta tekevät loppukäyttäjät, ja sen tuloksen perusteella tuote joko hyväksytään tai palautetaan lisäkehitykseen.

5 Regressiotestauksen organisointi manuaalisesti

Regressiotestauksella ei ole omaa vaihetta V-mallissa. Tässä luvussa kuvataan, milloin regressiotestausta tarvitaan. Organisointinäkökulmasta kerrotaan miten regressiotestauksen testijoukot voidaan luoda. Lopuksi analysoidaan tapaa, jolla regressiotestausta ajettiin tutkitussa projektissa.

5.1 Regressiotestauksen tarve

Regressiotestauksen tarkoitus on löytää virheet, jotka ovat syntyneet virheiden korjausten jälkeen tai silloin, kun sovellukseen lisätään uusia toiminnallisuuksia. Regressiotestaus on jatkuva prosessi. Regressiotestaus aloitetaan, kun joku toiminnallisuusosuus pääsee läpi toiminnallisuustestauksesta, ja jatkuu siihen asti, kunnes koko sovellus on läpäissyt regressiotestauksen. Regressiotestausta käytetään eri tilanteissa ja ohjelmistotuotannon vaiheissa, esimerkiksi

- ohjelmaan on lisätty uusia toiminnallisuuksia
- ohjelmaan on korjattu virheitä
- ohjelma on vakautunut ja valmis luovutukseen
- ylläpidon aikana, kun ohjelmaa muutetaan tai ennakoidaan ohjelmistosta riippumattomia ympäristömuutoksia. [Binder 1999: 760.]

Regressiotestaus suoritetaan muutosten tapahduttua sovelluksessa kehityksen, ylläpidon tai kehitysvaiheen aikana. Regressiotestausta on periaatteessa suoritettava koko laajuudessaan joka kerta, kun joku muutos tai korjaus on tehty. Regressiotestauksen avulla todetaan se, että muutokset eivät aiheuttaneet uusia virheitä. Näin kuitenkin harvoin käy. Regressiotestauksen ajaminen jokaisessa kehitysvaiheessa ja muutosten jälkeen on usein mahdotonta projektin aikataulun ja budjetin takia. Hyvä tapa ketterässä projektissa olisi ajaa regressiokierros jokaisen kehitysvaiheen valmistuttua ja näin varmistaa edellisen vaiheen toiminnallisuuden toimivuus. Kun järjestelmään korjataan laskuvirheet, rajapintavirheet, loogiset tai kontrollin kulkuun vaikuttavat virheet, järjestelmässä tulee ajaa regressiotestausta. Arkkitehtuurimuutokset vaativat mahdollisesti jopa kaikkien regressiotestien ajoa. Kosmeettisten virheiden korjaukset eivät välttämättä vaadi regressiota. Regression testijoukkoa ylläpidetään koko ajan ja virheen löydettyä lisätään tarvittaessa joukkoon uusi regressiotestitapaus, joka tarkistaa virhetilanteen. [Lewis 2005: 509.]

5.2 Regressiotestauksen suunnittelu

Regressiotestaus vie kaikista muista testauksen tyypeistä eniten aikaa ja resursseja, jos sitä ajetaan, joka kertaa regressiota vaadittavien muutosten jälkeen. Sen takia regression tarkka suunnittelu on tärkeää ja regressiotestaukseen valittu strategia voi säästää päiviä ja vapauttaa henkilöstöresursseja. Regressiotestitapausten suunnittelun aikana pitää vastata kysymykseen: mitä pitää testata, jotta asetettu kattavuusaste saavutetaan.

Regressiotestauksessa voidaan käyttää vanhoja testitapauksia, luoda uusia testitapauksia tai käyttää osa vanhoista ja täydentää niitä uusilla testitapauksilla. Viimeinen tapa sekä hyödyntää uudelleenkäyttöperiaatteetta että nostaa regression arvoa analysoimalla ja täydentämällä testijoukkoja.

Testitapausten uudelleenkäyttämisessä noudatetaan seuraavia metodologioita:

- Testitapauksia harvennetaan poistamalla osa vanhoista testitapauksista kokonaan.
- Käytetään valintateknikoita, jolloin testijoukosta valitaan regressioon vain tiettyjä kriteerejä noudattavat testitapaukset.

- Testitapaukset priorisoidaan ja suoritetaan ensin korkeamman prioriteetin testitapaukset ja, jos resursseja riittää, ajetaan muutkin. [Holopainen 2005: 30.]

Edellä mainitut metodologiat voidaan yhdistää. Silloin suunnittelu voidaan jakaa vaiheisiin:

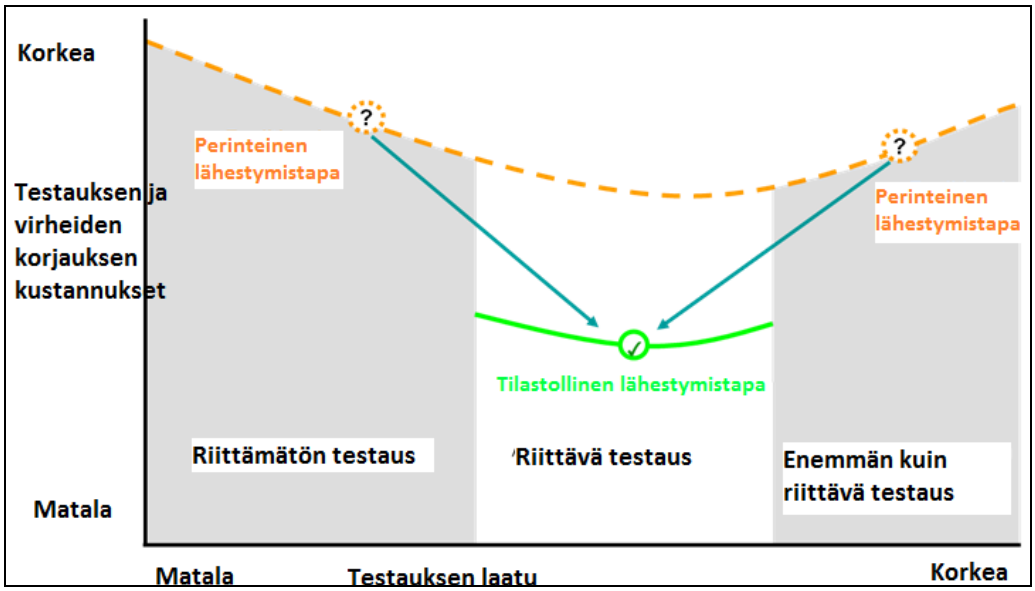
1. Poistetaan kelvottomat testitapaukset.
2. Valitaan jääneistä testitapaukset hyödyntäen valintatekniikkaa.
3. Luodaan puuttuvia tai laajempia testitapauksia.
4. Priorisoidaan testitapausten suorittaminen.

Testitapausten uudelleenkelpoistaminen on manuaalista toiminnallisuustestitapausten läpikäyntiä ja käyttökelvottomien testitapausten korjaamista tai poistamista joukosta [Holopainen 2005: 10]. Tämä prosessi on erittäin hankalaa, jos vanhoja testitapauksia ei ole päivitetty määrittelyjen muutosten jälkeen. Tässä tapauksessa testitapauksia on analysoitava uudelleen määrittelyjä vasten. Jos toiminnallisuustestauksen testijoukkoja ylläpidettiin koko ajan, silloin uudelleenkelpoistaminen ei vaadi analysointia ja testitapausten nimistä voi päätellä niiden kelvollisuuden.

Testitapausten valintatekniikoita on useita. Valintatekniikan tehtävä on löytää sellaiset sovelluksen alueet, joissa tehdyt muutokset ovat vaarallisimmat. Holopainen kuvaa kahden tyypin valintatekniikoita työssään: koodiin pohjautuvat ja määrityksiin pohjautuvat. Määrityksiin pohjautuvat valintatekniikat ovat saaneet enemmän huomiota viime vuosikymmenen aikana. [Holopainen 2005: 35.]

Toinen tapa jakaa valintatekniikat on valita kriteeriksi se, miten valintatekniikat päättävät testauksen riittävyydestä. Gao luettelee kolme tyyppiä: minimointitekniikat, kattavuuteen perustuvat tekniikat ja turvalliset tekniikat. Minimointitekniikat valitsevat mahdollisimman pienen määrän testitapauksista niin, että testaus olisi silti kattava. Kattavuuteen perustuvat tekniikat toimivat annetun kattavuuskriteerin perusteella. Nämä tekniikat valitsevat enemmän testitapauksia kuin minimointitekniikat, lisäämällä myös muutosten kannalta vaarallisimmat testitapaukset. Turvalliset tekniikat valitsevat ensin sellaiset testitapaukset, jotka paljastavat virheet muutosten jälkeen. [Gao 2003: 212–213.]

Regressiotestaukseen, jota käytetään tuotteen kelpoisuusmittarina hyväksymis-testaukseen, sopivat parhaiten kattavuuden perustuvat tekniikat, koska tässä tapauksessa on tärkeää arvioida tuotteen kokonaisuutena, eikä keskittyä yksittäisiin muutoksiin. Tässä insinöörityössä käsitellään Hexawise–työkalua esimerkkinä valintatyökalusta. Hexawise alentaa ajettavien testitapausten määrä säilyttämällä testauksen kattavuuden asteen samalla tasolla. Hexawise analysoi kaikki sovellukseen syötettävän datan yhdistelmät ja antaa tuloksena vain ne yhdistelmät, joiden ajaminen kattaa koko sovelluksen testausta, mutta sisältää mahdollisimman vähän toistoja. Työkalun käyttämä analyysi perustuu tilastotieteeseen. Työkalun harvennetulla testijoukolla voi edelleenkin löytää 100 % kaikista virheistä, joita löydetään tilanteessa, jossa kaikki yhdistelmät olisi käyty läpi testauksen aikana. Hexawisen kerrotaan vähentävän 30 – 40 % yhdistelmien määrästä (ks. kuva 10).



Kuva 10 Riittävän testauksen kustannukset Hexawise-työkalua hyödyntäen [http://hexawise.wordpress.com/ 5.1.2011]

Hexawisen tuottamat säästöt ovat vielä merkittävämpiä, jos niihen lasketaan mukaan sekä säästetty aika testitapausten suunnittelussa ja suorituksessa että myös duplikoitujen virheiden raportoinnissa ja analysoinnissa. Lisäasetusten avulla voidaan lisätä yhdistelmien määrä erityistä huomiota vaativilla parametreilla ja yhdistää parametrit toisistaan riippuviin ketjuihin. [http://hexawise.wordpress.com/ 5.1.2011]

Hexawise voi kuulostaa täydelliseltä valintatyökalulta, mutta silläkin on omat puutteensa. Hexawise toimii täydellisesti silloin, kun kaikki parametrit on syötetty

työkaluun analysoitavaksi. Tuskin kukaan enää testaa satojen syötteiden lomakkeet käymällä läpi kaikki parametrien mahdolliset yhdistelmät. Tavallisesti testaaja itse arvioi niiden tärkeyttä ja tekee valintoja määrittelydokumentaation perusteella. Satojen parametrien syöttäminen työkaluun ja niiden tarpeellisuuden arviointi voi viedä turhan paljon aikaa verrattuna siihen, kun testaaja itse tekee valintapäätöksiä.

Toiminnallisuustestauksen tavoitteet eroavat regressiotestauksen tavoitteista. Siitä johtuen pelkkä vanhojen testien hyödyntäminen ei riitä regressiotavoitteiden saavuttamiseksi. Tutkitussa projektissa sellaisia regressiotestausta varten luotuja lisätestitapauksia oli tehty loppukäyttäjien tekemistä skenaarioista. Käyttäjät kuvasivat skenaarioissa heidän todellisia tärkeimpiä työtehtäviään. Skenaariot yhdistävät monia testitapauksia ja tuottavat reaalikäytön näkökulman, josta on eniten hyötyä, kun valmistaudutaan hyväksymistestaukseen.

Priorisointiin kannattaa kiinnittää huomiota, jos regressiotestausta tehdään manuaalisesti. Jos resurssien määrä rajoittaa testausta, kannattaa kaikkein tärkeimmät testitapaukset suorittaa ensin. Priorisointia voidaan tehdä monella tavalla. Voidaan analysoida testauksen historiaa ja antaa korkeampi prioriteetti niille toiminnallisuusalueille, joissa oli havaittu eniten virheitä. Toinen lähestymistapa olisi toiminnallisuusalueen arvo käyttäjälle: mitä tärkeämpi alue käyttäjän tehtävien näkökulmasta sitä korkeampi prioriteetti. Muut lähestymistavat voivat olla toiminnallisuuden monimutkaisuus määrittelyiden analyysin perusteella, arkkitehtuurin monimutkaisuus jne.

5.3 Regressiotestauksen suorittaminen ja tulosten analysointi

Tässä luvussa kuvataan tilannetta, jolloin regressiotestausta käytetään siirtovaiheena toiminnallisuustestauksesta hyväksymistestaukseen. Tämä vaihe on kriittinen siinä mielessä, että hyväksymistestauksen epäonnistumisen jälkeen tuote palautetaan jatkokehitykseen, joka yleensä rikkoo projektiaikataulua ja on erittäin kallista. Ohjelmistoprojekteissa käytetään useita tapoja varmistaa tuotteen kelpoisuutta hyväksymistestaukseen – regressiotestauksen ajaminen on yksi niistä.

Insinöörityön pohjaksi tutkitussa projektissa tämä regressiotestauksen käyttötapa osoittautui todella tehokkaaksi. Se oli yksi eniten vaikuttaneista osatekijöistä

hyväksymistestauksen onnistumisessa. Regressiotestaukseen osallistuivat sekä testaustiimin jäsenet että asiakkaat. Asiakkaiden löytämät virheet ja antamat huomautukset käytiin läpi ja osa niistä korjattiin ennen hyväksymistestausta. Regressiotestauksen suorittaminen järjestettiin kierroksissa kahdesti ennen sovelluksen eri tuotantoversioiden luovuttamista hyväksymistestaukseen. Regressiokierrosten ajaminen on erityisen hyödyllistä, jos sovellus on laaja ja sen rakenne on monimutkainen, jolloin automaattisia skriptejä käytetään vain perusominaisuuksien testaamisessa.

Regressiokierroksia ajettiin kolme ja kierroksien välillä oli 2 - 3 viikon tauot. Taukojen aikana kehitystiimi korjasi virheitä ja valmisti uuden sovelluksen version. Yhden kierroksen kesto oli 2 viikkoa ja siihen osallistuivat sekä testaustiimin jäsenet että loppukäyttäjät. Ensimmäisessä kierroksessa testattiin kaikki regressioon valitut testitapaukset ja skenaariot. Testitapaukset oli priorisoitu siltä varalta, ettei kaikkia tapauksia ehditä testaamaan. Ensimmäisellä kierroksella testaustiimi ajoi toiminnallisuustestauksesta valittuja testitapauksia, ja loppukäyttäjät testasivat skenaariot. Kierroksen aikana saatuja virheraportteja analysoitiin ja lähetettiin korjattavaksi.

Projektikokemusten perusteella regression toinen kierros kannattaa aloittaa vasta silloin, kun kaikki korjaukseen lähetetyt virheet on korjattu ja validoitu. Toisen kierroksen aikana ajettiin ensin kaikki edellisen kierroksen hylätyksi tulleet testitapaukset ja sen jälkeen keskityttiin skenaarioihin. Toisen kierroksen tuloksena saimme vähennettyä sekä hylättyjen testitapausten määrää että kokonaisvirheiden määrää. Uusien virheiden määrä ja myös useimmiten niiden vakavuus oli pienempi.

Viimeisellä kierroksella käytiin kaikki skenaariot vielä kerran läpi. Loppukäyttäjät ja testaajat testasivat sovellusta rinnakkain, ja siitä saadut testitulokset analysoitiin erillään. Skenaarioiden lisäksi käytettiin tutkivaa testausta, jolloin testaajat vapaasti testasivat heidän valitsemiaan alueita. Tutkiva testaus paljastaa sellaisia virheitä, joita ei löydetä testitapausten suorittamisen aikana. Sitä suorittavat testaajat, jotka tuntevat parhaiten testattavan sovelluksen. Tutkiva testaus on tehokas tapa löytää virheet vain silloin, kun testaaja ymmärtää sen tavoitteet ja tunnistaa sovelluksen potentiaalisia virheitä ja ongelmallisia toiminnallisuuksia. Jos nämä ehdot eivät täyty, silloin tutkivasta

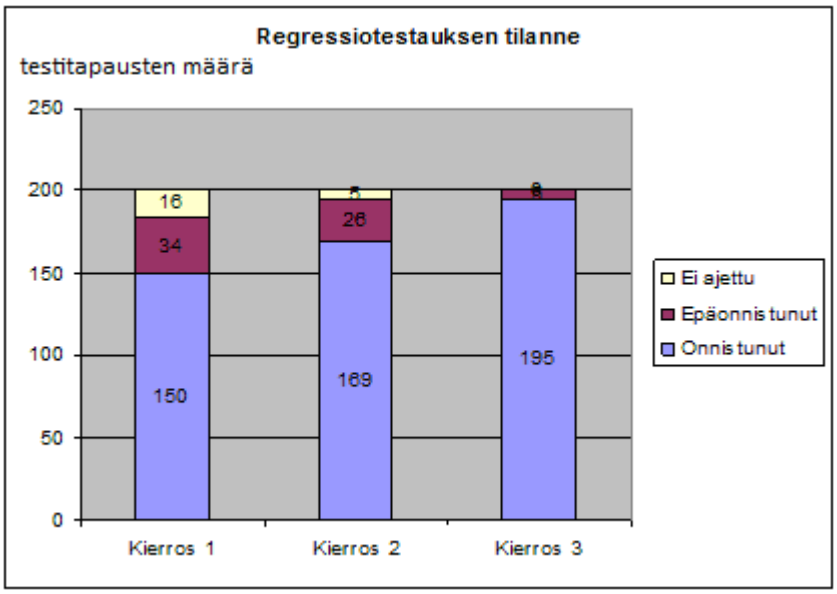
testauksesta voi tulla järjestämätön kokeilu, josta ei ole mitään hyötyä. Viimeisen kierroksen jälkeen löydettyjen virheiden korjausta jatkettiin ja sovellus siirrettiin hyväksymistestaukseen.

Organisoinnin kannalta on hyvä nimetä ainakin yksi testaaja seuraamaan virheraportteja ja analysoimaan niitä, tarkistamaan lokitiedostoja ja ilmoittamaan muutoksista tai virheistä muille. Regressiokierroksiin osallistuvat monet työntekijät, joten loppukäyttäjien osallistumisesta on sovittava etukäteen asiakkaan kanssa ja testaajien osallistuminen regressioon on huomioitava projektin aikataulussa. Regressiokierrosten aikana testausympäristön ja testattavan sovellusversion on oltava hyvin valmistettuja, jottei aika kuluisi odottamiseen tai turhien raporttien laatimiseen.

Regressiokierrosten tulosten ja kulun analysoinnin jälkeen tehtiin seuraavat johtopäätökset:

1. Raportoitujen virheiden analysointi kannattaa tehdä useammin kuin kerran viikossa kaikkien regressioon osallistujien kesken. Silloin voidaan käydä läpi tärkeimmät virheet ja selittää, mihin toiminnallisuuksiin ne vaikuttavat. Näin voidaan välttää suurin osa toistuvista virheraporteista ja parantaa erityisesti loppukäyttäjien laatimien raporttien laatua.
2. Kaikille osallistujille on heti ilmoitettava testausympäristössä tapahtuvista katkoista tai vakavimmista virheistä. Näin vältetään sekä turhia virheraportteja että loppukäyttäjille muodostuvaa mielipahaa sovelluksesta, kun heistä tuntuu, ettei mikään toimi.
3. Raportoidut virheet kannattaa ryhmittää alueittain ja ohjeistaa niiden korjauksen seurauksista. Esimerkiksi kosmeettisen virheen huoleton korjaus voi tuottaa uuden vakavan virheen sovelluksessa tai kaksi eri kehittäjien tekemää rinnakkaista korjausta voivat olla ristiriidassa ja rikkoa toiminnallisuuden.

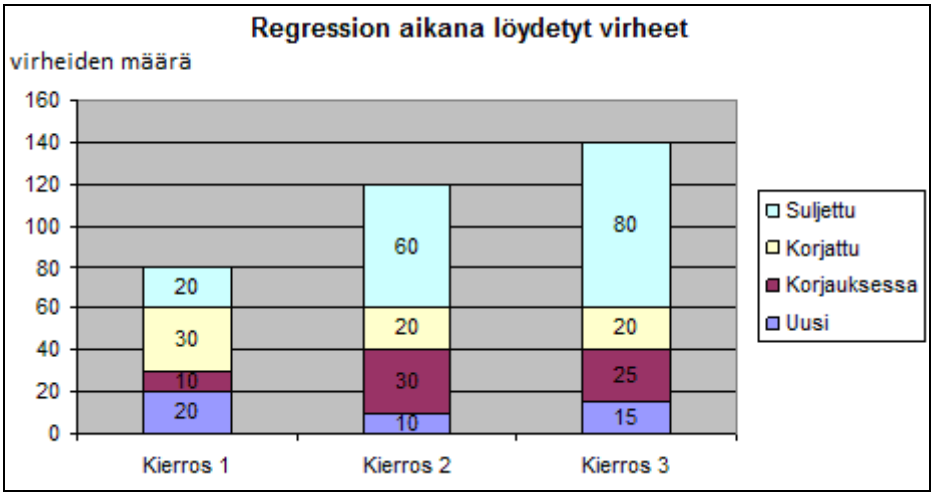
Regressiotestauksen tulokset raportoidaan kattavuudella ja virheiden analyyseillä. Kattavuutta voi tarkastella eri kierrosten välillä. Kuvassa 11 esitellään testitapausten ajotulokset jokaisen kierroksen ajamisen jälkeen.



Kuva 11 Regressiotestauksen tilanne kierrosten jälkeen

Kuvan 11 avulla voidaan huomata, että 100 % kattavuuden saavutettiin kolmannen kierroksen aikana. Läpipäässeiden testitapausten määrä kasvoi kierroksesta toiselle, mitä todentaa korjausprosessin onnistumisen, vaikkei kaikkia testitapauksia pystytty suorittamaan onnistuneesti.

Regression aikana löydetyt virheet voidaan analysoida sekä niiden tilan perusteella että niiden vakavuudella. Projektissa huomattiin, että kolmannen kierroksen aikana löydettyjen virheiden vakavuus oli pienempi kuin ensimmäisen kierroksen virheillä. kuvassa 12 näytetään regression aikana löydettyjen virheiden tilat eri kierrosten lopussa.



Kuva 12 Regression aikana löydettyjen virheiden tilat

Kuvasta 12 näkyy, että virheiden korjaus pitää jatkaa regression jälkeenkin. Kuitenkin suurin osa virheistä on ehditty korjaamaan ja testaamaan uudelleen regression aikana.

6 Toiminnallisuus- ja regressiotestauksen automatisointi

Tässä luvussa ensin perustellaan tarve automatisoinnille ja kerrotaan automatisointityökalujen kehittymisestä. Sen jälkeen keskitytään viimeisen sukupolven mallipohjaisiin työkaluihin, joista kuvataan Conformiq Tool Suite™ -työkalu. Tämän työkalun kuvauksen pohjalta käydään läpi testausprosessin muuttuminen työkalun käytön seurauksena.

6.1 Testausprosessin automatisoinnin tarve

Testauksen automatisoinnilla on yksinkertainen syy. Testauksen resurssit määritellään projektin alkuvaiheessa. Näitä resursseja ei välttämättä lisätä kehityksen edistyessä ja testattavan sovelluksen toiminnallisuuksien määrien kasvaessa. Käytännössä tämä tarkoittaa, että testaajat, jotka tekevät manuaalista testausta, eivät pysty kattavasti testaamaan sovellusta sen kasvettua tiettyyn pisteeseen. Ainoa tapa selviytyä tästä tilanteesta ja säilyttää testauksen tarpeeksi korkea kattavuusaste on testausprosessin automatisointi. Automatisoinnilla tarkoitetaan sitä, että testaajat luovat testitapausten skriptikirjastot, joita he ylläpitävät. Kun uusi toiminnallisuus lisätään mukaan testaukseen, testaajan on suunniteltava uudet skriptit testitapauksille. Yksi selkeimmistä eroista manuaalisen ja automatisoidun testauksen välillä on testitapausten suorituksessa: automatisoidut skriptit eivät vaadi testaajan jatkuvaa osallistumista niiden ajamiseen. Manuaalisessa testauksessa testitapausten suorittaminen vie suhteensa eniten aikaa. [Lewis 2005: 293–294.]

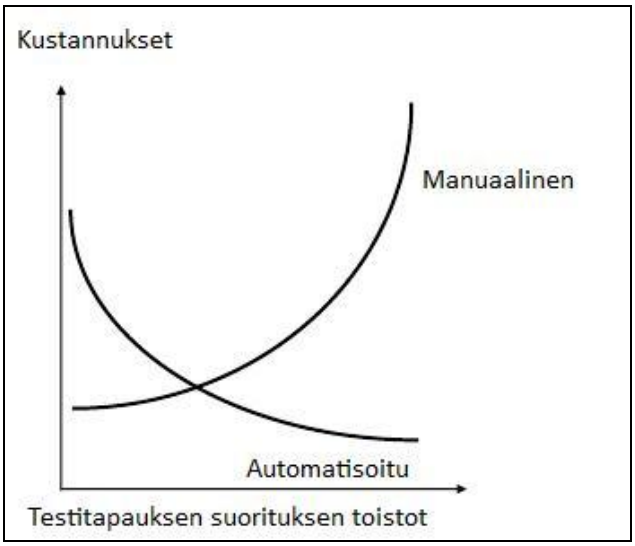
Testauksen automatisointi edellyttää testaustyökalun käyttöönottoa tai oman testausympäristön luomista. Testauksen automatisointityökalujen tarkoitus on automatisoida mahdollisimman paljon toistuvia testauksen tehtäviä. Sen lisäksi työkalu voi tallentaa ja järjestää suuriakin määriä dataa. Ei kuitenkaan ole työkaluja, joiden avulla olisi mahdollista automatisoida kaikkea; jokaisella työkalulla on oma tarkoituksensa. Testauksen automatisointi edellyttää testausprosessin kypsyyttä. Työkalut itsestään eivät lisää testauksen kypsyyttä, ne vain tukevat testauksen ja sen kypsyyden tasoa [Hass 2008: 361]. Testauksen automatisointi kuuluu yhteen

tavoitteeseen TMM:n tasolla 5. Taso 5 on TMM-mallin ylin taso, ja käytännössä se tarkoittaa, että ennen automatisointia testausprosessin on oltava suunniteltu, määritelty ja organisoitu. Tästä seuraa se, että ei-kypsä testausprosessi saa automatisoinnista vain lisää ongelmia, kun taas kypsä testausprosessi pystyy hyötymään eniten automatisoinnista.

Ennen testauksen automatisointia on tarpeen varmistaa monta asiaa, joista automatisoinnin onnistuminen on kiinni.

- Testauksen automatisointi kannattaa aloittaa vasta silloin, kun sovellus on vakaa ja toiminnallisuudet ovat loppuun määriteltyjä. Muuten testiskriptien ylläpito (muuttaminen määrittelyjen tai sovelluksen toteutuksen mukaisesti) vie paljon resursseja, ja testaustulos voi jäädä epäselväksi.
- Testaussuunnitelma on oltava tehty ja testausprosessi määritelty.
- Testaussuunnitelmassa automatisointiin valittujen testitapausten pitää olla yksiselitteisiä. Automaattisten testitapausten ajotulokset arvioivat koneet, jotka eivät pysty ajattelemaan, vaan tekevät täsmälleen, mitä skriptissä lukee.
- Testausympäristö ja automatisointityökalu tulee asentaa ja niiden toimivuus tulee tarkistaa etukäteen.
- Testaajilla, jotka suunnittelevat ja toteuttavat automatisoinnin, pitää olla positiivinen kokemus automatisoinnista ja hyvät ohjelmointitaidot.
- Automatisointiin on oltava määriteltynä oma budjetti, koska rahaa ja aikaa kuuluu etenkin automatisoinnin alkuvaiheessa ja työkalujen hankkimiseen. Koulutuksen tarve on myös suunniteltava, jos testaustiimissä on jäseniä, joilla on vähän kokemusta automatisoinnista. [Naik & Tripathy 2008: 391–392.]

Automatisoinnin on tuotettava lisäarvoa projektiin. Kuvasta 13 nähdään, että automatisoinnin kustannukset laskevat nopeasti automatisoidun testitapauksen toistokertojen kasvaessa.



Kuva 13 Testauksen suorittamisen kustannus/voitto kaavio [Hass 2008: 364]

Toisaalta, jos testitapaus suoritetaan vain muutaman kerran, sen suoritustavaksi kannattaa valita manuaalinen tapa. Automatisointi pystyy mahdollisesti tuottamaan muitakin hyötyjä kuin ajan säästöä:

- Testaajien tuottavuus paranee. Kerran suunniteltuja testiskriptejä suoritetaan jatkuvasti, jolloin testaajilla vapautuu aikaa testiskriptien laadun parantamiseen ja analysointitehtäviin.
- Regressiotestauksen kattavuus kasvaa, koska samassa aikavälissä ehditään suorittamaan enemmän automatisoituja testitapauksia verrattuna manuaalisesti suorittaviin testitapauksiin.
- Automatisoidut testitapaukset uudelleenkäytetään jokaisen ajon aikana, mikä säästää resursseja ja tehostaa testausprosessia.
- Testauksesta tulee yhtenäinen prosessi, joka ei riipu testaajan kokemuksesta. Testiaskeleet ajetaan aina samassa järjestyksessä ja virheiden toistuminen ei aiheuta ongelmia. Kehittäjien on helpompi löytää virhettä aiheuttanut tilanne.
- Testikierrosten väliaika vähenee. Se tarkoittaa, että kehittäjät saavat nopeammin palautetta toteutuksen laadusta ja voivat aikaisemmin puuttua ongelmallisiin toiminnallisuuksiin ja korjata virheet. Näin korjaussykli lyhenee ja projektin aikataulua tulee helpoimmin ennustaa.
- Sovelluksen ylläpitokustannukset vähenevät, koska automatisoituja testijoukkoja nopeuttavat sovelluksen testauksen myös ylläpidon aikana. Toisaalta korkeampi testauksen kattavuusaste auttaa löytämään enemmän

virheitä ja mahdollistaa niiden korjaamisen ennen tuotteen siirtämistä ylläpitovaiheeseen.

- Testauksen tehokkuus paranee, koska samassa aikavälissä ehditään suorittamaan enemmän testitapauksia ja testauksen kattavuusaste kasvaa. [Naik & Tripathy 2008: 391–392.]

Yllämainitut automatisoinnin tuottamat hyödyt parantavat testausprosessia ja tekevät automatisoinnista vielä kannattavampaa investointia.

6.2 Testauksen automatisoinnin kehitysvaiheet

Testauksen automatisointi alkoi 1980-luvulla. Ensimmäiset työkalut toimivat staattisella nauhoittamisperiaatteella ilman mitään skriptauskielten käyttöä. Ne tallensivat testaajan jokaisen toiminnon ja sitten toistivat niitä samassa järjestyksessä. Sellaiset työkalut eivät sietäneet mitään muutoksia sovelluksessa ja hälyttäneet virheitä jokaisesta poikkeamasta. Työkalut aiheuttivat isoja kustannuksia skriptien ylläpidossa, koska niitä yksinkertaisesti piti tallentaa uudestaan jokaisen sovelluksen muutoksen jälkeen. [Lewis 2005: 291.]

Seuraavaksi ilmestyivät nauhoittamistyökalut, joita mahdollistivat skriptauskielten käyttöä. Skriptit muuttuivat ohjelmiksi, joita testaajat voivat muokata lisäämällä ehtoja, poikkeustilanteita, jne. Siitä seurasi samanlaisia ongelmia kuin muidenkin ohjelmien kanssa – ne sisälsivät virheitä ja vaativat testaajilta ohjelmointitaitoa. Seuraavalla automatisoinnin tasolla ilmestyivät työkalut, jotka sisälsivät muuttujia. Se tarkoittaa, että skriptiin ei tarvinnut tallentaa staattista arvoa, vaan arvo haetaan skriptin ajonaikana konfigurointitiedostosta. Tätä kehitysvaihetta varten testaaja suorittaa manuaalisesti testitapauksen, josta työkalu tallentaa skriptin. Tämän jälkeen testaaja muokkaa skriptin korvaamalla staattiset arvot muuttujilla. Muuttujien arvot tallennetaan erilliseen tiedostoon, joka on tarvittaessa helppo päivittää sovelluksen muuttuessa.

Skriptausmenetelmät kehittyivät työkalujen rinnalla. Lewis mainitsee kaksi tekniikkaa, joiden avulla skriptien kirjoittaminen helpottui. Ensimmäinen on toiminnallisuuksien hajottaminen. Testitapaukset hajotaan perustoiminnallisuuksiin, joista luodaan skriptit. Toinen tekniikkaa käyttää avainsanoja, joiden avulla kontrolleriskripti lukee toimintoja ja tarkistaa tulokset. Avainsanatekniikka on helppo oppia ja käyttää uudelleen, ja se

mahdollistaa sovelluksen testauksen automatisoinnin muutamassa päivässä. [Lewis 2005: 297.]

Nauhoittamistyökalut vaativat testaajalta paljon manuaalista työtä: ensin skriptin nauhoittamista ja sen jälkeen testidataa sisältävien tiedostojen päivittämistä ja jopa skriptien nauhoittamista uudelleen, jos sovelluksessa tapahtunut muutos on liian iso ja uudet toiminnallisuudet eivät vastaa enää vanhaa skriptiä. Testitapausten luominen määrittelydokumenteista on manuaalista työtä, ja toisaalta oikean testidatan luominen on hankalaa ja aikavaativa prosessi. Väärin suunniteltu testidata voi jättää testaamatta osan syötteiden yhdistelmistä ja näin jättää osan virheistä löytämättä. Aikaisemmat versiot nauhoittamistyökaluista eivät ottaneet kantaa testidataan, jota käytetään skriptiä ajettaessa. Näitä työkaluja voidaan pitää niin hyvinä kuin niihin syötettävää testidataa.

Viimeisen sukupolven testaustyökalut generoivat itse määrittelyihin pohjautuvasta mallista testitapaukset (engl. Model Based Testing, MBT). Mallissa kuvataan sovelluksen käyttäytyminen kaavioiden ja skriptien avulla. Malli ei ota kantaa siihen, miten toiminnallisuudet pitää toteuttaa. Se vain selittää, mitä sovelluksen on tehtävä tietyissä tilanteissa. Mallit luodaan merkkijärjestelmien avulla, jotka perustuvat tilojen, siirtymien tai toimintojen kuvauksiin. Malli on pelkistetty kuvaus sovelluksesta. Työkalut auttavat verifioimaan mallin ja jopa animoimaan sen toiminnan, ja auttavat näin testaajaa ymmärtämään miten luotu malli toimii. [Utting & Legeard 2007: 27–28.]

Mallin luomisen jälkeen testausprosessi on automatisoitu. Testitapaukset saa generoitua mallista valitsemalla tarvittavien testitapausten määrän, tarkoituksen, jne. Osa työkaluista pystyy generoimaan myös testiskripteihin tarvittavan testidatan. Testitapaukset sisältävät sekä syötteet että odotetun sovelluksen käyttäytymisen, jota kutsutaan oraakkeliksi. Testitapaukset peilaavat mallia ilman mitään tietoa sovelluksen toteutustavasta, jolloin testaus noudattaa black box -periaatetta. Generoidut testitapaukset pitää konkretisoida eli yhdistää abstraktit testitapaukset sovellukseen. Tämä vaihe automatisoidaan luomalla adapteri, joka hoitaa jatkossa yhdistämiset. Adapteri muuttaa testitapaukset skripteiksi, jonka jälkeen ne ovat ajettavissa. Testitulokset saadaan analysoitua samassa työkalussa, joka laskee jokaisen ajon halutut metriikat. Testaajalle jää tehtäväksi paikantaa ja raportoida virheet.

Määrittelyjen muututtua testaaja päivittää mallin, jonka jälkeen kaikkien testitapausten päivitys vie yhden näppäimen painalluksen. [Utting & Legeard 2007: 6-8.]

Mallipohjaiset työkalut vievät automatisoinnin periaatteen ylimmälle tasolle. Automatisoitu testitapausten suunnittelu takaa niiden objektivisuuden ja halutun kattavuuden asteen. Työkalut yksinkertaistavat testijoukkojen ylläpidon – mallin tunteva testaaja päivittää sen taatusti paremmin ja nopeammin verrattuna vanhaan tapaan, kun hänen piti manuaalisesti löytää kaikki testiskriptit, joita muutos koskee, ja päivittää ne. Mallin luominen paljastaa sellaiset määrittelyt, joiden testattavuutta on mahdotonta kuvata. Testaajat tuntevat paremmin määrittelyt ja pystyvät kommunikoimaan mallin avulla määrittelijöiden ja toteuttajien kanssa.

Mallipohjaisilla työkaluilla on rajoitteensa. Työkalut saattavat jättää virheet löytämättä, mikä on jokaisen testaustavan huomioon otettava oletus. Mallipohjaiset työkalut vaativat testaajien kouluttamista ja testausprosessin ydinpohjaista muuttamista. Mallin luominen voi tuntua monimutkaiselta prosessilta verrattuna manuaaliseen testaukseen, koska testaajan pitää osata sekä ohjelmoida että piirtää kaavioita. Mallipohjaista testausta käytetään toiminnallisuus- ja regressiotestauksessa. Tarvittaessa samat skriptit voi ajaa myös kuormitustesteinä. Mallipohjainen testaus ei kata edes koko järjestelmätestausta. Sen lisäksi sovelluksessa voi löytyä toiminnallisuuksia, joiden mallintaminen on mahdotonta tai liian monimutkaista. Näiden toiminnallisuuksien testausta tehdään manuaalisesti. Mallipohjaisen testauksen aikana esiin tulee muitakin ongelmia. Esimerkiksi vanhentunut malli tuottaa testitapauksia, jotka eivät ole enää yhteensopivia sovelluksen kanssa. Automaattisesti generoidut testitapaukset löytävät paljon virheitä, joista osa on adapterin tai mallin omia virheitä. Mallipohjaisten työkalujen löytämien virheiden analysointi vie aikaansa, jonka tuloksena voidaan joutua korjaamaan mallia tai adapteria. [Utting & Legeard 2007: 54–55.]

6.3 Nauhoitettujen skriptien käyttö toiminnallisuus- ja regressiotestauksessa

Testiskriptit ovat useimmiten ohjelmoituun muotoon muutettuja manuaalisia testitapauksia. Testausympäristön luominen ja testiskriptien kirjoittaminen puhtaalta pöydältä on harvinainen lähestymistapa, koska se on erittäin vaikeaa ja kallista. Markkinoilla on kymmeniä avoimesti jaettavia ja myös maksullisia työkaluja, joiden käyttö ei vaadi suuria ohjelmointitaitoja testaajilta. Ne toimivat nauhoittamis-

periaatteella, eli testaaja suorittaa toiminnot testattavassa sovelluksessa ja työkalu nauhoittaa siitä toistettavan skriptin. Nauhoitetut skriptit ovat sen jälkeen ajettavissa ilman testaajan erillistä osallistumista. Testiskriptit nauhoitetaan työkaluilla ja ne yhdistetään kokonaisuuksiin joko työkalussa tai automatisoidussa ympäristössä. Nauhoitetut testiskriptit vaativat testitapauksen kulun tarkan suunnittelun. Se tarkoittaa, että ensin luodaan sanallisesti kuvattuja testitapauksia. Suunnittelu vie silloin paljon aikaa, jotta saadaan toimivat testausympäristö ja testiskriptit. Myös testidatan keksiminen ja tallentaminen vie aikansa. Nauhoittaminen on aina suunniteltava etukäteen, ja työkalujen hyvä osaaminen on välttämätöntä.

Nauhoitustyökaluja on monenlaisia. Yksi avoimista työkaluista on selainpohjainen työkalu nimeltään Selenium. Se simuloi reaaliajassa sovelluksen ajamista web-selaimessa. Testaaja suorittaa käyttäjän toiminnot Firefox-selaimessa ja Selenium nauhoittaa skriptin, joka sen jälkeen voidaan ajaa. Sen lisäksi testaaja voi itse luoda testiskriptit HTML-taulukoissa, joita Selenium tulkitsee skripteiksi. Selenium tukee testitapausten paketointia kokonaisuuksiin. Integroimalla Selenium CruiseControl sovelluskehykseen, joka automatisoi kokoamisprosessin, saadaan aikaiseksi automatisoitu testipakettien ajaminen. Seleniumin huonoja puolia on sen käyttöliittymäpainotteisuus, jonka vuoksi loogiset ja esim. tietokantaoperaatiot on testattava jollain muulla työkalulla. [Holmes & Kellog: 1–6.]

Nauhoitetut skriptit ovat suorituksen aikana objektiivisia ja toistavat toiminnot tarkasti skriptin mukaan. Mutta, jos jotain tarkistusta ei ole huomioitu skriptissä, sitä ei kukaan enää testaa ja näin osa toiminnallisuudesta voi jäädä testaamatta. Silloin suunnitteluvaiheen katselmointiin on panostettava. Käsin kirjoitetut skriptit ovat myös virhealttiita: nekin ovat ohjelmoituja ja voivat sisältää virheitä. Erilaiset ongelmat työkalujen ja ympäristön kanssa on ratkaistava ennen kuin varsinaisia tuloksia saa aikaiseksi.

Nauhoitetut testiskriptit säästävät aikaa testien suorituksessa. Testiskriptit ajetaan useimmiten yöaikana, jolloin testaajat pystyvät analysoimaan tulokset seuraavana aamulla. Virheiden raportointi vaatii virheen paikallistamista, koska nauhoitettujen skriptien suorittaminen tuottaa myös turhia hälytyksiä. Esimerkiksi näppäimen nimen muutos tai kielivirhe sen nimessä voi aiheuttaa monien testitapausten

epäonnistumisen, vaikka ongelma ei kokonaisuudessa olekaan vakava. Nauhoitettu skripti ei tunnista uutta nimeä ilman skriptin päivittämistä. Tästä seuraa tietty kriittisyys nauhoitettujen skriptien suorituksen tulosten suuntaan. Virheraporttien laatiminen nauhoitettujen testien tuloksista on haastavaa ja vaatii lisää tutkimista, joka on tehtävä käsin.

Regressiotestausta voidaan automatisoida esimerkiksi käyttämällä valmiita nauhoitettuja skriptejä toiminnallisuustestauksesta. Jos toiminnallisuustestaus oli tehty manuaalisesti, se tarkoittaa käytännössä, että regressiotestaustakin tehdään manuaalisesti. Regressiotestauksen aikana on jo myöhäistä ottaa käyttöön uusia työkaluja ja ympäristöjä.

Vaikka kaikkien nauhoitettujen skriptien käyttäminen regressiossa näyttäisi houkuttelevalta, kannattaa silti käyttää uudelleenkelpoistamista ja valintateknikoita. Ison testijoukon regressioskriptien ajaminen voi tuntua helpolta, koska sitä tekevät tietokoneet, mutta tulosten analysointi on manuaalista työtä. Jos testijoukossa on ollut kelvottomia tai toistuvia testiskriptejä, virheiden määrä on paljon isompi ja niiden analysointiin tuhlautuu aikaa.

6.4 Automatisoitu testien suunnittelu Conformiq Tool Suite™ -työkalun avulla

Conformiq Tool Suite™ on mallipohjainen työkalu, jonka avulla automatisoidaan testitapausten suunnitteluprosessi. Työkalu yhdistää vaatimukset, määrittely-dokumentit, mallin, testaussuunnitelmat ja automaattisesti suoritettavat testitapaukset yhteen täysin automatisoituun prosessiin nimeltä Automated Test Design™. Työkalun ydin tunnistaa käyttäjän luomaa sovelluksen mallia. Ytimen lisäksi työkaluun on integroitu black box –testausperiaatteen metodologiat, vaatimusvetoinen testaus (engl. requirements-driven testing), ehtokattavuus ja raja-arvo analyysi. Nämä testauksen tekniikat huomioidaan automaattisesti testitapausten suunnittelussa Conformiq Tool Suite™ –työkalun avulla. Testitapausten generointialgoritmi ei tuota satunnaisia testitapauksia. Työkalu tuottaa optimoidun testausjoukon, joka vastaa käyttäjän valitsemia metriikoita ja testauksen tavoitteita. Testitapausten lisäksi työkalu valmistaa jäljitettävyysmatriisin ja testitapausten riippuvuutta kuvaavat taulukot, joiden avulla testausprosessia analysoidaan. [http://www.conformiq.com/faq.php 12.2.2011.]

Työkalun käyttö alkaa mallin luomisesta. Testaajat eivät suunnittele enää testitapauksia ja testidataa, vaan analysoivat määrittelydokumentaatiota ja luovat sen pohjalta mallin, joka kuvaa valitulla abstraktitasolla, miten sovelluksen pitää toimia. Testaaja voi itse valita, millä tarkkuustasolla hän tekee mallin. Näin hän vaikuttaa tulevaan testaussuunnitelmaan ja testitapausten sisältöön. Mallia luodaan Java-syntaksia käyttävällä mallinnuskielellä, joka voidaan täydentää tilakaavioilla. Mallinnuskieli tunnistaa olio-ohjelmoinnin käsitteitä: luokka, luokat, periytyminen, säikeet ja asynkronoitu kommunikointi mallin säikeiden välillä. Olemassa olevat mallit voi tuoda IBM Rational, Rhapsody, Sparx system Enterprise Architect, HP Quality Center, IBM RequisitePro ja DOORS -työkaluista. Conformiq Tool Suite™ sisältää oman mallinnustyökalun Conformiq Designer, joka on rakennettu Eclipse®-kehitysympäristön päälle. Kun malli on valmis, työkalu analysoi sen ja tuottaa sekä testitapauksia sisältävän testaussuunnitelman että syötteet ja odotetut tulokset. Työkalu osaa generoida testidatan analysoimalla mallissa kuvatut rajapinnat. Testitapaukset tuotetaan sanallisessa (HTML) ja skriptimuodossa. Testiskripteille voidaan valita skriptauskieliksi Visual Basic, TCL, Perl, Java, Python, C++, XML. Sanalliset testitapaukset voi luovuttaa asiakkaalle testaussuunnitelmana. [http://www.conformiq.com/features.php 12.2.2011.]

Mallin valmistamisen jälkeen kaikki testaajan tehtävät ovat yksinkertaisia. Testitapausten generointi testidatan kanssa vie yhden napin painalluksen. Sen jälkeen testitapaukset voidaan viedä automatisoituun testausympäristöön, jossa ne suoritetaan. Testaajan on ohjelmoitava liittymä tai adapteri testiympäristön ja työkalun välillä, jotta testitulokset saadaan takaisin työkaluun analysoitavaksi. Tilanteessa, jossa sovelluksen määrittelyt muuttuvat, testaajan on päivitettävä vain malli ja sen jälkeen käskeä työkalu generoimaan uudelleen testijoukot. Työkalu esittelee testauksen tuloksia ja auttaa analysoimaan niitä.

Työkalun tuottajan sivuilla sanotaan, että työkalua käyttävät asiakkaat raportoivat 400 % tuottavuuden kasvua. Monet yritykset käyttävät säästyneet resurssit edelleen prosessin kehitykseen tai siirtävät testauksessa toimineet työntekijät muihin tehtäviin. Pääedut työkalun käytöstä:

- Parempi testauksen kattavuus saavutetaan poistamalla satunnaiset testaajan virheet testitapausten luomisessa tai suorituksessa.

- Testitapauksen suunnittelu vie keskimäärin 5 kertaa vähemmän aikaa, vaikka vaatimusten kattavuus testeillä kasvaa.
- Testauksen yleislaatu kasvaa, koska testitapaukset ovat yhdenmukaisia.
- Työkalu automaattisesti generoi jäljitettävyysmatriisin ja analysoi sen.
- Testijoukkojen ylläpito yksinkertaistuu, koska testaajan ei tarvitse enää päivittää testitapauksia vaan kaikki muutokset tehdään suoraan malliin.
- Mallin avulla voidaan löytää uusia kommunikointitapoja sovelluksen suunnittelijoiden kanssa. [http://www.conformiq.com/faq.php 12.2.2011.]

Vaikka Conformiq Tool Suite™ -työkalun valmistaja lupaa aika paljon, kannattaa olla riittävän kriittinen lupausten suhteen. Esimerkiksi vanhat virheet, joita testaajat tekivät testitapausten suunnittelussa siirtyvät nyt mallin luomiseen. Conformiq Tool Suite™ ei voi keksiä testitapauksiin mitään ylimääräistä, mitä ei annettu mallissa. Jos mallissa puuttuu joku rajoitus tai ehto, sitten ne puuttuvat kaikissa automaattisesti suunnitteluissa testitapauksissa.

Toinen ongelma on se, että jos testaaja haluaa manuaalisesti tehdä muutokset skripteihin, automaattinen päivitys ei ota niitä myöhemmin huomioon. Tällöin tulee joko luopua kokonaan automaattisesta päivityksestä tai tehdä muutokset käsin joka kerta, kun testijoukon uusi versio valmistuu.

Kolmas potentiaalinen ongelma on koulutus ja testausprosessien uudelleenmäärittely. Conformiq Tool Suite™ on sen verran innovatiivinen, että vanhat testausprosessit eivät välttämättä sovi sen käyttöön. Testaajien on opittava suunnitteluprosessi ja unohdettava kaikki perinteiset lähestymistavat toiminnallisuustestauksessa. Tämä ei käy projekteille, joissa testaus on jo suunniteltu. Jos jostain syystä työkalun käyttö epäonnistuu tai osoittautuu olevan epäsopiva projektisovelluksen kanssa, testaustiimi jää pulaan, koska silloin toiminnallisuustestauksen suunnittelu tulee aloittaa uudelleen. Käytännössä se voi tarkoittaa koko testauksen epäonnistumista tai vähintään aikataulusta myöhästymistä.

Conformiq Tool Suite™ työkalun kuvauksesta on helppo huomata, että se on tarkoitettu toiminnallisuus- ja regressiotestaukseen. Kaikki muut järjestelmätestauksen tyypit kuten suorituskyky-, turvallisuus- ja käytettävyystestaus jäävät sen ulkopuolelle.

Toisaalta on vaikeaa edes kuvitella mallin tarkkuutta, jotta testitapauksiin tulisi tieto käyttöliittymän komponenttien asettelusta. Todennäköisesti nämä käyttöliittymän testit myös jäävät työkalun ulkopuolelle. Kaiken kaikkea Conformiq Tool Suite™ ei ratkaise monia rajoituksia, joita on ollut muillakin automatisoinnin työkaluilla.

6.5 Automatisoinnin arviointi

Automatisoinnin valitseminen on useimmiten järkevä tapa tehostaa testausprosessia. Pelkkä manuaalinen testauksen suorittaminen vie niin paljon resursseja, että se on harvinainen ratkaisu nykyään. Suosittu vaihtoehto on testauksen osittainen automatisointi. Ensisijaisesti automatisoidaan sellaiset testit, joiden suorittaminen manuaalisesti on joko mahdotonta tai vie liian paljon aikaa. Sellaisia ovat stressi- ja suorituskykytestaus sekä regressiotestaus ja esitestaus (engl. smoke testing). Muiden testausvaiheiden automatisointi voi viivästyä ongelmien vuoksi.

Työkalun valinta on ydin kysymys testauksen automatisoinnista. Ongelmat työkalujen kanssa johtuvat monista syistä. Yleisin syy ovat epärealistiset odotukset. Työkalut eivät ratkaise olemassa olevia ongelmia, niiden tarkoitus on tehostaa prosessia. Vaikka myyntiedustaja lupaakin merkittäviä säästöjä ja ihmeellisiä testauksen uusia ratkaisuja, käytännössä asiat voivat mennä eri tavalla. Seuraavaksi kuvataan Lewisin luettelemat tyypillisimmät ongelmat.

Ensimmäinen ongelma on se, että kerran automatisoitujen testitapausten oletetaan toimivan sen jälkeen aivan automaattisesti. Automatisoidut skriptit vanhenevat samalla nopeudella kuin sovelluksen kehitystyö edistyy. Vanhat skriptit tulee päivittää ajan tasalle ja uusia skriptejä pitää luoda tunnettujen virheiden paljastamiseksi tai uusien toiminnallisuuksien testaamista varten jatkuvasti.

Testauksen työkalujen kanssa voi ilmestyä teknisiä ongelmia, kuten minkä tahansa muun työkalun kanssa. Siihen tulee varautua ja aloittaa työkalun asentaminen ja työkalun oppiminen ajoissa. Jotkut edistyneet työkalut ovat liian kalliita pienille yrityksille, joten automatisointi voi osoittautua olevan liian suuri investointi. Jotkut pitävät automatisointia liian monimutkaisena verrattuna siihen kuinka paljon testitapauksia jää automatisoinnin ulkopuolelle. [Lewis 205: 320–322.]

Hyvin valmistettu automatisointityökalun käyttöönotto hyödyntää koko projektia. Yllämainitut mahdolliset ongelmat ovat helposti ymmärrettäviä. Etukäteen tiedostamalla ne ja varautumalla niihin, ne voidaan välttää. Analysoimalla olemassa oleva testausprosessi ja resurssit voidaan päättää, minkä tyyppinen työkalu kannattaa valita. Automatisoinnin astetta voi kasvattaa eri projekteissa pohtimalla kokemuksia ja säiyttämällä onnistuneita käytäntöjä.

7 Automatisoidun ja manuaalisen tapojen vertailua

Testauksen automatisointi ei koskaan kata koko toiminnallisuus- ja regressiotestausprosessia. Joten on turha valita, testataanko ainoastaan työkaluilla vai manuaalisesti. Kyseessä on joko kokonaan manuaalinen tapa tai yhdistelmä molemmista. Testauksen tavoitteena on löytää virheet ja todistaa, että sovellus tekee, mitä asiakas tilasi. Manuaalinen testaus vie paljon aikaa, ennen kuin kaikki toiminnallisuusalueet ovat testattu [Cauldwell 2008: 96–97]. Tilanteessa, jolloin korjaustyöt tai kehitys edistyvät nopeammin kuin testaajat ehtivät testaamaan sovelluksen läpi, on iso riski, että virheitä jää huomaamatta ja näin projektin aikataulu ja tuoteen laatu vaarantuvat. Automatisoinnista on apua, kun toistuvat, helpot testitapaukset suoritetaan nopeasti ja toistetaan heti, kun on tarvetta. Testaajilla jää siten enemmän aikaa suorittaa monimutkaisia testejä, jotka vaativat ihmisen päätöksentekokykyä.

Mallipohjaisten työkalujen käyttöönotto kannattaa tehdä vasta silloin, kun on kokemusta testauksen automatisoinnista, skriptien kirjoittamisesta ja automatisoidun testausympäristön pystyttämisestä. Vaikka skriptien avulla voi automatisoida suurimman osan testeistä, eivät kaikki alueet ole testattavissa automaattisesti tai niiden automatisointi ei olisi järkevää. Esimerkiksi käyttöliittymän kymmeniä komponentteja sisältävän sivun asettelun tarkistaminen on lähes mahdotonta toteuttaa ohjelmallisesti. Lisäksi testaajan visuaalinen arviointi on paljon nopeampaa. Yleensä automaattisesti kannattaa suorittaa sellaiset tarkastukset kuin laskutoiminnot tai loogiset toiminnot, tietokantapäivitysten varmistukset, sivujen ohjaus ja lataaminen. Hewlett Packardin asiakkaiden antama keskiarvo on, että projektissa pystytään automatisoimaan noin 60 % testitapauksista, jolloin 40 % ajetaan manuaalisesti. [Hewlett-Packard 2007: 11.]

Hewlett-Packard, HP Quality Center -testaustyökalun valmistaja, suosittelee käyttämään manuaalista testausta seuraavissa projekteissa:

- Testattavaa sovellusta muutetaan nopealla tahdilla tai sovelluksen käyttöliittymä ja logiikka eivät pysy samana.
- Sovellus luodaan harvoin käytettävällä teknologialla, jolle ei ole automatisointi-työkaluja
- Sovelluksen tai käyttöliittymän komponentit ovat räätälöityjä itsenäisiä sovelluksia. [Hewlett-Packard 2007: 11.]

Automatisoinnin asteen valitseminen on testauksen johdon tehtävä. Kaikkien vaikuttavien osatekijöiden huomaaminen tuo tarkempaa tulosta ja testaus-suunnitelmasta tulee helpommin toteuttaa.

Automatisoitu testaus tuo usein korkeamman kattavuuden lyhyemmässä ajassa kuin manuaalinen testaus. Tässä tulisi huomioda se, että osa testitapauksista on ajettava manuaalisesti ja automaattisen ajon tulosten analysointi voi viedä aikaa testaajilla. Automatisoitu regressiotestijoukko helpottaa tuotteen testaamisen ennen hyväksymistestausta. Pitää muistaa, että samoilla testitapauksilla pystyy löytämään vain tiettyjä virheitä. Jos vanhalla testijoukolla ei löydy enää uusia virheitä, silloin tutkiva manuaalinen testaus, loppukäyttäjien osallistuminen testaukseen ja testijoukkojen laajentaminen auttavat löytämään testijoukon ulkopuolelle jääneitä virheitä ennen sovelluksen käyttöönottoa.

Henkilöstöresurssit vaikuttavat testausprosessin sisältöön ja onnistumiseen. Automatisoitu testaus säästää paljon resursseja vain silloin, kun testaajilla on vahva osaaminen automatisoinnissa ja sovelluksen testaus sopii automatisoinnin kohteeksi. Muuten testaajilla menee liian paljon aikaa automatisoinnin perusteiden, työkalun ja testausympäristön oppimiseen.

Manuaalinen testaus ei tunne uudelleenkäytettävyyttä. Testitapausten suoritus on toistettava joka kerta käyttäen samaa resurssien määrää. Automatisoitu testaus vaatii enemmän aikaa skriptien tai mallin valmistamisessa, mutta sen jälkeen testitapaukset voidaan käyttää uudelleen säästämällä resursseja.

Testaukseen liittyvä muutosten hallinta on hankalaa sekä manuaalisessa testauksessa että automaattisessa tavassa, kun sen toteuttamistavaksi valitaan nauhoittamis-työkaluja. Mallipohjaiset työkalut automatisoivat sellaisten testijoukkojen ylläpidon, joita on generoitu mallista. Mallin ylläpito on selkeämpi, koska siinä vältetään redundanssia – määrittelymuutokset päivitetään yhteen malliin eikä useisiin testi-tapauksiin.

8 Yhteenveto

Insinöörityön tavoitteena oli tutustua toiminnallisuus- ja regressiotestaukseen, niiden organisointiin ja selvittää, milloin kannattaa käyttää manuaalista testausta ja milloin automatisointi tuottaa säästöä ja tehostaa testausprosessia.

Työkokemuksen ja kirjallisuuden analyysin avulla voitiin muodostaa selkeitä päätelmiä testauksen automatisoinnin välttämättömyydestä. Vaikka automatisoinnin tarve testauksessa kuulostaisi itsestäänselvältä asialta, kirjallisuuden avulla saatiin myös selkeitä ohjeistuksia, milloin automatisointia ei kannata tehdä, ja jos sitä tehdään, mitä kannattaa automatisoida.

Manuaalisen testauksen osuus projektissa voi vaihdella riippuen sekä testattavan sovelluksen monimutkaisuudesta että testausprosessin kypsyydestä. Insinöörityössä kuvattiin tapoja, joiden avulla voidaan tehostaa manuaalista testausta. Yksi niistä on regressiokierrosten organisointi, jota käytettiin tutkitussa projektissa. Toinen tapa tehostaa manuaalista testausta sisältää valintatekniikoita, testitapausten uudelleenkelpoistamista, priorisointia ja uusien laajempien testitapausten suunnittelua.

Insinöörityössä käsiteltiin testauksen kypsyysmallin, jonka avulla testausprosessin organisointia voidaan arvioida ja parantaa. Testausprosessin kypsyyden ja valmiuden automatisoinnin käyttöönoton välillä löydettiin yhteyden, jonka mukaan automatisointi vaatii korkeimman testausprosessin kypsyyden.

Testaustyökaluja tarvitaan nykyään sekä automatisointitehtäviin että virheiden ja muutospyyntöjen elinkaaren seurannassa. Insinöörityössä kuvattiin joukko työkaluja, joista osaa käytettiin tutkitussa projektissa ja muihin tutustuttiin kirjallisuuden avulla.

Automatisointityökaluja käsiteltiin niiden kehitysjärjestyksessä. Viimeisen sukupolven testauksen automatisointityökaluja tarjoavat uuden mallipohjaisen testausprosessin, joka automatisoi testitapausten suunnittelun ja ylläpidon, mutta tuo testaajan tehtäviin mallin suunnittelu ja liittymien ohjelmointi.

Insinöörityössä saavutettiin asetettuja tavoitteita. Automatisoitu ja manuaalinen testauksen organisoinnin lähestymistavat käsiteltiin ja verrattiin. Automatisoinnille asetettiin esivaatimukset: testausprosessin kypsyys, automatisoinnin käyttöönoton vaiheistus ja potentiaalisiin ongelmiin varautuminen. Automatisoinnissa kokeneita testaajia voivat siirtyä mallipohjaisen testaukseen, jolloin testausprosessi muuttuu.

Insinöörityötä voidaan käyttää testausprosessin suunnittelussa ja organisoinnissa. Automatisoinnin asteen valitsemisessa auttavat sekä työkalujen arvioinnit että manuaalisen testauksen vahvojen ja heikkojen puolien analyysi.

Työssä ei otettu kantaa muihin kuin toiminnallisuus- ja regressiotestauksen vaiheisiin. Sen takia työtä ei kannata käyttää muiden testausvaiheisin suunnittelussa tai organisoinnissa.

Lähteet

Best practices for implementing automated functional testing solutions. 2007. Hewlett-Packard. White paper.

Binder R. 1999. Testing object-oriented systems. Addison-Wesley.

Burnstein, Ilene. 2003. Practical software testing. Springer.

Burnstein, Ilene, Suwanassart, Taratip, Carlson, Robert.1996. Developing a testing maturity model for software test process evaluation and improvement. International test conference. IEEE.

Cauldwell, Patrick. 2008. Code Leader: Using People, Tools, and Processes to Build Successful Software. Hoboken, NJ, USA: John Wiley & Sons, Inc.

Copeland, Lee. 2004. A Practitioner. Norwood, MA, USA: Artech House.

Dustin, Elfride. 2003. Effective software testing. Pearson education, Inc.

Farell-Vinay, Peter. 2008. Manage software testing. Auerbach publications.

Fujita, H. (Editor). 2008. New Trends in Software Methodologies, Tools and Techniques. Amsterdam, NLD: IOS Press.

Gao, J.Z., Tsao, Jacob H.-S., Wu, Ye. 2003. Testing and quality assurance for component-based software. Artech House.

Hass, Anne Mette Jonassen. 2008. Guide to Advanced Software Testing. Norwood, MA, USA: Artech House.

Holmes, Antawan, Kellog, Mark. 2006. Automating functional tests using Selenium. IEEE: AGILE 2006 Conference.

Holopainen, Juha. 2005. Regressiotestaus ja testien valintatekniikat. Pro-gradu tutkielma. Kuopion yliopiston tietojenkäsittelytieteen laitos.

Jacobs, Jef, van Moll, Jan, Stokes, Tom. 2000. The Process of Test Process Improvement. Xootic magazine, November 2000, s. 23–29.

Kan, Stephen H. 2002 Metrics and Models in Software Quality Engineering. 2nd edition: Addison Wesley.

Kosmatov, Nikolai. 2010. Constraint-Based Techniques for Software Testing. Teoksessa Meziane, Farid (editor), Vadera, Sunil (editor). Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects. IGI Global.
Lewis, E. William. 2005. Software testing and continuous quality improvement. 2nd edition: Auerbach publications.

Meyers, J. Glenford. 2004. The art of software testing. 2nd edition: John Wiley and sons.

Naik, Sagar, Tripathy, Piyu. 2008. Software Testing and Quality Assurance: Theory and Practice. USA: John Wiley & Sons.

Pyhäjärvi, Maaret, Pöyhönen, Erkki. 2004. Strategioista suunnitelmiin - selkeyttä käsitteiden sekamelskaan. Verkkodokumentti. 6.2.2011. http://testausosy.ttlry.fi/webfm_send/101.

Schuh, Peter. 2005. Integrating Agile Development in the Real World. Hingham, MA, USA: Charles River Media.

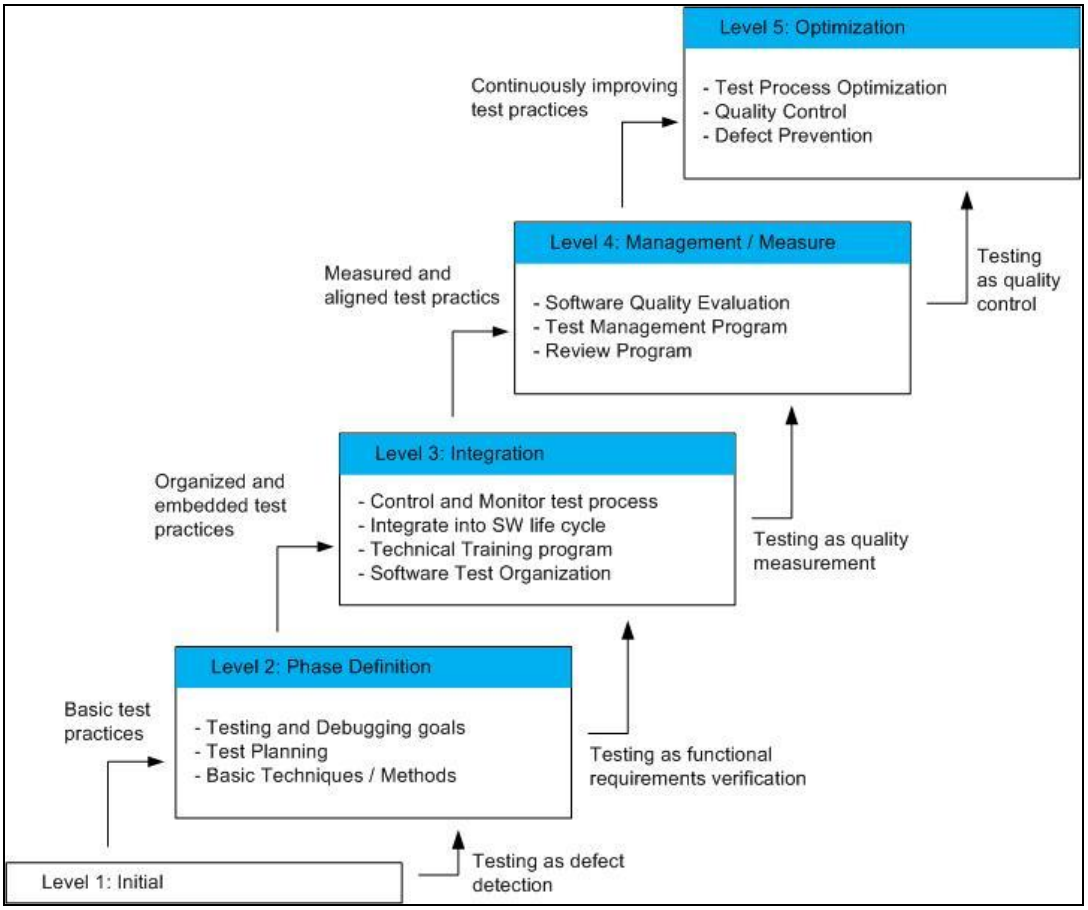
Thayer, Richard H. 1990. Software Engineering Project Management, A Top Down View. IEEE Tutorial. Software Engineering Project Management. IEEE Computer Society Press, Los Alamitos, CA.

Utting, Mark, Legeard, Bruno. 2007. Practical model-based testung. A tools approach. Morgan Kauffmann Publishers.

Watkins, John. 2001. Testing IT: An Off-the-Shelf Software Testing Handbook. Port Chester, NY, USA: Cambridge University Press.

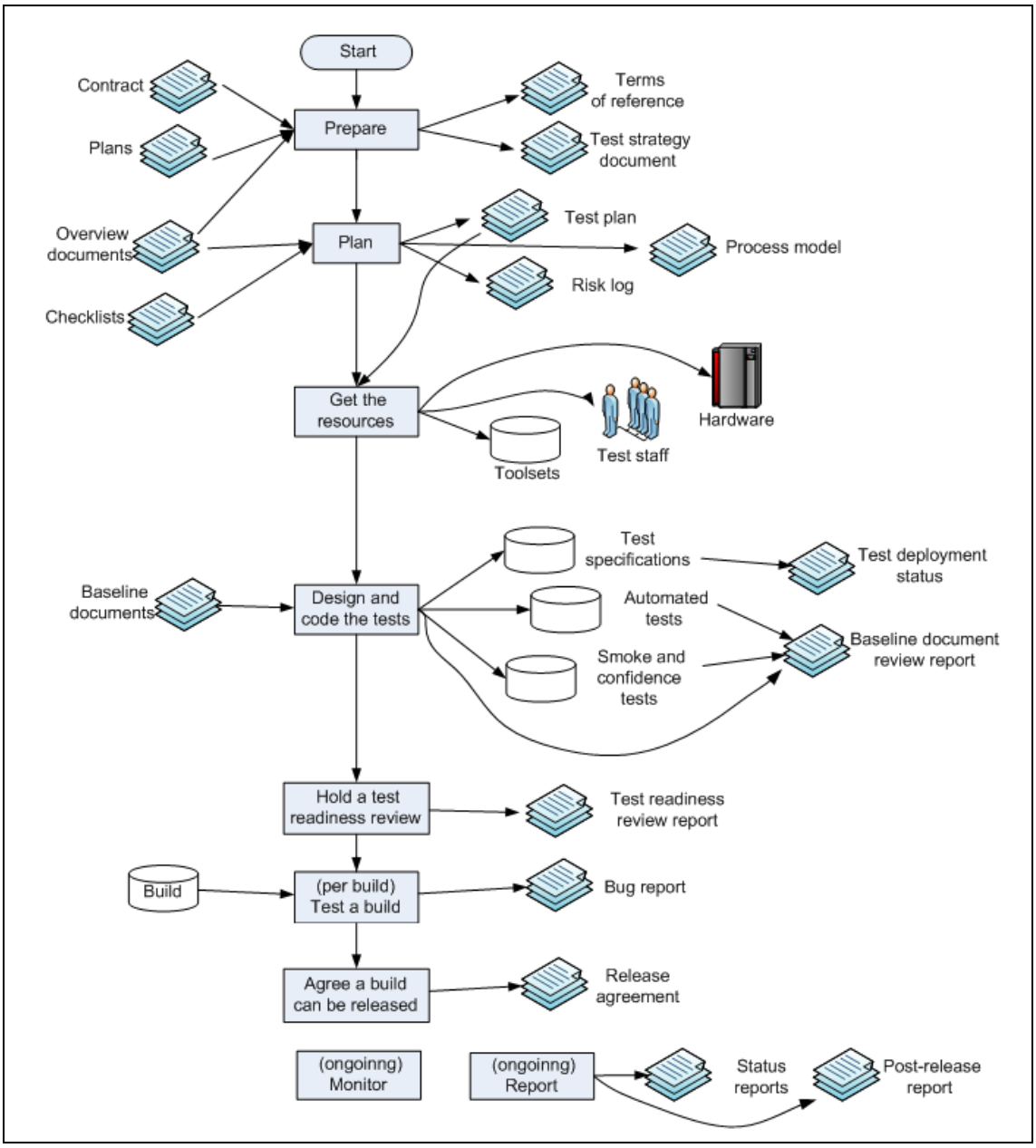
White, Brian A. 2000. Software configuration management strategies and Rational® ClearCase. A practical introduction. Addison Wesley.

Testauksen kypsyysmallin tasot



Kuva 1 Kypsyysmallin rakenne [Jacobs ym.2000: 24]

Toiminnallisuustestauksen prosessi



Kuva 1 Toiminnallisuustestauksen prosessi, [Farell-Vinay 2008, 20]