HAAGA-HELIA
ammattikorkeakoulu

Onnistunut testaus ja ketterät menetelmät

Virve Väyrynen

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2009

HAAGA-HELIA
ammattikorkeakoulu

Tiivistelmä

30.10.2009

Tietojenkäsittelyn koulutusohjelma

Tekijät Virve Väyrynen	**Ryhmä**
Opinnäytetyön nimi Onnistunut testaus ja ketterät menetelmät	**Sivu- ja liitesivumäärä** 54
Ohjaajat Jyri Partanen	

Perinteisesti testaus- ja tarkastusprosessit noudattavat V-mallia. Koska ketterät menetelmät tuovat muutoksia prosessiin, keskityin tarkastelemaan niiden ominaisuuksia. Komponentin testaus on hyvä esimerkki siitä, että testausprosessia kannattaa suunnitella etukäteen. Siksi kohdensin tutkimukseni nimenomaan komponenttien testaukseen.

Raportin tarkoitus on nostaa esiin asioita, joita pitää käsitellä ja ratkaista, kun suunnitellaan testausta ja testauksen automatisointia. Tutkin sitä, miten saadaan aikaan onnistunut testaus.

Tutkimusmetodina on laadullinen tutkimus. Hain tietoa analyysini pohjaksi kirjallisuudesta, seminaareista, testausseminaareista ja haastattelemalla asiantuntijoita. Jotta raportti ei olisi paisunut liian laajaksi, jätin tarkastelun ulkopuolelle tietoturva- ja suorituskykytestauksen. Onnistunutta testausta lähestyin muun muassa pureutumalla projektien epäonnistumisen syihin sekä poimimalla esimerkkejä onnistuneista projekteista.

Johtopäätökset

Ketteriä menetelmiä käyttävä projekti keskittyy tuottamaan liiketoiminnan arvonlisäystä (business value). Ketteriä menetelmiä käyttävä tiimi kehittää ohjelmaa ominaisuus kerrallaan. Näin ei tuhlata resursseja turhien toimintojen kehittämiseen ja muutoksiin voidaan vastata nopeasti. Tarkastelun perusteella kypsyystasoltaan korkealla tasolla oleva tiimi pystyy vastaamaan muutoksiin nopeasti, siksi tutkitaan prosessin kypsyysmallia (CMMI). CMMI mallin mukaan kypsyys on organisaation mittari, mutta myös tiimin voi hyödyntää mallia kypsyystasonsa kehittämisessä.

Ketteriä menetelmiä käyttämällä testaukseen voidaan saada nopeutta ja joustavuutta. Ketterät menetelmät korostavat ohjelman laatua ja automatisointia. Onnistuneen testauksen näkökulmasta tärkeää on toimiva kehitys- ja testausympäristö. Kehitysympäristön pitää sallia automatisoitujen testien kehittäminen, ajaminen ja myös käsin tehtävä testaus. Keskeistä on pystyä tekemään nopea muutos tuotannossa.

Ketterä ohjelmistokehitys voi käyttää testivetoista ohjelmistokehitysmallia. Automaatio on avain menestyksekkääseen ketterillä menetelmillä tehtävään ohjelmistokehitykseen. Kehitystyö on hyvä aloittaa pilottiprojektilla ja kehittää kestäviä testejä. Yksikkötason testien rinnalle pitää rakentaa myös toimiva aloitustestijoukko (Intake Test). On myös hyvä huomioida, että tarvitsemme erilaisia testejä eri testitasoilla.

Asiasanat
ketterä testaus, testaus, testaus ketterässä ohjelmistokehityksessä, testausautomaatio

HAAGA-HELIA
ammattikorkeakoulu

Abstract

30.10.2009

Degree Programme in Business Information Technology

Authors Väyrynen, Virve	**Group**
The title of thesis The designing of successful testing and agile software development	**Number of pages and appendices** 54
Supervisors Jyri Partanen	

A traditional testing and audit process follow the V-model. Because the agile software development is changing the process, this study is focusing on agile development. The testing of a component gives a good sample of a task, where we have to plan the testing process in advance. For this reason the target of the study is the testing of a component.

The purpose of this study is to promote facts that companies need to handle and resolve when they plan software testing and software test automation. The study will research how we achieve successful testing.

The research method of this study is a qualitative study. The information was sought in real situations like in seminars, test seminars and by interviewing experts. The study does not cover the security testing or the performance testing. The approach of this study is to examine the reasons why projects fail and pay attention successful projects to find a good sample to achieve successful testing.

Conclusions

Agile software project focus on returning business value. Agile software development team develops the software one feature at the time to make sure unnecessary features are not developed. A quick response to a chance is desired. A team with a high maturity level can respond to changes quickly. Therefore, the study has a paragraph about a process maturity model (CMMI). CMMI model measures the whole organization level, but it is also possible for a team to develop the maturity of the team.

With agile testing the development can have more velocity and flexibility. Agile development focus on high quality and high automation level. To accomplish a successful testing process it is important to have a working development and testing environment. Environment should allow the development and running of automated tests, as well as the manual testing. It is essential to be able to make changes in production quickly.

The agile software development can use the practice of Test-Driven-Development. Automation is a key to a successful agile development. It is a good practice to start with a pilot project and develop robust tests. On the side of unit level test harness, there should be also a working intake test set. It is also beneficial to develop testing and focus on the test levels; we need different kind of tests on different levels.

Key words
agile testing, testing, agile software development, test automation

Sisällys

Kuvat

Taulukot

Symboli- ja määritelmäluettelo

Dynaaminen testaus	Komponentin tai ohjelmiston evaluointia itse ohjelman tai sen osan suorituksen aikana ja perusteella kutsutaan dynaamiseksi testaukseksi.
Extreme Programming	(XP) ketterä menetelmä ohjelmiston kehitykseen.
JUnit	JUnit on yksinkertainen runko tai malli (framework) jolla tehdään toistettavia testejä.
Ketterät menetelmät	(Agile Methods) esim. Scrum, Lean, Extreme Programming (XP).
Lasilaatikkotestaus	White Box Testing. Testitapausten suunnittelussa ja toteuttamisessa käytetään hyväksi tietoa ohjelman toteutuksesta eli käytännössä testin suunnittelussa tulee olla käytössä ohjelman tai sen osan lähdekoodi ja toiminnallinen määrittely.
Mustan laatikon testaus	Black Box Testing. Mustalaatikkomenetelmällä ohjelmaa testataan ilman tietoa ohjelman sisäisestä rakenteesta, sen lähdekoodista yleensä tai algoritmista, jonka pohjalta ohjelma on toteutettu.
Open Source	Avoin lähdekoodi tarkoittaa mitä tahansa ohjelmistoa, jonka lisenssi täyttää Open Source Initiativen (http://www.opensource.org) määrittelemät vaatimukset.
OMG	Object Management Group on avoin konsortio, joka kehittää ja ylläpitää spesifikaatioita tietokoneteollisuudelle.
Scrum	Scrum on ketterä menetelmä ohjelmiston kehitykseen.

Staattinen testaus	Ohjelmiston testausta, ilman että ohjelmakoodia suoritetaan, kutsutaan staattiseksi testaukseksi.
Systeemityö-, prosessimalli	Malli kuvaa, miten ohjelmistotuotanto etenee ja mitä menetelmiä työssä käytetään sekä miten tuote dokumentoidaan (kuvausmalli) mm. Rational Unified Process (RUP), Rapid Application Development (RAD), erilaiset vaihejakomallit, Personal Software Process (PSP), Team Software Process (TSP), Agile Methods ja Extreme Programming (XP).
TDD	Test-Driven-Development, testauslähtöinen ohjelmointitapa on menetelmä, jonka pääperiaatteena on, että ohjelmoijat kirjoittavat matalan tason yksikkötestejä ennen ohjelmayksikköjen toteutusta.
Validointi	Validation, kelpoistaminen. Validoinnilla varmennetaan, että toteutettava järjestelmä vastaa loppukäyttäjän tarpeita.
Verifiointi	Verification, todentaminen. Verifioinnilla tarkoitetaan ohjelman oikeellisuuden ja oikean toiminnan todentamista.

1 Johdanto

1.1 Raportin tarkoitus ja tutkimuskysymykset

Kohdealue on komponentin testauksen suunnitteleminen, testaus ja testauksen kehittäminen ketterissä menetelmissä. Raportin tarkoitus on nostaa esiin asioita, joita pitää käsitellä ja ratkaista kun suunnitellaan testausta ja testauksen automatisointia.

Tutkitaan sitä, miten saadaan aikaan komponentin **onnistunut** testaus. Näkökulma tutkimukseen on tuoda testaus ohjelmistoprojektiin alusta alkaen ja tutkia miten voidaan toteuttaa komponenttien suunnittelu niin, että komponenttien testattavuus on hyvä (Design for Testability). Tutkitaan tilannetta kun tiimi käyttää ohjelmointityössä ketteriä menetelmiä kuten esim. Scrum, XP-ohjelmointi ja Test-Driven-Development (TDD) -menetelmiä.

Tutkimuksen keskeiset kysymykset:

1. Mitä uutta ketterät menetelmät tuovat testaus- ja kehitystyöhön?
2. Mitä on testattavuus (Design for Testability)?
3. Minkälaista uutta tutkimustietoa löytyy Test-Driven-Developmentista ym.?
4. Miten ketterissä menetelmissä otetaan huomioon eri testaustasot?
5. Miten testausta kannattaa kehittää tulevaisuutta varten?
6. Mitä testattavuuden kannalta tärkeitä ominaisuuksia komponenttitason testauksessa voi olla mukana?

1.2 Miksi aihe on tutkimisen arvoinen?

Testaus on muutostilassa, muutokset tapahtuvat todella nopeasti. Nopea muutos tarkoittaa sitä, että saatava tieto ei ehkä ole kovin hyvin tieteellisesti tutkittua (Itkonen 2006). Muun muassa Helian Tietotekniikkaseminaari -kurssille tehdyssä tutkimuksessa on tutkittu testauksen automatisointia ja todettiin automatisointiprojektien epäonnistuvan helposti.

Scrum-prosessissa joka iteraatiokierroksen aikana tiimi kehittää toimintatapojaan. Tavoitteena on löytää tiimin tehokkuutta lisääviä ja toimintaa parantavia toimintatapoja. Halutaan myös löytää uusia näkökulmia testauksen suunnitteluun ja automatisointiin. Testauksen

automatisointi on avainasia ketterillä menetelmillä toteutettujen projektien onnistumisessa. Tarkastelen asiaa testaajan näkökulmasta. Tutkimus pyrkii tuomaan lisätietoa aiheesta.

1.3 Käytettävät tutkimusmenetelmät

Käytettävät tutkimusmenetelmät ovat kvalitatiivisia, laadullisia. Tietoa hankitaan kirjallisuudesta sekä testauksesta tehdyistä tutkimuksista, seminaareista ja kursseilta. Ketterien ohjelmistokehitysmenetelmien käyttöä ja tunnettavuutta edistävä yhteisö (Agile Finland ry) parantaa ketterien menetelmien tunnettavuutta ja järjestää seminaareja, koulutusta, konferensseja. Sytykkeen Testauksen osaamisyhteisö (TestausOSY - FAST) järjestää testausaiheisia kursseja joilta saa monipuolista tietoa testauksesta. Kursseilla tutustuu testauksen asiantuntijoihin ja saa tuoretta tietoa ja uusia vinkkejä, joita voi käyttää testaustyössä. Tietoa hankitaan myös haastattelemalla asiantuntijoita ja testausaiheisesta kirjallisuudesta.

Tietoa haettiin myös luonnollisissa todellisissa tilanteissa. Osa tutkimuksen tuloksista perustuu osallistuvaan havainnointiin eli testaajana itse tehtyihin havaintoihin ja kehittämisideoihin.

1.4 Ympäristö

Tutkimuksen ympäristö on kuvitteellinen.

1.5 Raportin lähtökohdat

Olen työskennellyt ketteriä menetelmiä käyttävissä projekteissa, joissa testaaja työskentelee Scrum-tiimin jäsenenä. Scum-prosessi ei varsinaisesti tunne testaajan roolia. Olen huomannut, että kehittäjien, jotka tuntevat ohjelman sisäisen rakenteen (White Box), näkökulma testaukseen on erilainen kuin testaajan näkökulma (Black Box). Tässä tutkimuksessa haluan tutkia komponentin suunnittelua, testauksen suunnittelua, komponentin rakentamista ja testausta testaajan näkökulmasta. Tutkin mitä uutta Test-Driven-Development tuo testaukseen.

1.6 Rajaus

Projekti on selvitys- ja ratkaisuhanketyyppinen opinnäytetyö. Siinä ei tutkita ohjelmistoprosessia. Tutkimuksessa ei keskitytä tietoturvatestaukseen eikä suorituskykytestaukseen.

2 Teoriatausta

2.1 Testauksen määritelmiä

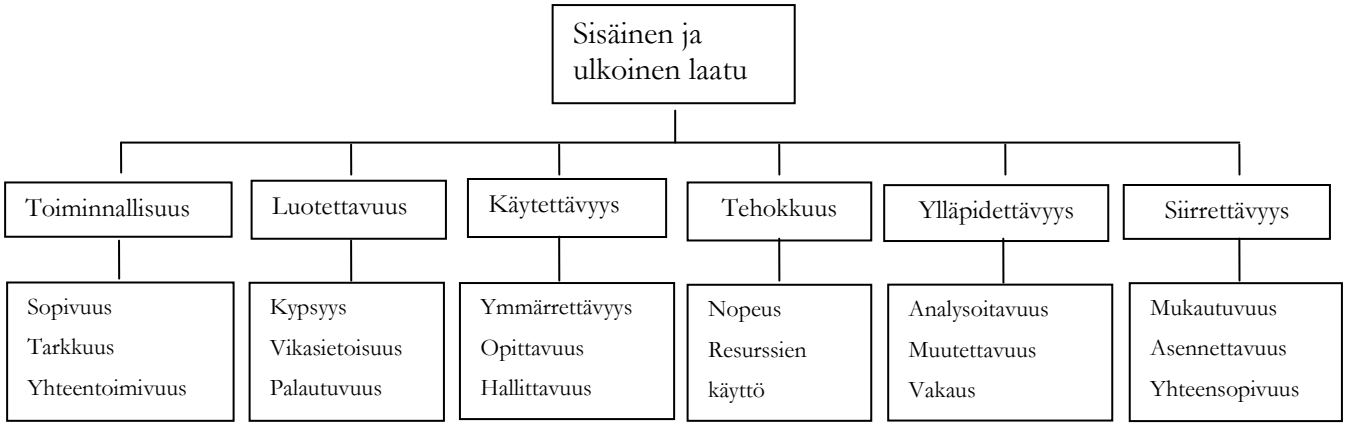
Glenford Myers määrittelee testauksen näin: *"Testing is the process of executing a program with the intent of finding errors."* Toisin sanoen testaaja yrittää kaikin tavoin rikkoa ohjelman ja löytää virheet. Myers ei siis määrittele testausta, kuten ohjelmoijat tai projektipäälliköt monasti määrittelevät: *"Testing is the process of establishing confidence that a program does what it supposed to do."* (Myers 2004.)

Pol ja Veenendaalin määritelmä on seuraava: *"Testing is a process of planning preparation and measuring aimed at establishing the characteristic of an information system and demonstrating the difference between the actual and required status."*

Heidän mukaansa testaus on siis hallittua, teknistä kurinalaista toimintaa (a professional dicipline). Testauksessa mitataan ohjelmiston laatu. Testaus tuottaa tietoa ohjelmiston suunnittelu- ja toteutusvaiheen tilasta ja osoittaa laadun parantamiselle hyviä kohteita (riskit, suorituskyky, käytettävyys). Testaus on vaikeaa. Kehitettävät ohjelmat ovat koko ajan monimutkaisempia ja sisältävät yllättäviä virheitä. Testauksen osuus ohjelmistoprojektien kustannuksista on merkittävä. (Pöyhönen, 2003.)

2.2 Testaus ja laatu

ISO 9126-standardi luettelee keskeiset laatuattribuutit sekä näiden osa-attribuutit. (Fenton, Pfleeger, 1997)



Kuva 1. Keskeiset laatuattribuutit ja osa-attribuutit. (Fenton, Pfleeger, 1997)

Laadun määritelmiä on paljon, eräs määritelmä on *"Tuotteen tai palvelun kaikki piirteet ja ominaisuudet, joilla tuote tai palvelu täyttää sille asetetut tai oletettavat vaatimukset"* (ISO).

Alan asiantuntijoiden mukaan laatu tarkoittaa myös:

- sopivuutta käyttöön ja tarkoitukseen (Joseph Juran)
- asiakkaan nykyisten ja tulevien tarpeiden täyttämistä laadun avulla (Edvards Deming)
- vastaavuutta vaatimuksiin (Philip Crosby)
- toiminnan laadun avulla jatkuvasti kilpailukykyisenä pysymistä muuttuvissa olosuhteissa (Timo Silén) (Silén 2001)

Ohjelmiston hinta- ja laatutekijät määritellään ohjelmistoprojektin määrittelyvaiheessa. Ohjelmiston hinta määrää mitkä ominaisuudet projektin johto ja -asiakas nostaa esiin. (Haikala, Märijärvi 2001: 177, 183)

Testattavuus (Testability) on yksi laatutekijä.

Yleinen virheellinen oletus on, että testaus on laadunvarmistusta (QA) (Kaner 2004: 6). Testaus on vain yksi laadunvarmistuksen osa. Testauksella ei voi muuta kuin todentaa, että

virheitä löytyy. Testaus ei pysty takaamaan mitenkään, että virheitä ei löydy enää lisää. QA on laajempi prosessi, jonka tarkoitus on laadun rakentaminen. (Itkonen 2005.)

2.2.1 Miten laatua voidaan rakentaa ja mitata?

Laadun käsite on muuttunut alkuperäisestä tuotteen virheettömyydestä kokonaisvaltaiseksi liikkeenjohdon käsitteeksi. Nykyisin laatu käsitetään yhä useammin yrityksen laaja-alaiseksi kehittämiseksi ja johtamiseksi, jonka tavoitteena on asiakkaiden tyytyväisyys, kannattava liiketoiminta ja pitkällä tähtäimellä myös kilpailukyvyn säilyttäminen ja kasvattaminen. (Silén 2001.) Yrityksen toimintatavat ovat yrityksen laatujärjestelmä. (Turun yliopisto 2007.)

Laatu rakennetaan prosessilla johon on rakennettu laatua parantava kierre, joka johtaa koko ajan paraneviin tuloksiin. Onnistuneessa laadunrakennusprosessissa on määritelty selkeät mitattavissa olevat tavoitteet. Ne on jaoteltu välitavoitteiksi. Prosessin kulun seuranta on järjestetty ja siitä annetaan palautetta. Prosessin päätteeksi tulokset mitataan ja prosessiin osallistuneet ihmiset palkitaan. Prosessin jälkeen tehdään aina arviointi, hyvä ja huonot puolet kerätään yhteen ja niistä opitaan, tarkoituksena se, että virheitä ei enää toisteta. (Silén 2001.)

Laadun rakentamiseen löytyy klobaaleja malleja (best practices), jotka rakentavat laatua, pienentävät riskejä ja parantavat kustannustehokkuutta. Ohjelmistotuotannon erityispiirteiden takia ohjelmistotuotteen laadukkuuden arvioinnissa painotetaan hyviä toimintatapoja. Ohjelmiston kehittämisessä käytettäviä toimintaprosesseja kehitetään ja niitä noudattamalla syntyy tietynlaatuinen lopputulos. Tälläisiä on esim.:

- hyvä hallintotapa (cobit) – tilaaja tuottaja – malli (hyvän johtamisen malli)
- palvelujohtaminen (itil)
- sarjan ISO9001 standardi
- eurooppalaiseen laatukriteeristöön perustuva EFQM-malli
- Team Software Process (TSP)
- Software Process Improvement (SPI)
- Test Process Improvement (TPI)
- TMap
- benchmarking-tekniikka

Muita ohjelmistotuotannossa käytettäviä laatujärjestelmiä ovat kyvykkyys- ja kypsyysmallit, joista tunnetuimpia ovat SPICE ja CMMI. (Turun yliopisto 2007.)

CMMI on apuväline organisaation prosessin parannukselle. CMMI määrittelee tavoitteet ja prioriteetit parantamiselle, määrittelee ja ohjeistaa laadukkaiden prosessien ominaisuudet ja sisällön ja toimii mittarina nykyisen prosessin ja toimintatapojen arvioinnissa. CMMI ei ole prosessi, vaan kokoelma hyvän prosessin ominaisuuksia ja elementtejä. (Turun yliopisto 2007.)

CMMI mittaa yrityksen prosessien kehitystä kyvykkyystasojen (capability level) ja kypsyystasojen (maturity level) avulla. Kyvykkyydellä (capability) viitataan yksittäisen prosessin ominaisuuteen. Kyvykkyystasoilla edetään taso kerrallaan, seuraavaa tasoa aletaan kehittää vasta kun edellinen taso on saavutettu. Kypsyys (maturity) ilmentää organisaation saavuttamaa tilaa eli vakiintumista. Jatkuva soveltamistapa (continuous representation) mahdollistaa valittujen prosessien kehittämisen eri tahdissa ja eri tasoilla. (Salo 2007.)

Level	Focus	Process Areas
5 Optimizing	Continuous Process Improvement	Organizational Innovation and Deployment Causal Analysis and Resolution
4 Quantitatively Managed	Quantitative Management	Organizational Process Performance Quantitative Project Management
3 Defined	Process Standardization	Requirements Development Technical Solution Product Integration Verification Validation Organizational Process Focus Organizational Process Definition Organizational Training Integrated Project Management Risk Management Decision Analysis and Resolution
2 Managed	Basic Project Management	Requirements Management Project Planning Project Monitoring and Control Supplier Agreement Management Measurement and Analysis Process and Product Quality Assurance Configuration Management
1 Initial		

Kuva 2. CMMI:n kypsyysmalli, perusversio (Nevalainen 2005)

Kyvykäs prosessi on ennustettava ja varmatoiminen eli sen hajonta on saatu hallintaan, ja sillä on ennakoiva kyky saavuttaa tiukkojakin tavoitteita. Kyvytön prosessi käyttäytyy epävakaasti ja siinä on kaikenlaisia riskejä. Kypsyys ilmentää organisaation vakiintumista eli ”kestävyyttä”. Mitä parempi kypsyystaso on, sitä vähemmän epävarmuutta ja riskejä. Useimmiten korkeampi

kypsyys ilmentää myös parempaa tehokkuutta, onhan turha sählinki jäänyt pois eikä epäonnistumisista johtuvia laatukustannuksia juurikaan tule. (Nevalainen 2005.)

Mallissa käytetty prosessilyhenne	Prosessin nimiehdotus	Prosessin oleellinen sisältö
PP (Project Planning)	Projektin suunnittelu	Projektin laajuuden määrittely, estimointi, suunnittelu, sitoumusten hankinta projektin eri osapuolilta
PMC (Project Monitoring and Control)	Projektin valvonta ja ohjaus	Suunnitelman toteutumisen seuranta, projektin ohjauksessa tarvittavien toimenpiteiden tunnistaminen ja toteutus, sidosryhmien hoito projektin aikana
RSKM (Risk Management)	Riskienhallinta	Organisaation ja projektin tasolla tapahtuva näkyvä ja jatkuva riskien tunnistaminen, toimenpidesuunnittelu ja riskien rajoittaminen
IPM (Integrated Project Management)	(Yhtenäinen) projektijohtaminen	Projektin toteuttaminen organisaation määrittelemien standardien ja menettelyjen mukaisesti, sekä palautetiedon keruu niiden soveltuvuudesta ja parantamistarpeista
QPM (Quantitative Project Management)	Mittauspohjainen projektijohtaminen	Organisaation määrittelemien mittareiden ja niiden tavoitearvojen saavuttamiseen pyrkivä käytännön projekti

Kuva 3. CMMI-malliin sisältyvät projektijohtamisen prosessit (Nevalainen 2005)

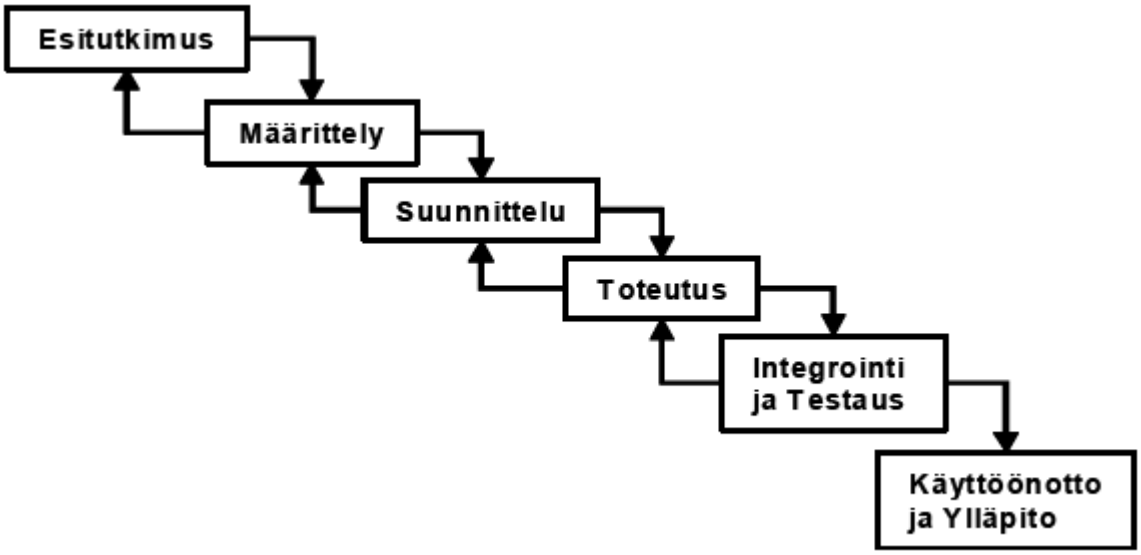
CMMI-mallissa käytetty kypsyyden määritelmä on: *the extent to which an organization has explicitly and consistently deployed processes that are documented, manage, measured, controlled and continually improved.* Määritelmä sisältää siis kyvykkyysasteikon ja lisäksi vaatimuksen, että ne ovat kunnolla käytössä organisaation laajuisesti. Kypsyys on siis ennen kaikkea organisaation mittari. (Nevalainen 2005).

Riskienhallinta on osa laadunhallintaa. Tavoitteena on

– riskien tunnistaminen
– riskien minimointi riskistrategian avulla
– proaktiivinen potentiaalisten riskitekijöiden seuranta (Deloitte 2005).

2.3 Testaus perinteisissä projekteissa

Ohjelmiston kehitystyötä kuvattaessa käytetään erilaia vaihejakomalleja. Malli kuvaa vain osaa todellisuudesta, ohjelmistokehitysprojektissa voidaankin katsoa montaa eri mallia samaan aikaan. Perinteisin ja käytetyin vaihejakomalli on vesiputousmalli. (Haikala & Märijärvi 2003: 36, 52.)

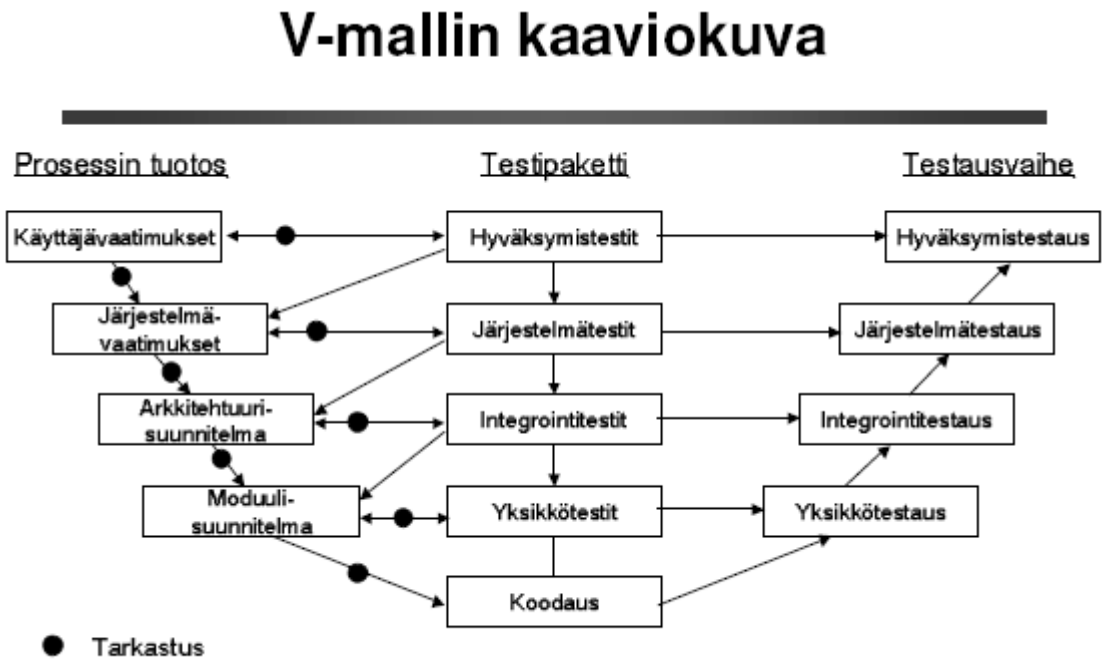


Kuva 4. Vesiputousmalli (Haikala & Märijärvi 2003)

Malleissa ohjelmiston kehitystyö ja elinkaari on jaettu vaiheisiin. Perinteinen vesiputousmalli sopii projekteihin, joissa toteutetaan järeitä järjestelmiä, järjestelmän käyttöikä on pitkä ja käyttäjien vaatimukset ovat tiedossa. (Huhtamäki 2007.)

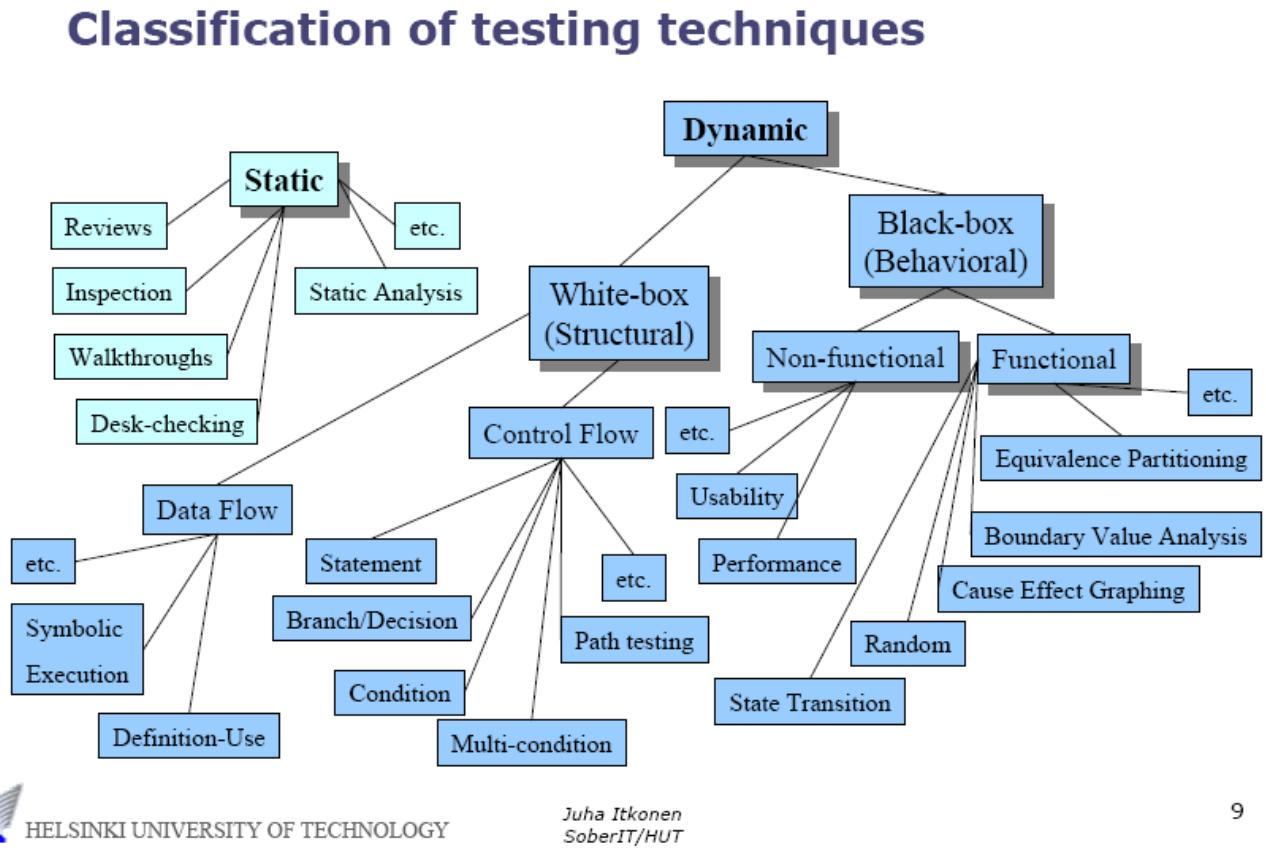
Testauksen klassinen vaihejako on V-malli. Testaus kuuluu tiiviisti ohjelmistokehityksen jokaiseen vaiheeseen. Testaustasot ovat yksikkötestaus, integrointitestaus, järjestelmätestaus ja hyväksymistestaus. (Taina 2004).

Testausta varten on oltava käsitys ohjelman syöte- ja tulosavaruudesta ja tiedettävä oikea lopputulos. On oltava spesifikaatio, jonka perusteella testitapaukset voidaan suunnitella. Virhe (error, mistake, bug) on poikkeama spesifikaatiosta. (Tieturi 2003.)



Kuva 5. V-malli (Taina 2004)

2.3.1 Testaustekniikat



Kuva 6. Testaustekniikoiden luokittelu (Itkonen 2005)

2.3.2 Staattinen testaus (Static Testing)

Testit joita suoritetaan ohjelman ollessa staattisessa tilassa, siis ohjelmaa suorittamatta:

- Tarkastukset
- Analysoinnit
- Koodikattavuuden mittaaminen
- Dokumentaation testaus (documentation testing)
- Standardien testaus (standards testing)

Staattisen testauksen avulla voidaan löytää jopa 60 % ohjelmiston virheistä. Staattisen analyysin työkaluilla voidaan mitata koodin kompleksisuus ja tietoa käytetään hyväksi, kun testataan nämä riskipohjaista testausta vaativat osat. Riskipohjaisessa testauksessa osat varmistetaan toimiviksi. (Boehm, Basili 2001.)

2.3.3 Dynaaminen testaus (Dynamic Testing)

Testit joita suoritetaan käyttämällä ohjelmaa:

Lasilaatikkotestaus (White Box Testing)
Testaus suoritetaan koodaajan näkökulmasta, eli tunnetaan toteutus. Painopiste on teknisen toteutuksen testaamisessa.

Mustan laatikon testaus (Black Box Testing)
Testaus suoritetaan käyttäjän näkökulmasta, eli ei kiinnitetä huomiota toteutukseen. Painopiste on toiminnallisuudessa joten testaus perustuu määrittelyihin.

2.3.4 Testitapaukset

Käyttötapauksista (Use Case) johdetaan testitapaukset. Testitapaus (Test Case) on ohjelmistolta vaaditun ominaisuuden tai toiminnon testi, jolle on määritelty lähtötiedot, toimenpiteet ja odotettu tulos. (Tieturi 2003.)

Testitapaukset erotetaan **positiivisiin** ja **negatiivisiin testitapauksiin**. Positiiviset testitapaukset osoittavat, että järjestelmä toimii oikein, joten testitapausten pitäisi onnistua

(pass). Negatiivinen testitapaus testaa toimintoja, joissa järjestelmän käyttäjä tai toinen liittyvä järjestelmä antaa ohjelmistolle vääränlaisen syötteen, jonka pitäisi epäonnistua (fail). Syöte voi olla esimerkiksi omituinen merkki (’~^&”) palveluikkunan otsikossa tai liitetiedoston nimessä. Negatiivinen testaus (negative testing) on järjestelmän toiminnan kannalta erittäin tärkeää, ja sen pois jättäminen järjestelmän rakennusvaiheessa aiheuttaa vaikeasti korjattavia virheitä. (Lyndsay 2003.)

Negatiivisesta testauksesta köytetään myös termiä **fuzzing**. Fuzzing on testausmenetelmä jossa ohjelmistolle annetaan väärämuotoista dataa syötteenä, tavoitteena saattaa ongelma vikatilaan. (Tikkanen, 2009.)

Testitapausten tekemisessä käytetään perus testaustekniikoita:

- EP = Equivalence Partitioning
- BVA = Boundary Value Analysis
- CE= Cause/effect
- EG = Error guessing
- ECP = Equivalence Class Partitioning

Virheet pitää löytää mahdollisimman aikaisin, jotta testaus on taloudellisempaa. Testausprosessissa tarkkaillaan jatkuvasti hintaa, laatua ja sitä koska testaus lopetetaan. (Virkanen 2003.)

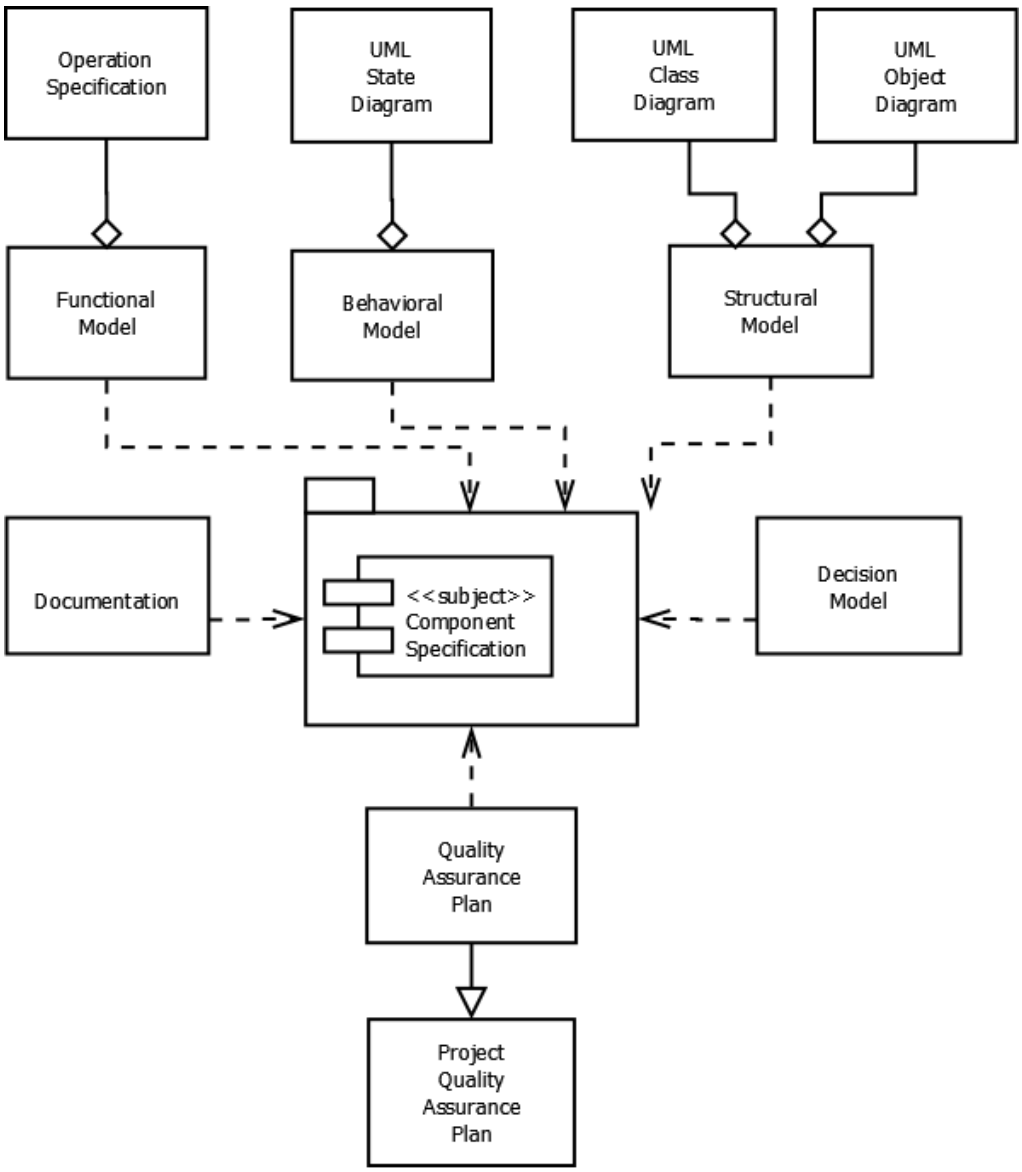
2.4 Komponentti

Objektiorientoituneiden metodien, ohjelmistoarkkitehtuurien, design patterns -metodin, Software Process Improvement (SPI) metodin, UML kielen (Unified Modeling Language) ja mallipohjaisen ohjelmistokehityksen kehittyminen johtivat siihen, että objektiorientoituneet ohjelmat yleistyivät. Vesiputousmallista haluttiin siirtyä käyttämään evoluutioon (spiraali) perustuvia vaihejakomalleja ohjelmistojen kehityksessä. COTS (Commersial off-the-shelf) eli kolmannen osapuolen valmiiksi kehittämät komponentit ja Open Source ohjelmistot yleistyivät. Ohjelmistojen kehittyminen monimutkaisemmiksi ja ohjelmointikäytäntöjen kehittyminen johtivat siihen että haluttiin rakentaa uudelleenkäytettäviä, valmiiksi testattuja komponentteja. (Taipale 2007, 24.)

2000 luvulla uusi ohjelmistokehityksen trendi on komponettipohjainen ohjelmistosuunnittelu (Component-Based Software Development CBD) ja komponenttien rakentaminen niin, että uudelleenkäytettävyys kehityksen on keskeinen tavoite. Päämääränä on lyhentää kehitysaikaa, pienentää ohjelmistokehityksen kustannuksia, rakentaa tuotteita joiden laatu on korkeampi ja testaus on suoritettu kunnolla. Asian ydin on uudelleen tehtävän työn välttäminen (Boehm 2006). Komponentit pitävät sisällään esimerkiksi metodeja, luokkia, objekteja, funktioita, moduuleita, suoritettavia ohjelman osia, tehtäviä (tasks), aliohjelmia ja sovellusten aliohjelmia. (Taipale 2007, 24.)

2.4.1 Komponentin määritelmä

"A component is a reusable unit of composition with explicitly specified provided and required interfaces and quality attributes that denotes a single abstraction and can be composed without modification." Tämä on Hans-Gerhard Grossin eräs määritelmä komponentista.



Kuva 7. Komponentin spesifikaatiota kuvaavat sivutuotteet (Gross 2005)

Vapaasti käännettynä; *"Komponentti on uudelleenkäytettävä rakenne jolla on yksikäsitteiset selvästi määritellyt, toimitetut ja tarvittavat laatuominaisuudet ja rajapinnat, ja joka voidaan koota ilman muunnosta."* Komponentista on myös monia vastaavanlaisia määritelmiä. (Gross 2005, 2.)

Komponentin määrittely sisältää kaikki dokumentit joita komponentin suunnittelu, rakentaminen, integroiminen, testaaminen ja käyttöönotto sisältävät. (Gross 2005, 2.)

2.4.2 Software Product Family

Komponenttien kehittyminen edisti myös tuoteperheiden (Software Product Family) kehittämisen. Tuoteperhe on joukko tuotteita, jotka jakavat samanlaisia ominaisuuksia tai toiminnallisuuksia. Kun koko tuoteperhe rakennetaan kokonaisuutena, tuotteiden yhteiset

piirteet ovat hallittavissa ja hyödynnettävissä. Yhteisiä osia voidaan hallita ja käyttää uudelleen ja tarkoituksena saada mittavia kustannussäästöjä. (Tevanlinna, Taina & Kauppinen 2004.)

2.4.3 Arkkitehtuuri

Ohjelmiston arkkitehtuuri muodostaa rungon kun rakennetaan onnistuneesti tehokkaita ohjelmistojärjestelmiä. Ohjelmiston arkkitehtuuri sallii tai estää ohjelmiston laatuattribuuttien, kuten suorituskyky tai toimintavarmuus, kehittämisen. Arkkitehtuurien kehittäminen sisältää arkkitehtuurien suunnittelun, arvioinnin, elinkaaren integroinnin ja jälleenrakennuksen. (SEI 2009.)

On mahdotonta rakentaa yhtenäinen komponenteista koottu järjestelmä ilman johdonmukaista arkkitehtuuria. Arkkitehtuuri voi koostua useista ohjelmistostruktuureista ja on koottu elementeistä, elementtien näkyvistä ominaisuuksista ja niiden välisistä suhteista. Yleisiä arkkitehtuurirakenteita ovat vaikkapa hajautettu, kerroksinen, prosessoitu ja clientserver. (Toroi 2009, 21.)

2.4.4 Komponenttien testaamisen haasteita

Komponenttipohjaiset ohjelmistot ja ketterä ohjelmistokehitys näyttää olevan ohjelmistokehityksen nouseva kehityssuunta. Testauksessa tämä kehitys tarkoittaa komponenttipohjaisten järjestelmien testausta ja ketterillä menetelmillä tuotettavien ohjelmien testausta sen lisäksi, että testataan suunnittelulähtöisten (plan-driven) vaihejakomallien mukaan. Ketterillä menetelmillä toteutettu ohjelmistokehitys vaatii esimerkiksi testitapausten ohjelmointia ennen varsinaisen ohjelman ohjelmointia ja korostaa hiljaisen tiedon (tacit knowledge) merkitystä. Komponenttipohjainen ohjelmistokehitys ja ketterät menetelmät korostavat testausprosessin ja tiedonhallinnan kehittämistä koska molemmat vaikuttavat testausprosessin sisältöön ja tiedonhallintaan. (Taipale 2007; 33.)

Komponenttien testaus on haastavaa. Boehm (2006) on huomioinut, että komponentit ovat läpinäkymättömiä ja niitä on vaikeaa debugata. Komponentit ovat usein rakenteeltaan erilaisia, koska komponentteja pitää kehittää kilpailukykyisesti. Komponenttien kehitystä on vaikea kontrolloida, komponentin uusi versio julkaistaan keskimäärin 10 kuukautta edellisen julkaisun jälkeen ja yleensä valmistajan komponentin tuki lakkaa noin kolmen päivitys tai kehityskierroksen jälkeen. (Taipale 2007, 24.)

Taulukko 1. Binderin mukaan nämä uudet vaarat ovat läsnä kun käytetään objektiorientuneita kieliä (Taipale 2007, 25.)

1. Dynaamiset sidokset ja monimutkaiset perimisrakenteet saavat aikaan monia virhemahdollisuuksia koska sidokset voivat olla epätoivottuja tai oikea käyttö onkin ymmärretty väärin.
2. Rajapintojen ohjelmoinnissa tapahtuu virheitä kun käytetään proseduraalisia ohjelmointikieliä. Koska objektiorientoituneissa kielissä käytetään monia pieniä komponentteja, ohjelmissa on paljon enemmän rajapintoja ja siksi virheet rajapinnoissa lisääntyvät.
3. Objektit käyttävät eri tiloja (state) mutta ohjelman tilojen kontrolli on usein rakennettu ohjelman sisään niin, että hyväksyttävä tapahtumien sekvenssi on vaikea havaita. Tilojen hallintaan liittyvät virheet lisääntyvät.

Toroi (2009) on tutkinut komponenttipohjaisten järjestelmien testausta. Tutkimuksessa käytetty järjestelmä on tehty vesiputousmallin mukaan. Tutkimuksen johtopäätöksenä huomioitiin:

1. Integraattorin ja asiakkaan roolit pitää ottaa paremmin mukaan testausprosessiin. Näissä molemmissa rooleissa pitää käyttää erilaisia testausmetodeja kuin mitä kehittäjät käyttävät.
2. Tutkimuksen mukaan ohjelmistoyhtiöt eivät panosta ohjelmistojen yhdenmukaisuuteen vapaaehtoisesti. Asiakkaiden tulisi vaatia, että ohjelmistot tehdään standardien mukaan. Viranomaiset ovat heränneet ja heidän puoleltaan tulee uusia vaatimuksia, tämä onkin ainoa asia mikä parantaa ohjelmistojen laatua.
3. Määritysten mukaisuuden ja yhteentoimivuuden testaus (Interoperability conformance testing) tarvitsee hyvät laatuspesifikaatiot. Ilman asianmukaisia spesifikaatioita yhteentoimivuuden testausta ei voi tehdä niin, että yhdenmukaisuus voidaan varmistaa. Tällä hetkellä terveydenhuollon toimialueella ei ole asianmukaisia spesifikaatioita tämäntyyppiseen testaukseen.
4. Testausprosessin jatkuva parantaminen on tärkeää. Kun testausprosessia parannetaan, organisaation on kiinnitettävä huomiota testausprosesseihin ja myös testitapausten dokumentointiin, regressiotestaukseen ja siihen millä testaustasolla organisaatio on. (Toroi 2009, 52.)

Edellinen tutkimus perustuu kokemuksiin, jotka on saatu kun Suomen terveydenhoitojärjestelmän käyttämät ohjelmistot on tilattu eri toimittajilta komponentteina.

2.4.5 Mallipohjainen testaus (Model based testing)

Mallipohjaisen testauksen ideana on, että testitapausten sijaan ohjelmoija tai testaaja laatii testimallin, josta voidaan käsin tai automaattisesti generoida ääretön määrä erilaisia testitapauksia.

Model Driven Testing tarkoittaa sitä, että piirretään malli, josta automaattisesti generoidaan testitapauksia. Testitapausten generointiin käytetään tarkoitukseen kehitettyä työkalua. **Model Based Testing** tarkoittaa sitä, että testit luodaan ja automatisoidaan mallin avulla. Kun testauksessa käytetään black box -menetelmää, malli voi olla erilainen kuin toteutus, mutta silti mallin avulla voidaan luoda tarpeellisia ja yllättäviä testiskenaarioita. (Hartman 2006.)

Testimalli luodaan käyttäen OMG:n Model-Driven Architecture:sta tuttuja menetelmiä, kuten mallintamalla UML-kielellä tai mallintamalla suunnittelumallien (Design Patterns) avulla. (Heckel, Lohmann 2003). Suunnittelumallit ovat malleja, jotka kuvaavat suunnitelmatason ratkaisun johonkin toistuvasti esiintyvään ohjelman tai ohjelmiston toteutusongelmaan.

Mallipohjaisen lähestymistavan etuna on se, että testimalleista generoidut testitapaukset sisältävät sellaisia tapahtumasekvenssejä, joita kukaan ei ole ennen edes tullut ajatelleeksikaan. Kunhan vain mallin tekemiseen panostetaan kunnolla, sen avulla voidaan löytää uusiakin virheitä. Vahvoin alue mallipohjaiselle testaukselle on rinnakkaisuuden ilmiöiden hallinta, joka ihmisille on vaikeaa erilaisten vaihtoehtoisten suoritusten valtavan määrän takia. (Robinson Harry 2005.)

2.5 Ketterät menetelmät

Ketterän kehityksen perusmääritelmä Agile Manifesto julkaistiin vuonna 2001. Manifeston sisältö on seuraava:

> *Me etsimme parempia keinoja ohjelmistojen kehittämiseen tekemällä sitä itse ja auttamalla siinä muita. Tässä työssämme olemme päätyneet arvostamaan*
> **Yksilöitä ja vuorovaikutusta** *enemmän kuin prosesseja ja työkaluja*

Toimivaa sovellusta *enemmän kuin kokonaisvaltaista dokumentaatiota*
Asiakasyhteistyötä *enemmän kuin sopimusneuvotteluita*
Muutokseen reagoimista *enemmän kuin suunnitelman noudattamista.*
Vaikka oikeallakin puolella on arvoa, me arvostamme vasemmalla olevia asioita enemmän.
(Beck ym. 2001.)

Perusarvojen lisäksi manifestissa (Beck 2001) kuvataan 12 periaatetta, joita ketterä ohjelmistokehitys noudattaa. Suomennettuna ne kuuluvat seuraavasti:

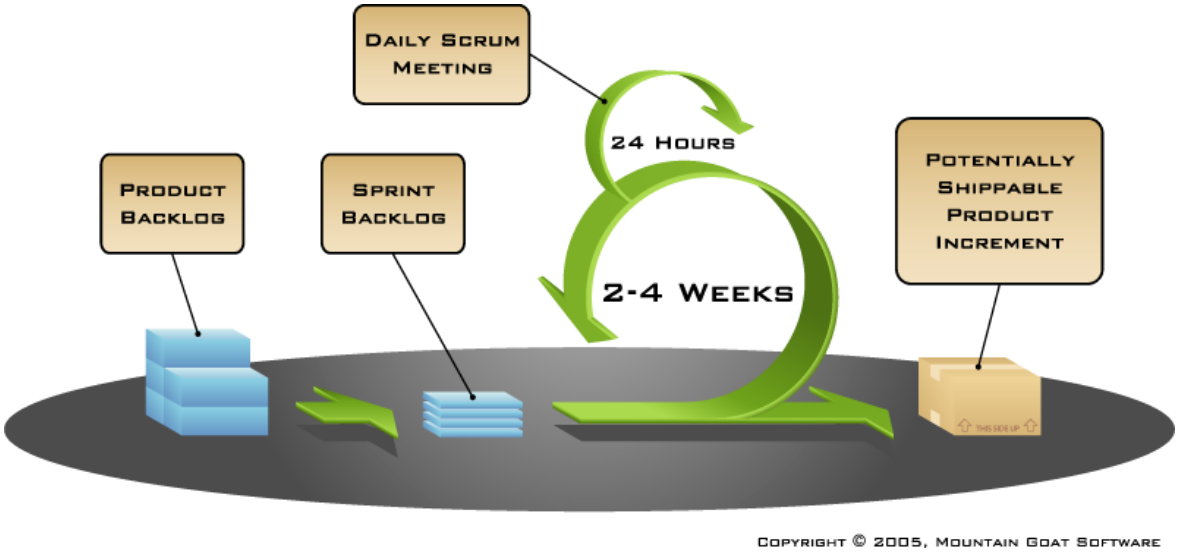
1. Tärkeintä on täyttää asiakkaan vaatimukset jatkuvilla ja riittävän aikaisilla ohjelmistotoimituksilla.
2. Muuttuvat vaatimusmäärittelyt hyväksytään ja otetaan vastaan tervetulleina, myös kehityksen loppuvaiheessa. Ketterät menetelmät valjastavat muutoksen asiakkaan kilpailueduksi.
3. Toimivia ohjelmaversioita toimitetaan säännöllisesti, muutaman viikon tai kuukauden välein, mieluummin useammin kuin harvemmin.
4. Liiketoiminnan ammattilaisten ja ohjelmistokehittäjien on työskenneltävä yhdessä päivittäin koko projektin ajan.
5. Projektit rakennetaan motivoituneiden yksilöiden ympärille. Heille annetaan ympäristö ja tuki jota he tarvitsevat, sekä luotetaan siihen, että he saavat työn tehtyä.
6. Kaikkein tehokkain tapa tiedonvälityksessä kehitystiimille ja kehitystiimin sisällä on kasvokkain tapahtuva keskustelu.
7. Toimiva ohjelmisto on ensisijainen työn edistymisen mitta.
8. Ketterät menetelmät suosivat kestävää kehitystä. Rahoittajien, kehittäjien ja käyttäjien tulisi kyetä pitämään jatkuvasti yllä tasainen työtahti.
9. Jatkuva huomion kiinnittäminen tekniseen erinomaisuuteen, hyvään rakenteeseen sekä suunnitteluun lisää ketteryyttä.
10. Yksinkertaisuus – tekemättömän työn maksimoinnin taito – on olennaista.
11. Parhaat rakenteet, vaatimukset ja suunnitelmat nousevat itseorganisoituvista tiimeistä.
12. Tasaisin väliajoin tiimi arvioi, miten voisi tulla entistä tehokkaammaksi, ja kehittää toimintaansa sen mukaisesti. (Huttunen 2006, 16.)

Ketterä ohjelmistokehitys on inkrementaalista (pienet versiojulkaisut tiheään tahtiin), yhteistyössä tapahtuvaa (asiakas ja kehittäjät toimivat jatkuvasti yhdessä ja pitävät tiiviisti yhteyttä), suoraviivaista (menetelmä itsessään on helppo oppia ja se on hyvin dokumentoitu) ja sopeutuvaa (on mahdollista tehdä viime hetken muutoksia). (Huttunen 2006, 16.)

Ohjelmoijien kommunikaatiotaitoihin liittyy myös kirjoittamattoman hiljaisen tiedon (tacit knowledge) siirtyminen henkilöltä toiselle: ketterissä menetelmissä periaatteena tiedon siirrossa on ihmisten välinen keskustelu, suunnitelmaohjautuvissa välineenä ovat paperit ja tiedostot. (Huttunen 2006, 16.)

2.5.1 Scrum

Esimerkkinä ketteristä ohjelmistokehitysmenetelmistä katsotaan tarkemmin miten ohjelmistotuotetta kehitetään Scrum menetelmän avulla.

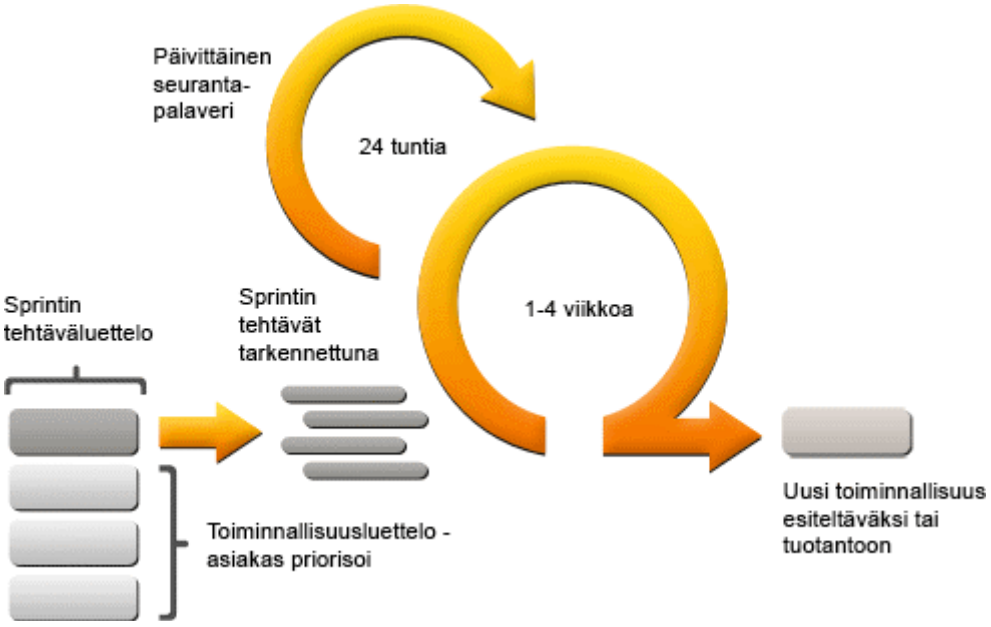


Kuva 8. Scrum graafisesti kuvattuna (Mountaincoat Software 2005)

Asiakas priorisoi toiminnallisuudet jotka halutaan kehittää ohjelmistotuotteeseen. Asiakas ylläpitää toivelistaa jota kutsutaan product backlogiksi.

Baseline on nykyinen tilanne tuotannossa, sisältäen vanhan ylläpidettävän koodin, johon tiimi kehittää uusia ominaisuuksia – tai kokonaan uuden projektin tai vain uuden komponentin joka lisätään tuotantoon. Baseline sisältää kaiken koodin, dokumentaation, määrittelyt, tekniset ratkaisut, arkkitehtuurin, bugit (tunnetut bugit jotka on päätetty jättää korjaamatta ja bugit joita ei ole vielä löydetty). Baseline sisältää myös teknisen- ja laatuvelan (technical depth and quality depth).

Iteraation eli sprintin alussa tiimi ja asiakas yhdessä valitsevat backlogista korkeimmalle priorisoidut ominaisuudet (sprint backlog) jotka tiimi kehittää Scrum spintin aikana valmiiksi ohjelmiston osaksi. Sprintin lopussa tiimi on saanut valmiiksi ohjelmiston osan (potentially shippable product ingrement) joka voidaan sellaisenaan asentaa tuotantoon. Valmis ohjelman osa on valmiiksi suunniteltu, koodattu ja testattu. Sprint kestää ideaalitapauksessa 2-4 viikkoa. Joka päivä pidetään scrum kokous (daily scrum meeting) ja kokouksen aikana selviää mitä tiimi tekee, miten tuotteen kehittäminen edistyy. Scrum menetelmä parantaa näkyvyyttä tiimin toimintaan, tiimin toimintaa haittaavat esteet ratkaistaan heti kun ne havaitaan. Scrum menetelmässä on sisäänrakennettu palautesykli. Sprintin lopussa järjestetään retrospekti eli palaveri, jossa tiimi erityisesti tarkastelee toimintaansa ja pyrkii kehittämään sitä muuttamalla toimintatapaansa. Tämä sykli jatkuu Sprint kerrallaan niin kauan kuin tuotteen omistaja (Product Owner) näkee hyödylliseksi investoida jatkokehitykseen.



Kuva 9. Prosessina Scrum on erittäin yksinkertainen ja voidaan helposti kuvata yhdellä kaaviolla (Koskela 2009)

Scrum on yksinkertainen selittää ja helppo ottaa käyttöön.

2.6 Lean production system

Lean on alun perin tuotantojärjestelmä, jota Toyota on menestyksekkäästi käyttänyt autojen valmistuksessa (Ogbeide 2008). Lean toimintaperiaate on käyttökelponen myös muissa valvotuissa menetelmissä. Lean tuotantojärjestelmän menetelmät eivät sellaisenaan suoraan sovi ohjelmistosuunnitteluun. Hyvän ohjelmiston suunnitteleminen ei ole tuotantoprosessi

(production process), vaan se on kehitysprosessi (development process). Voidaan ajatella, että kehitysprosessissa luodaan resepti jota tuotantoprosessissa noudatetaan. (Poppendieck 2003.)

Lean on yksinkertainen selittää, mutta vaikea ottaa käyttöön. (Van Cauwenberghe, P. & Tung, P. 2009).

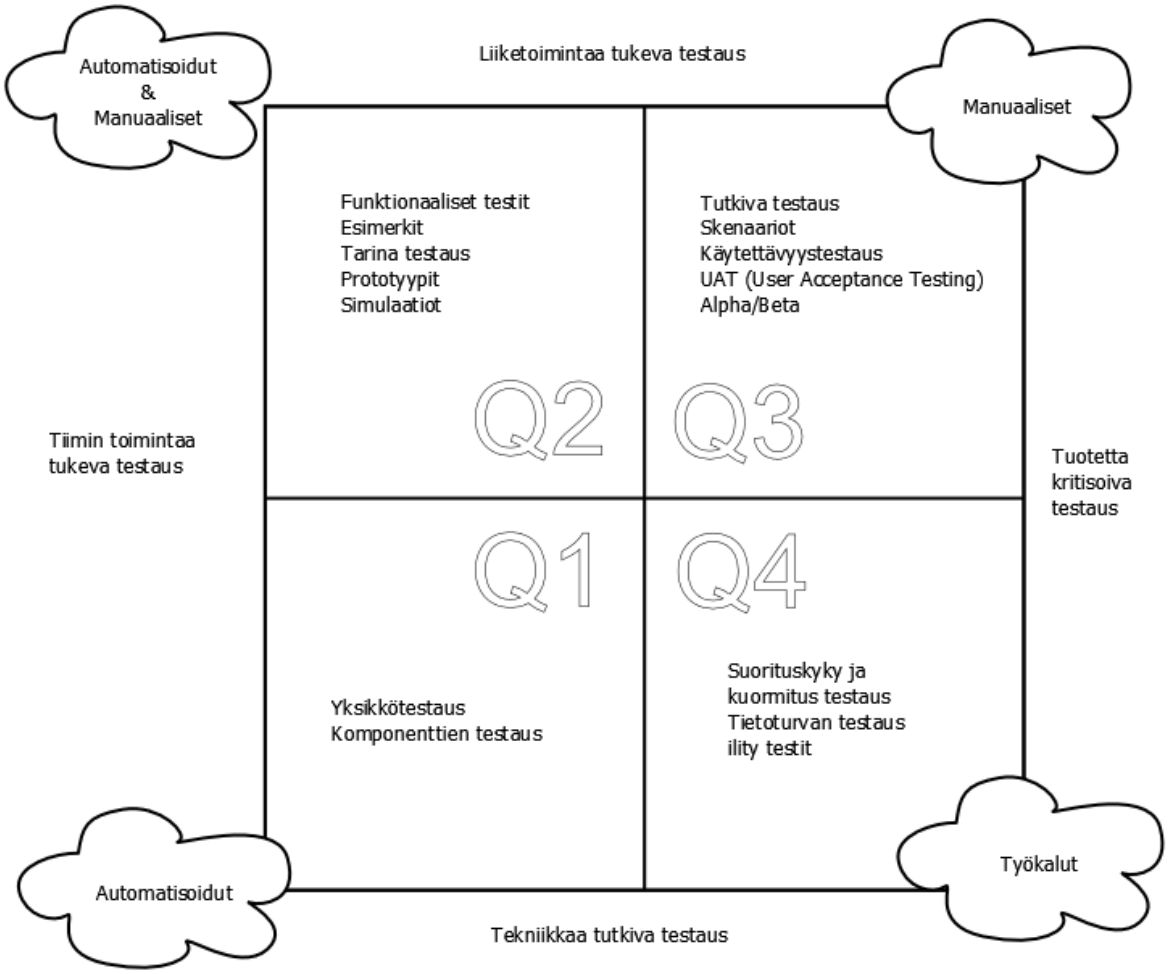
Lean tuotantojärjestelmän mukaan kaikki mikä ei tuota lisäarvoa asiakkaalle on tuhlausta (waste) (Ogbeide 2008). Waste käsite näyttää olevan kaikille erilainen ja aiheuttaa toisistaan poikkeavia tulkintoja. Ohjelmistoprojektissa ei ole aina selvää mikä on tuhlausta.

Kolme suurinta ohjelmistoprojekteissa yleisesti tehtävää tuhlausta ovat Allan Wardin mukaan: 1. Keskeneräisten töiden luovutukset (Hand-offs) ja kaikki tilanteet joissa on erotettu vastuu (mitä tehdään), tieto (kuinka tehdään), toiminta (itse tekeminen) ja palaute (tuloksista oppiminen). 2. Organisaatio on hajaantunut eri paikkoihin ja hajaantumisesta seuraa tiedonkulun katkos. 3. Toiveajattelu; arvataan mieluummin kuin kerätään faktatietoa. Myös testaaminen pelkkää määrittelydokumenttia vastaan voidaan katsoa tuhlaukseksi. (Ward, A. 2007.)

2.7 Testaus ketterissä projekteissa

Ketterät menetelmät eivät varsinaisesti tunne testaajan roolia vaan tiimin jäsenet ovat moniosaavia (cross-functional) ja tiimi on itseohjautuva. Tämä tarkoittaa sitä, että tiimi itse päättää kuinka Sprintin tehtävälistan tehtävät organisoidaan ja jaetaan.

Tiimi on yhdessä (whole-team approach) vastuussa testauksesta. Tiimissä voi olla testaukseen erikoistunut jäsen, hän pääsee testaukseen mukaan jo yksikkötestausvaiheessa. Ketterillä menetelmillä tehtävä kehitys on iteratiivista ja inkrementaalista. Testaus voidaan aloittaa jo ennen kun mitään on koodattu kun käytetään testauslähtöistä ohjelmointitapaa (TDD). Jokainen tarina tai toiminto testataan heti kun se valmistuu.



Kuva 10. Agile Testing Quadrants (Crispin 2009; 98)

Lisa Crispin on esittänyt erityyppisiä ketterillä menetelmillä tehtävissä projekteissa tarvittavia testejä kuvassa 'Agile Testing Quadrants' (Kuva 10. Agile Testing Quadrants (Crispin 2009; 98)). Kuva auttaa vastaamaan kysymykseen "Miksi testaamme?". (Crispin 2009.)

Vasemmalla kuvassa (Kuva 10. Agile Testing Quadrants (Crispin 2009; 98)) nähdään tiimin toimintaa tukeva testaus ja oikealla tuotetta kritisoiva testaus. Kritisoiva positiivisessa mielessä, lähinnä varmistetaan, että valmistetaan oikeanlaista tuotetta.

2.7.1 Tiimin toimintaa tukeva testaus

Tiimin tekemä sprint alkaa siitä, että asiakas on priorisoinut käyttötapauksen (user story) jolla on korkein liikearvo (business value). Tiimi jakaa käyttötapauksen tarvittaviin palasiin (task), yhden palan tekeminen kestää yleensä korkeintaan vuorokauden. Kuva 10. Agile Testing Qua-

drants (Crispin 2009; 98) tyyppistä taulukkoa tai muistilistaa voi pitää apuna kun suunnitellaan tarvittavat testit.

Liiketoimintaa tukeva testaus

Ylhäällä kuvassa nähdään liiketoimintaa tukeva testaus. Ketterissä menetelmissä asiakas on koko ajan tiimin käytettävissä. Funktionaaliset testit, esimerkit, tarina testaus (story testing), prototyypit ja simulaatiot tukevat asiakkaan tuotteelle asettamia vaatimuksia. Kaikkia tähän ryhmään kuuluvia testejä ei voi automatisoida, mutta jotta testit tukisivat tiimin toimintaa, korkea automaatiotaso on välttämätöntä. Tähän ryhmään voidaan laittaa myös käyttöliittymätestit, jotka käyttävät stubeja ja mock objekteja. Asiakas voi niiden avulla käyttää valmiin näköistä käyttöliittymää ennen kuin varsinainen liiketoimintaluokan koodaus tehdään. (Crispin 2009.)

Kun muut kuin Lisa Grispin ovat esittäneet tätä lohkoista koottua taulukkoa, hyväksymistestit (acceptance tests) on laitettu tähän lohkoon. LisaGrispin katsoo testejä laajemmin (Crispin 2009;130). Hän jakaa hyväksymistestit osiin ja jakaa ne kaikkiin lohkoihin. Tässä lohkossa on liiketoimintaa tukevat testit jotka suunnitellaan ja kirjoitetaan ennen kuin sovellus on koodattu. Testit kirjoitetaan niin, että asiakas ymmärtää ne. Tässä osassa voidaan käyttää liiketoimintalähtöistä kieltä (business domain language).

Teknologiaa tutkivat testit

Yksikkötason testaus ja komponenttitason testaus nähdään kuvan alalaidassa, missä nähdään teknologiapohjaiset testit. Kannattaa huomioida, että kaikki yksikkö ja komponenttitason testit pitäisi olla automatisoitu. (Crispin 2009). Kuvasta puuttuvat komponenttien integrointitestit. Komponenttien kunnollinen integrointi, integrointitestien automatisointi, integrointitestien oikea käyttö eri testaustasoilla ja toisten tiimien toimittamien komponenttien kanssa on ketterien menetelmien toimivuuden kannalta yksikkötestaustakin tärkeämpää.

TDD

Ketterissä menetelmissä käytetään yleisesti testaus lähtöistä ohjelmointitapaa (TDD). Test-Driven-Development (TDD) on ohjelmointitekniikka, jossa automatisoidut testitapaukset kirjoitetaan ennen varsinaisen ohjelmistokoodin kirjoittamista, joten koodi automaattisesti

läpäisee testit. Testaajana toimii toinen ohjelmoija, kun ohjelmointityö tehdään pariohjelmointina. JUnit testausframework on yleisesti käytetty menetelmä testauslähtöisessä ohjelmointitavassa.

2.7.2 Testaus varmistaa, että tehdään oikeanlainen tuote

Oikeassa laidassa nähdään tuotetta kritisoivat testit. (Kuva 10. Agile Testing Quadrants (Crispin 2009; 98)

Liiketoimintaa tukeva testaus

Iteraation aikana asiakkaalle esitellään kehitetyt toiminnallisuudet (demo). Asiakas katsoo ja käyttää ohjelmaa, kommentoi ja antaa mahdolliset muutosehdotukset. Tähän sektoriin kuuluvat testiskenaariot, työnkulkutestit (workflow), tutkiva testaus (exploratory testing), istuntopohjaiset testit, käytettävyystestaus, navigointi, käyttöohjeet ja online-ohjeet, raporttien testaus, lokitiedostojen monitorointi, testidatan luonti. Sovellusta voi myös käyttää vaikka jonkun API:n kautta, verkosta, näppäinkomennoilla tai jotenkin muuten ilman käyttöliittymää (UI). (Grispin 2009.)

Teknologiaa tutkivat testit

Ei-toiminnallisten vaatimusten (nonfunctional requirements) testaukseen kuuluvat turvallisuus, suorituskyky, muistinhallinta, luotettavuus, yhteentoimivuus, skaalautuvuus, palautuvuus, tietojen konversio, konfikuraation testaus ym. Näistä on hyvä olla tarkistuslista vaikka kaikissa projekteissa näitä ei tarvita. Nämä vaatimukset ovat sellaisia, että moni asiakas luulee toimittajan hoitavan nämä automaattisesti. (Grispin 2009.)

Turvallisuus on tärkeä tekijä suuressa osassa ohjelmissa. Turvallisuustestaus kuuluu kaikkiin kuvassa (Kuva 10. Agile Testing Quadrants (Crispin 2009; 98)) nähtäviin lohkoihin. Tähän neljänteen lohkoon kuuluvat vielä testit jotka testaavat tietoturvahaavoittuvaisuuksia, joita hakkerit käyttävät. Jos nämä vaatimukset eivät täyty tarvitaan iso ja kallis muutos kehitettävään ohjelmaan. (Grispin 2009.)

Tietoturvatestauksessa ja suorituskykytestauksessa kannattaa käyttää asiantuntijoiden apua. Tietokanta-asiantuntijat auttavat konversioissa, tietoturva-asiantuntijat auttavat arvioimaan

riskejä ja tuotannontukitiimi auttaa palautumisessa ja kun testit kaatuvat. Tiimin jäsenet voivat kehittyä näillä alueilla itsekin asiantuntijoiksi. Aina kun nämä testit eivät ole kunnossa ketteriä menetelmiä käyttävä tiimi tuo asian näkyväksi. (Grispin 2009.)

Suorituskyky, skaalautuvuus, palautuvuus ja kuormitustestaus pitää aloittaa jo projektin alussa (Grispin 2009; 239).

3 Automatisoinnin mahdollisuudet

3.1 Mitä testauksen automatisointi on?

Automatisointi on manuaalisen testauksen suorittamista koneellisesti, jonkin testausohjelman avulla. Ennen kuin voidaan automatisoida, täytyy toimiva manuaalinen testausjärjestelmä olla olemassa. Sen täytyy sisältää yksityiskohtaiset testitapaukset ja odotetut tulokset, jotka perustuvat vaatimusmäärittelyihin ja suunnitteludokumentteihin. Toiseksi erillinen testausympäristö, jossa on testitietokanta, joka voidaan aina tallentaa uudelleen tietyssä vakiomuodossa siten, että testitapaukset voidaan suorittaa uudelleen, kun sovellukseen tulee muutoksia. (Pohjalainen 2003.)

Automatisoinnin avulla on voitu suorittaa muutamassa minuutissa testejä, jotka ovat aikaisemmin vaatineet aikaa useita tunteja. Parhaimmillaan automatisoitu testaus on pienentänyt testauksen kustannuksia 80 %. (Pohjalainen 2003.)

Onnistuakseen automatisointi vaatii hyvää testausosaamista. Testitoimenpiteiden automatisointi on ohjelmistokehitystä: tarvitaan standardeja ja ohjelmointikuria, kuten normaalissa ohjelmistokehityksessä. (Pohjalainen 2003.)

3.2 Mitä kannattaa automatisoida?

Usein toistettavat testitapaukset, kuten **aloitustesti joukko (Intake Test), savutestit (Smoke Test)** ja **regressiotestaus** ovat hyviä automatisointikohteita. (Tieturi 2003.)

Savutestit ovat positiivisia testitapauksia, testijoukko jonka tarkoitus on tarkistaa version perustoiminnallisuudet. Savutestien läpäiseminen on yleensä testausryhmälle merkki siitä, että testaus voidaan aloittaa. (Kaner, Bach, Pettichord 2002.)

Regressiotestaus tarkoittaa korjausten jälkeen jo suoritettujen testien suorittamista uudelleen, testien tarkoituksena on löytää korjauksen mahdollisesti aiheuttamat uudet virheet.

3.3 Mitä ei kannata eikä voi automatisoida?

Iso osa testaamisesta jää edelleen ihmisten tehtäväksi. Tutkivaa testausta ei voi luonteensa vuoksi automatisoida, eikä luovuuttakaan voi jättää koneiden tehtäväksi.

Tutkiva testaus (Exploratory testing) on tehokas tapa aloittaa testaus ja analysoida ohjelmassa olevia riskitekijöitä. Tutkiva testaus ei ole testaustekniikka vaan lähestymistapa testaukseen. Ominaista tekniikalle on samanaikainen testien suunnittelu, testien- ja testattavan ohjelman opiskelu ja oppiminen, jotta voidaan luoda hyviä testitapauksia. Kaikki tehdyt testaustoimenpiteet pitää kurinalaisesti merkitä muistiin. Apuna voi käyttää esimerkiksi nauhoittavaa työkalua.

Riskiohjattu testaus. Projektin alussa tehdään riskianalyysi, joka kertoo mihin asioihin pitää kiinnittää huomiota ja toteuttaa/testata ajoissa. Idea on korjata tai ennaltaehkäistä potentiaaliset ongelmat ja niistä aiheutuvat virheet ennen kuin niistä aiheutuu ongelmia. Jotta näin voidaan tehdä, täytyy testattavat asiat priorisoida. Tämä on oikea lähtökohta projektin ja sen testauksen suunnittelulle. Jos mikään riskeistä ei realisoidu, projektilla on hyvät mahdollisuudet onnistua.

Käytettävyys testaus. Loppukäyttäjä voi käyttää ohjelmaa yllättävällä tavalla. Ohjelmoija ja testaaja usein tuntevat ohjelman ja tiedostamattaan käyttävät ohjelmaa white box näkökulmasta.

Ohjelman tilan testaus on mahdollista mutta usein haastavaa. Ohjelma siirtyy tilasta toiseen. Ohjelmalle annettavat syötteet ovat jossakin tilassa hyväksyttäviä, mutta kun siirrytään seuraavaan tilaan ne voivat ollakin tyhjiä tai kelpaamattomia. Ohjelman tilan testauksessa testataan kaikki ohjelman tilat ja tarkastetaan järjestelmällisesti, että ohjelma toimii kaikilla syötteillä niin kuin sen pitääkin. Tilan testaus on erityisen vaikeaa Internet sovelluksissa, koska niissä esiintyy tilapohjaisia virheitä paljon. (Tieturi 2003.)

3.4 Testaus työkalut (Test Tools)

Testaustyökalut voidaan jakaa kahteen ryhmään.

3.4.1 Testauksen tukemisen välineet (prosessiautomaatio)

Välineet, joiden käyttö voi tukea sekä käsin suoritettavia, että automatisoituja testejä ja joiden käyttöön ei liity oletusta käsin testaamiselta välttymisestä. Väline antaa tietoa käsin suoritetun testauksen tilasta, raportoi testit ym.

3.4.2 Testiautomaatiovälineet (suoritusautomaatio)

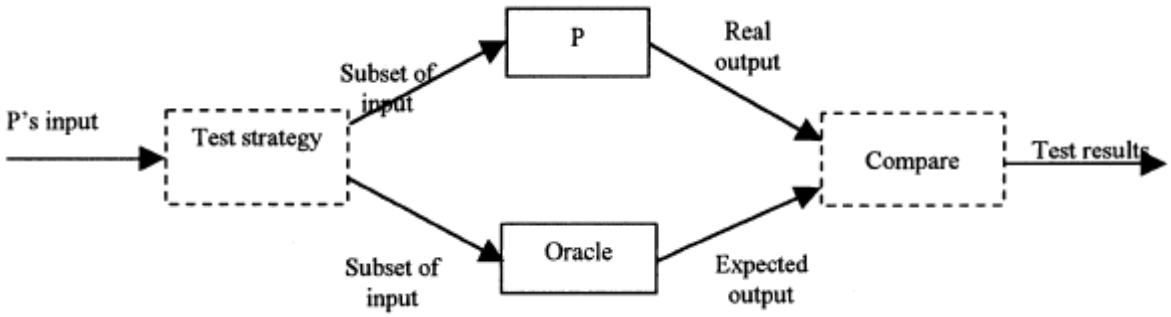
Välineet, joilla suoritetaan testausta tietokoneavusteisesti ja joiden käyttöön liittyy oletus käsin testaamiselta välttymisestä.

Keveimmillään ohjelmointi hoituu nauhoittamalla käyttöliittymässä tehtävät hiiren klikkaukset ja syöttökenttiin kirjoitettavat tekstit. Tämän jälkeen nauhoitusrobotti suorittaa annettua toimintoa vaikka loputtomiin.

Äänittävillä työkaluilla tehdyissä skripteissä on se ongelma, että käytettävä testidata on kovakoodattuna niiden sisällä. Uuden skriptin tekeminen on periaatteessa helppoa, se onnistuu kopioimalla vanha ja muokkaamalla sitä hieman. Tosielämässä testiskriptit eivät kuitenkaan ole niin yksinkertaisia eikä pientenkään muutosten teko välttämättä onnistu. Testin ajamiseen liittyy keskeisesti testiaineiston teko ja poisto eli testin palauttaminen alkutilaan. Suurempi ongelma tulee eteen myöhemmin kun testattava kohde muuttuu ja kymmenet tai sadat testiskriptit täytyy päivittää. Skriptien ylläpidosta tulee helposti niin iso ongelma, että testit on nopeampi koodata tai nauhoittaa uudestaan. Tällöin automaatiolla haetut uudelleenkäytön edut jäävät toteutumatta ja automaation hyödyt ovat kyseenalaisia.

3.4.3 Framework

Kun suunnitellaan testausta, kannattaa tarkastella miten voidaan rakentaa oma testausframework. Kun automatisoituja testejä kirjoitetaan ja ajetaan, tarvitaan useissa testeissä aina samat askeleet. (Meszaros 2007; 298.)



Kuva 11. Yksinkertaistettu testausprosessi (Manolache, Kourie, 2002)

Testausprosessin kulku yksinkertaistettuna:

1 suunnittele testi
2 äänitä testi
3 aja testi
4 vertaile tulos
5 palauta testi alkutilaan (recover), myös liittyvät järjestelmät

Vaikein ja tärkein vaihe on oikean tuloksen vertailu (compare). Frameworkin kehittäminen kannattaakin aloittaa tästä vertailuoraakkelin kehittämisestä. Testi voidaan ajaa käsin, ja oraakkeli vertaa onko testi oikein.

Testitapaukset (test case) kootaan usein suuremmiksi kokonaisuuksiksi, testijoukoiksi (test suite), jotka testaavat kokonaisen alijärjestelmän. Positiiviset ja negatiiviset testitapaukset kannattaa pitää eri kokonaisuuksissa. Testijoukkojen ylläpito pitää olla helppoa.

Sovelluskehys (framework) kannattaa suunnitella sellaiseksi, että eri henkilöiden on helppo katselmoida (review) testejä. Koodi kannattaa pitää yksinkertaisena. Testiautomaatioon siirryttäessä kannattaa käyttää avointa koodausta (open coding), jossa toistettava koodi pidetään paikallaan, ei siis siirretä ohjelmointikirjastoon, kuten normaalissa ohjelmointiprosessissa. Testien katselmoinnit kannattaa ottaa tavaksi. Testaus kannattaa suorittaa rajapinnan kautta. (Kaner, Bach, Pettichord, 2002: 109-124.)

Vielä muutama vuosi sitten testaussovelluskehys piti rakentaa itse, mutta nykyään on olemassa valmiita hyviä testaussovelluskehyksiä. On suositeltavaa käyttää valmista testaussovelluskehystä. Esimerkkinä testauksen kannalta hyvästä sovelluskehyksestä on Spring sovelluskehys, joka on alusta asti rakennettu testausvetoisesti. Yksikkötason testit voidaan ajaa

jo ennen kuin sovellus on julkaistu palvelimelle ja myös integrointitestit voidaan automatisoida helposti.

3.4.4 Epäonnistuneita testaamisen automatisointikokemuksia

Testausautomaatioprojektit epäonnistuvat usein. Yksi epäonnistumisen syistä on väärin valittu työkalu. Toinen merkittävä epäonnistumisten syy on testauksen väärä kohde. Esimerkiksi käyttöliittymien kautta tehtävä testien automaattinen suoritus on yleistä, mutta harvoin kustannustehokasta, toisin kuin työkalujen myyjät väittävät. (Kaner 1997.)

Myös ketterien menetelmien projekteissa on kokemuksia epäonnistumista. Tässä lueteltuna 10 yleisintä testausvirhettä, jotka XP-tiimi voi tehdä:

- yritetään toimittaa automatisoidut yksikkö testit hyväksymistestauksen tilalle
- ei automatisoida hyväksymistestejä
- luullaan, että automatisoidut testit riittävät
- annetaan asiakkaan hyväksyä tuote testaamatta sen ominaisuuksia
- annetaan asiakkaan määrätä jokainen yksityiskohta
- aliarvioidaan integrointitestauksen tarve
- aliarvioidaan ohjelman tilatestauksen tarve
- unohdetaan ei-toiminnalliset ominaisuudet (käytettävyys, suorituskyky)
- negatiiviset testitapaukset poistetaan automatisoitujen testien joukosta.
- testausta ei oteta mukaan ohjelmiston suunnitteluvaiheessa (Hendrickson 2006.)

3.4.5 Vääriä oletuksia testiautomaatioon liittyen

Testaus on ”sarja toimintoja”

Itse asiassa, testaus on vuorovaikutusta, jota rytmittää ohjelmiston arviointi. Monet vuorovaikutuksista ovat niin monimutkaisia, moniselitteisiä ja muuttuvia, että niitä ei voida kuvata järkevästi etukäteen.

Testaus tarkoittaa samojen asioiden toistamista kerta toisensa jälkeen

Jos testitapauksella ei alkujaan löydy virhettä, on todennäköistä, että sillä ei koskaan löydy virhettä. Jos testeissä on vaihtelevuutta, kuten yleensä käsin testatessa on, myös jo olemassa olevien virheiden paljastuminen on mahdollista uusien syntyneiden lisäksi.

Voimme automatisoida testauksen toiminnot

Jotkut asiat ovat vaikeita koneelle, mutta helppoja ihmisille. Erityisesti testauksen tulosten arviointi ”onko tämä oikein” on moniulotteista ja kaikkia ulottuvuuksia ei voi automatisoida.

Automatisoitu testi on nopeampi koska se ei tarvitse ihmisen puuttumista asiaan

Kaikki automatisoidut testit vaativat ihmisen puuttumista asiaan, ainakin rikkinäisten testien korjaamisessa ja tulosten analysoinnissa – onko virhe testissä vai testattavassa sovelluksessa.

Automaatio vähentää ihmisten tekemiä virheitä

Automaatio vähentää joitain virheitä, mutta toisaalta luo toisenlaisia virheitä. Automaatiolla voi päästä eroon virheistä joita ihmiset tekevät, kun annetaan tehtäväksi lista yksinkertaisia ja puuduttavia tehtäviä.

Käsin tehtävän ja automatisoidun testauksen kuluja ja hyötyjä voidaan järkevästi verrata

Käsin testaaminen ja automatisoitu testaaminen ovat hyvin erilaisia prosesseja eivätkä kaksi tapaa suorittaa sama prosessi – ihminen kykenee näkemään asioita moniulotteisesti samaa prosessia suorittaessaan ja tämä ero on merkittävä. Tasan samoja testejä tuskin tehtäisiin käsin tai välttämättä olisi oikeasti tarpeenkaan tehdä käsin. Automaatio on osa moniulotteista hyvää testausstrategiaa.

Automaatio johtaa merkittäviin resurssi- ja kustannussäästöihin

Automaation kustannus muodostuu automaation kehittämisestä, käyttämisestä, ylläpitämisestä ja automaation aiheuttamien uusien tehtävien tekemisen kustannuksista (testitapauksien tarkempi dokumentointi, automaation testaaminen ja dokumentointi, tulosten läpikäynti, muutosten analysointi automaatiovaikutusmielessä, jaetun automaatiotestijakson kehittämisen koordinointi). Automaatiolla ei yleensä vähennetä työtä.

Automaatio ei vaikuta heikentävästi testausprojektiin

On vaarallista automatisoida jotain mitä ei ymmärretä. Testausstrategia ja automaation osuus siinä pitää ymmärtää ennen automaatiota, tai tekninen ihmetys saattaa peittää alleen todellisen tavoitteen eli sovelluksen testaamisen. (Bach 1999, Pyhäjärvi 2005.)

3.4.6 Onnistuneita testauksen automatisointikokemuksia

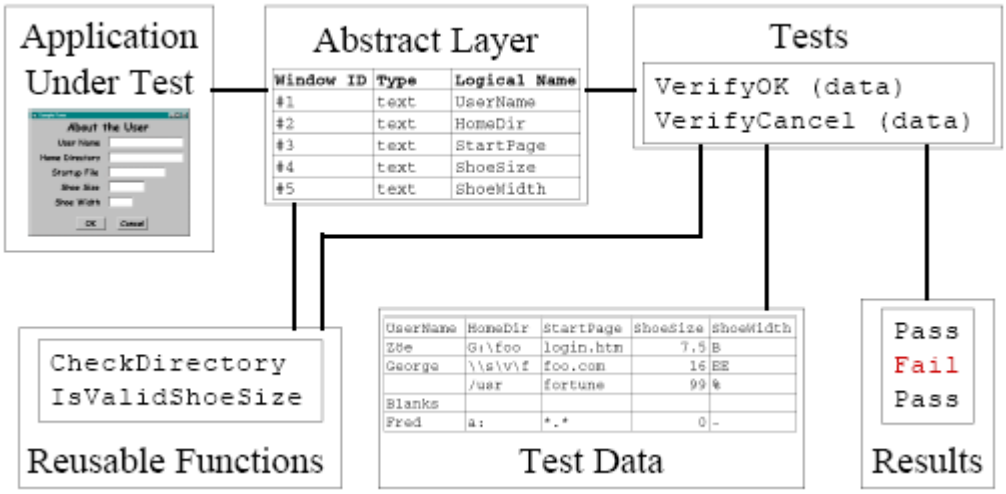
Mitä onnistuneessa projektissa tehtiin?

Elisabeth Hendricksonilla on kokemusta sekä onnistuneen, että epäonnistuneen testausprojektin johtamisesta. Hän on vertaillut eroja onnistuneen ja epäonnistuneen projektin välillä. Tässä on tiivistettynä onnistuneeseen projektiin vaikuttaneita tekijöitä.

- projektin tavoitteet oli asetettu ja tiimi oli tietoinen niistä
- manuaalista testausta ei poisteta vaan sitä autetaan automatisoimalla usein toistettavat testit
- projektilta ei odotettu tuloksia heti
- ryhmällä on testauskokemusta
- testausdokumentaatio ja testitapaukset ovat valmiina ja numeroitu yksilöllisesti
- laatupäälliköllä on automatisointikokemusta
- tiimin johtajilla (projektipäällikkö ja automatisointijohtaja) aikaisempia kokemuksia myös epäonnistuneista testausprojekteista
- tiimin johtaja osallistui projektin ohjelmointityöhön koko ajan
- tiimi kommunikoi ja raportoi viikoittain
- automatisointitestaustiimin rooli on ymmärretty siten, että tiimi on palveluorganisaatio ja se on keskittynyt apuvälineiden ja uusien työkalujen rakentamiseen (Hendrickson 1999.)

Projektin tarkoitus on nopeuttaa testauksen kiertoa ja tähän on kierron nopeuden mittaamiseen on osoitettu mittarit. Käytetään automatisointitestaustyökalua, jonka arkkitehtuuri on suunniteltu automaattisen testausprojektin käyttöön. Hyvä työkalu on suunniteltu niin, että testejä on helppo ylläpitää ja siirtää. (Hendrickson 1999.)

Automatisointityökalun elementit:



Kuva 12. Automatisointielementit (Hendrickson 1999)

Aineisto-ohjattua (data-driven) testaustyökalun käyttö näyttäisi toimivan hyvin, sen avulla luodaan skripti, joka käyttää tiedostossa olevaa testiaineistoa. Kun halutaan tehdä myös erityyppisiä testejä, on syytä viedä aineisto-ohjattu testaus astetta pidemmälle ja siirtyä avainsanaohjattuun (keyword-driven) testaukseen. (Hendrickson 1999.)

Hyvin suunniteltu skripti ja sen elementit

Taulukko 2. Hyvin suunniteltu skripti ja sen elementit (Hendrickson 1999)

Testiskripti sisältää asennuksen, testitapahtumat ja tuloksen. Testien suoritusjärjestystä voi vaihtaa. Testiaineisto ei ole kovakoodattua. Tulokset tarjoavat tietoa testistä Testin tuloksen (Pass/Fail)	Test Something [Setup Actions] *Setup Actions drive the application under test actions to the point where the test can be performed.* [Test Action(s)] *Test Actions perform the test.* [Verify Results]

päätteleminen on automatisoitu	*Verify Results evaluates the expected results and determines whether the test passes or fails.*

Gerald Meszaros lisää vielä jokaiseen testiin testin hajottamisvaiheen (Meszaros 2007, 358-361).

3.4.7 Onnistuneen projektin tunnusmerkit

Onnistumisen tunnusmerkit ovat säästö kokonaiskustannuksissa, tehokkaampi testaus, lyhyempi toimitusaika, luotettava lopputulos ja prosessien kehittäminen tulevaisuutta varten (Hendrickson 1999).

3.4.8 Testausstrategia (Test Strategy)

> *"Strategy: "The set of ideas that guide your **test design**."*
> *Logistics: "The set of ideas that guide your **application of resources** to fulfilling the test strategy."*
> *Plan: "The set of ideas that guide your **test project**."*
> *Plan = Strategy + Logistics"*

Testausprojektiin liittyy kiinteästi testin suunnittelu ja testiin liittyvä materiaalivirtojen ohjailu eli testiaineiston hallinta. Testiaineiston hallinta on testauksen automatisoinnin kannalta kriittinen tekijä. Automaattisen testauksen rakentaminen on tietokannan hallintaa (alkutila-vertailu-palautus backupilta). Kun automatisoitua testiä rakennetaan, tämä pitää voida tehdä nopeasti. Testin luontiympäristö ja testin käyttöympäristö on erotettava toisistaan.

Kysymyksiä, joihin pitää saada vastukset ennen automatisointia:

- Kuka ajaa automatisoidut testit? Kuinka usein?
- Kuka päivittää automatisoituja testejä?
- Miten testitulokset analysoidaan?
- Miten analysoidaan ovatko automatisoidut testit vielä toimivia ja tarpeellisia? Kuka poistaa tarpeettomat testit?
- Kuka suunnittelee ja laatii uudet automatisoidut testitapaukset?

Testausympäristön perustoimivuus on testien automatisoinnin tekemisen kannalta tärkein asia.

4 Havainnot, eli vastaukset tutkimuskysymyksiin

Seuraavaa tarkastelua ohjaavat havainnot ketterillä menetelmillä työelämässä tehdyistä testauksista sekä näitä tukeva kirjallisuus ja muut aineistot.

4.1 Mitä uutta ketterät menetelmät tuovat testaus- ja kehitystyöhön?

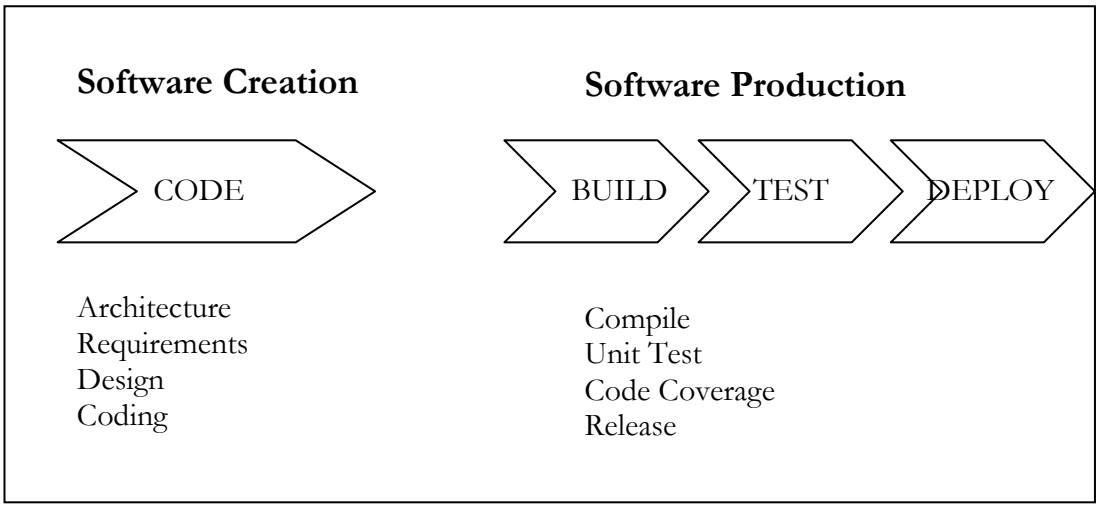
Johtaminen muuttuu, siirrytään käyttämään ketteriä käytäntöjä. Tiimiin luotetaan, tiimi on itse-ohjautuva ja tiimi yhdessä on vastuussa työskentelystään.

Tiimityöskentely tiivistyy, tiimin jäsenillä on muuttuneet roolit (product owner, scrum master, team member). Testaajan roolia ei välttämättä tunneta mutta tiimissä voi olla testaukseen erikoistunut jäsen.

Tiimin toiminta tehdään näkyväksi, tiimin hoitamat käyttötapaukset (user stories) ja tehtävät (tasks) ovat näkyvillä tietosäteilimessä (information radiator). Testit ovat myös näkyvillä. Testit ja ohjelmointi valmistuvat samaan aikaan. Suunnittelupalaveri (sprint planning), päivittäiset seurantapalaverit, retrospektiivinen palaveri (retrospective) sisältävät myös testauksen suunnittelun ja seurannan.

Tiimi tarvitsee toimivan kehitys ympäristön (tai monta). Tiimin on pystyttävä täyttämään määrittely Done (definition of done) periaatteen joka iteraatiokerralla. Ketterillä menetelmillä halutaan välttää hand-outit, ketterä tiimi vie itse suunnittelemansa ja valmistamansa toiminnon tuotantoon. Testien kehittäminen kannattaa aloittaa tekemällä aloitustesti joukko (intake test), joka testaa ohjelman käyttöliittymästä tietokantaan, kaikki kerrokset. Aloitustestijoukko ajetaan päivittäin, aina kun sovellukseen tehdään muutos. Tiimin pitää testata myös komponentin integrointi, systeemitestaus ja julkaisu tuotantoon.

Kehitysympäristössä tarvitaan jatkuva integraatio (continuous integration) ympäristö, joka rakentaa kehitettävän tuotteen joka päivä. Kun kaikki toimii vision mukaan, tiimi pystyy toimittamaan uuden version joka päivä tuotanto- ja integrointiympäristöön.



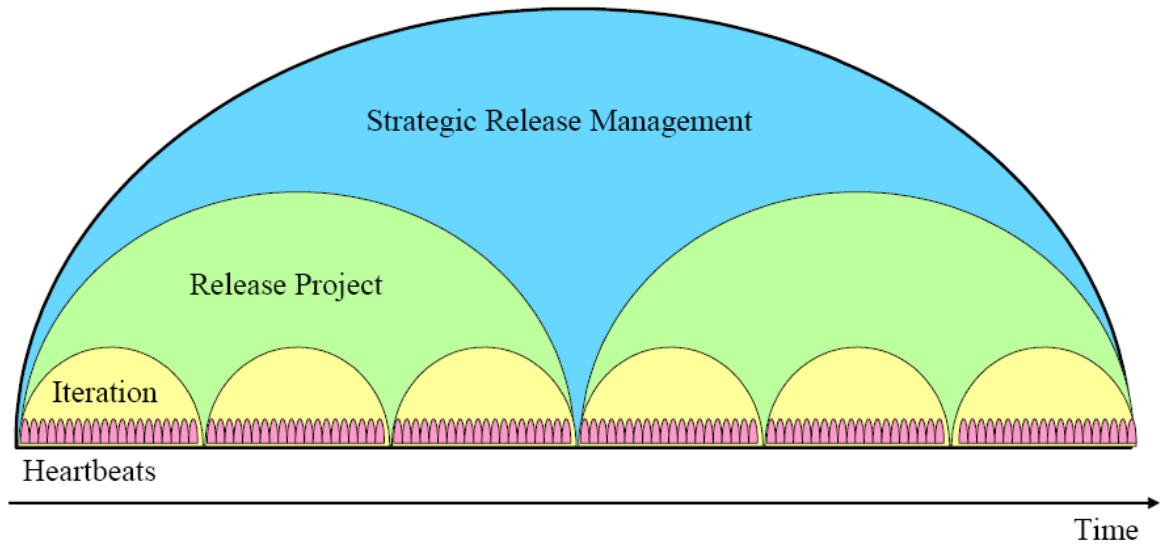
Kuva 13. Julkaisu joka päivä (Electric Cloud 2009)

Kuva 13. Julkaisu joka päivä nähdään tavoitetila, joka olisi testauksen kannalta ihanteellinen. Yllä oleva kuva on piirretty kehitys ja tuotantoympäristön hallintaa automatisoivan työkalun kehittäjän sivuston inspiroimana (Electric Cloud 2009). Täytyy muistaa, että kuva on piirretty valmistajan sivuston inspiroimana, mutta kuva selittää hyvin tilannetta, jossa uusi ketterä tiimi on. Joka päivä montakin kertaa julkaistaan uusi versio rakennettavasta ohjelmasta, kuvan kaikki vaiheet käydään läpi. Ohjelmisto on siis aina testattavissa automaattisesti ja käsin. On hyvä huomioida testien ajamisaika; suositus on, että kaikki testit voidaan ajaa puolessa tunnissa.

Arkkitehtuuri on hyvä olla suunniteltu ohjelmoinnin alusta asti. Tavoitteena on tehdä ohjelmistosta kapea siivu valmiiksi. Esimerkiksi, kun arkkitehtuuri on kerroksittainen, tehdään jokaisesta kerroksesta (3 kerrosmallissa käyttöliittymäkerros, liiketoimintakerros, tietokerros) ohjelmoitavaan toimintoon liittyvät toiminnot. Tavoite testata kaikki kerrokset, ja samalla kun toiminnallisuuksia lisätään iteraatioissa, testejä lisätään.

4.1.1 Testauksen heartbeats

Ohjelmistoa ja testausta samaan aikaan kehitettäessä kannattaa käyttää hyväksi ohjelmistokehityksen iteraatioita ja testauksen sykettä (heartbeats). Ideaalitapauksessa yksi heartbeat on yksi päivä sprintissä. Yksi iteraatio on yksi sprintti. Olisi ihanteellista jos hardbeat voisi rakentaa sovelluksen yksikkötestausympäristöön ja integraatioympäristöön. Joka päivä ajetaan kaikki automatisoidut testit ja joka päivä testejä lisätään.



Kuva 14. Testauksen heartbeats (Itkonen 2005)

Iteraation versiot voidaan myös asentaa eri palvelimille ja testataan viimeisintä versiota. Kun testattava sovellus joskus kaatuu, testiä voidaan jatkaa toisella palvelimella. Iteraatiot kannattaa numeroida niin, että sovelluksesta saa automaattisesti versionumeron virheilmoitukseen. Strategic Release Management kuvaa tiimin yhdessä tekemää testausstrategiaa.

4.1.2 Koodin tarkastukset (auditoinnit)

Ketteriä menetelmiä käyttävä tiimi ottaa käytännöksi säännölliset ohjelmistokoodin tarkastukset. Helpoin tapa aloittaa tarkastukset on pariohjelmointi. Tehokkaampaa on kuitenkin kun tarkastus tehdään toisen tiimin kanssa tai alan asiantuntija on mukana. Tietoturvan testaamisen kannalta koodin tarkastus on välttämätöntä, ja tarkastuksessa on hyvä käyttää tietoturvaan erikoitunutta asiantuntijaa. (Testing Experience, 2009, 16.)

4.1.3 Uuden tiimin haasteet käyttöönottovaiheessa

Uusi tiimi ei vielä osaa toimia tiiminä yhdessä ja voidaan tehdä virheitä. Esim.

- Kiire – testausta ei automatisoida vaan se jää käsin tehtäväksi
- Tiimi kehittää sovellusta, komponentteja, mutta ei tiedä miten ne julkaistaan
- Testistrategia ei ole selvillä, jokainen tiimin jäsen käsittää testauksen eri tavalla
- Testaus- ja/tai kehitysympäristö ei ole käytettävissä

– Mini waterfall (Crispin 2009; 47) – contininious integration ympäristö ei toimi oikein, testausta ei pystytä tekemään (järjestelmä ei ole testattavissa) vaan tiimi tekeekin kehitystyötä minivesiputousmallin mukaan. Testaukselle jää aikaa tunti sprintin lopussa.
– Cowboy coding – kehittäjät (omaksuvat vain haluamansa osat ketteristä menetelmistä)
– Tiimi keskittyy nopeuteen (Velocity) laadun (Quality) sijaan
– Kuva 10. Agile Testing Quadrants (Crispin 2009; 98) – osa tarvittavista testeistä puuttuu eikä niitä ole edes käsitelty

Kun uusi tiimi rakennetaan, tai aina kun halutaan tehdä iso organisaatiomuutos, asiaan sisältyy paljon riskitekijöitä.

Testauslähtöisen ohjelmointitavan käyttöönotto muiden muutosten ohella on myös riskitekijä. Lisa Crispin on luonut listan automaation esteistä:
– Ohjelmoijien asenne (testaajat testaavat ohjelman)
– Automaation oppiminen on vaikeaa alussa (oppimiskäyrä)
– Alkuvaikeudet testauksen aloittamisessa
– Ohjelmakoodi muuttuu erittäin usein
– Vanha ohjelmakoodi (legacy code)
– Pelko
– Vanhat tavat (Crispin 2009, 265.)

4.1.4 Organisaatiotason haasteet

Taipaleen tutkimuksesta selviää, että organisaatioiden suurin haaste on tiedonhallinta (knowledge management). Suurin kustannuserä ohjelmistokehityksessä on henkilöstökustannukset ja siksi saadaan merkittäviä säästöjä kun ohjelmointityö ohjataan halvempiin maihin (outsoursing). Outsoursing on vaikeaa kun ketterät menetelmät suosivat hiljaista tiedonhallintatapaa.

Taulukko 3. Miten perinteinen ja ketterällä menetelmällä toteutettu ohjelmistokehitys eroavat toisistaan (Taipale 2007; 26)

	Perinteinen	Ketterä menetelmä
Keskeiset olettamukset	Järjestelmä voidaan määritellä täydellisesti, kaikki on ennalta	Pienet ryhmät voivat luoda korkealaatuisia, mukautuvia

	arvattavissa. Järjestelmä voidaan rakentaa huolellisen ja laajamittaisen suunnitelman kautta.	ohjelmistoja käyttämällä jatkuvan suunnittelun, parantamisen ja testaamisen periaatetta nopean palautteen ja jatkuvan muutoksen avulla.
Hallinta	Prosessi keskeinen	Ihmiskeskeinen
Johtamistyyli	Hallitse ja kontrolloi	Johda ja tee yhteistyötä
Tiedonhallinta	Täsmällinen	Hiljainen
Roolien osoitus	Yksilökeskeinen – suosii erikoistumista	Itse ohjautuvat tiimit – rohkaisee roolien vaihtoa
Tiedonvälitys	Virallinen	Epävirallinen
Asiakkaan rooli	Tärkeä	Äärimmäisen tärkeä
Kehitysmalli	Elinkaarimalli (vesiputous, spiraali tai joku variaatio)	Evolutiivinen toimitusmalli
Toivottu organisaatiomuoto/rakenne	Mekaaninen (byrokraattinen ja muodollinen)	Orgaaninen (joustava ja osallistuva)
Teknologia	Ei rajoituksia	Suosii OO teknologiaa

Vesiputousmallissa ei ole mitään vikaa, mutta jos se ei sovellu organisaatiolle, siirtyminen ketteriin menetelmiin olisi hyvä tehdä hallitusti.

Positiivinen muutosprosessi organisaatiossa on vaikea toteuttaa. Antti Aro, John P. Kotter, Timo Erämetsä neuvovat miten organisaatiossa toteutetaan hallitusti myönteinen muutostilanne. Jos ajatellaan että Test-Driven-Developmentin käyttöön siirtyminen on haastavaa, voidaan tarkastella kuinka TDD:n käyttöön siirtyminen tehdään pienin askelin. Aloitetaan kartoittamalla nykytilanne ja tavoitetila. Nykytila on että vain muutama ohjelmoija käyttää TDD. Tavoitetila on että monta kaikki ohjelmoijat siirtyvät käyttämään TDD. Laaditaan suunnitelma, jossa asteittain hallitusti siirrytään TDD käyttöön.

Etenemissuunnitelma voisi olla tällainen:

- Etsitään konsultti, jotka hallitsee TDD:n käytön ja jolla on käytännön kokemusta vastaavista tosielämän projekteista
- Etsitään testauskonsultti, jolla on kokemusta epäonnistuneista automaatio projekteista (huonoista esimerkeistä oppii parhaiten)
- Valitaan pilottiprojekti

– Konsultit suunnittelevat ja koodaavat ja dokumentoivat osan projektin koodista TDD:tä käyttäen
– Ohjelmoijille esitellään hyvän ammatitaidon suuntia (orientaatio)
– Ohjelmoijille demotaan pilottiprojekti (tavoitetila konkretisoituu)
– Tarvittava koulutus järjestetään
– Koulutuksen jälkeen ohjelmoijat voivat soveltaa oppimaansa käytännön työssä
– Uuteen ohjelmointitapaan siirrytään hallitusti - seurataan projektien kannattavuutta
– Kehitystä seuraamaan asetetaan yhdessä sovittu mittaristo
– Kun rutiini uuteen ohjelmointitapaan saavutetaan, siirrytään isompiin projekteihin
– Kehitystä seurataan mittariston avulla
– Tavoitetila saavutetaan pienin askelin

Nämä kaikki askeleet pitää ottaa mukaan, jotta siirrytään myönteisen muutoksen portaita muutosgurujen mallin mukaan; ensin orientaatiovaihe, sitten tavoitetila konkretisoituu kun nähdään demo, sitten annetaan koulutus jne. (Aro 2002, Kotter 1996, Erämetsä 2003.)

4.2 Mitä on testattavuus (Design for Testability)?

Testattavuuden suunnittelu eli DFT (Design for Testability) pyrkii parantamaan sisäisten rakenteiden havaittavuutta ja ohjattavuutta siten, että sisäisten lohkojen funktiot voidaan testata. DFT on määriteltävissä seuraavasti: ”Mikä tahansa suunnittelurakenne, mikä voi parantaa kohteen näkyvyyttä testaustilanteessa, pienentää testauskustannuksia tai parantaa vikakattavuutta.”. (Koivukangas 2002.) Tämä määritelmä on elektroniikkateollisuudesta, mutta se sopii myös ohjelmistoteollisuudelle.

Testattavuus tarkoittaa sitä, että ohjelman sisäinen rakenne on sellainen, että automatisoidut testit voidaan ajaa. On hyvä käytäntö käyttää rajapintoja joiden kautta ajetaan tarvittavia testejä. Testattavuus tarkoittaa myös sitä komponentit rakennetaan niin, että testit voidaan ajaa rajapinnan kautta. Kannattaa rakentaa UI (user interface) sellaiseksi, että tarvittaessa se on helppo vaihtaa ilman liiketoimintakerroksen muutoksia. Näin testit ovat vielä käyttökelpoisia.(Kaner, C., Bach, J. & Pettichord, B. 2002; 119.)

Testattavuus on usein parempi investointi kuin automaatio. Testattavuutta lisää kaikki näkyvyys. Kun testaaja näkee lähdekoodin, kun testaaja näkee lähdekoodin muutokset.

Testattavuutta lisää esimerkiksi virheiden logitus, ohjelman datan ja muistin vianmääritysohjelmat, tapahtuman laukaisimet, testirajapinnat. (Kaner 2002, 122-123)

Testattavuutta voidaan lisätä myös käyttämällä ”Design-For-Testability Patterns” suunnittelumalleja. Käyttämällä malleja testejä voidaan ajaa hankalissakin tilanteissa.(Meszaros, 2007; 677.)

4.3 Minkälaista uutta tutkimustietoa löytyy TDD ym. menetelmistä

Test-Driven-Developmentissa kirjoitetaan testit yksi kerrallaan, vuorotellen tuotantokoodin kanssa. Test-Driven-Developmentissa on yleinen käytäntö, että kaksi ohjelmoijaa tekee koodauksen pariohjelmointina. Kun halutaan käyttää uudelleenkäytettäviä testimetodeja, ne voidaan sijoittaa yläluokkaan tai helper luokkaan. Test-Driven-Development käsittelee testikoodia kuten ohjelmointikoodia, testikoodi voidaan organisoida eri tavoin, testiaineistossa käytetään suunnittelumalleja (design pattens) ja katselmointeja aivan kuten tuotantokoodissakin. (Meszaros, 2007; 813, 638). Lähestymistapa testaukseen on tiukempi kuin vuonna 2002 kirjoitetussa kirjassa ”Lessons Learned in Software Testing ”, jossa suositeltiin aloittamaan testikoodin kirjoittaminen ilman monimutkaisia logiikkaa (Kaner 2002; 113).

Test-Driven-Development eroaa hieman Test-First-Developmentista. Test-First-Development ei ota kantaa siihen, että testi ja sitä vastaava tuotantokoodi kirjoitetaan vuorotellen. Kaikki testit siis voidaan kirjoittaa ennen tuotantokoodia. (Meszaros, 2007; 813.)

Empiirisen tutkimuksen mukaan Test-Driven-Development todennäköisesti parantaa ohjelman laatuominaisuuksia pienillä kustannuksilla. Varsinkin ohjelmakoodin kompleksisuus vähenee, ohjelmat ovat pienempiä ja paremmin testattuja. Nämä sisäiset laatuominaisuudet voivat huomattavasti parantaa ohjelman ulkoisia laatuominaisuuksia kuten vähentää virheitä ja parantaa ohjelman ylläpidettävyyttä, ohjelman ymmärrettävyyttä ja ohjelman uudelleenkäytettävyyttä. (Janzen, D. 2006.)

Tutkimukset eivät valitettavasti todista testausvetoisen ohjelmointitavan nopeuttavan ohjelmistokehitysprosessia merkittävästi nopealla aikavälillä. Kokeneet ohjelmoijat hyötyvät menetelmästä enemmän. Tämä on hyvä tiedostaa kun suunnittelee testausstrategiaa. (Proffit, J. 2008.)

4.3.1 ATDD (Acceptance Test-Driven Development)

ATDD eli hyväksymistestivetoisen kehityksen perusajatus on, että jo ennen ohjelmointia tehdään automaattisia hyväksymistestejä, joilla toteutettavan ohjelmiston toimintoja voidaan saman tien testata. ATDD rakentaa samalla ohjelmalle automaattiset hyväksymistestit.

4.3.2 DDD (Domain-Driven Design)

DDD menetelmässä mallinnetaan sovellusalue ja mallin mukaan valitaan sovelluksen arkkitehtuuri. Domain-kielet kuvaavat kehitettävää ohjelmaa niin, että Domain-ekspertti, joka tuntee käyttöalueen, pystyy lukemaan kieltä ja suunnittelemaan ohjelmaa mallintajan kanssa.

4.4 Miten ketterissä menetelmissä otetaan huomioon eri testaustasot?

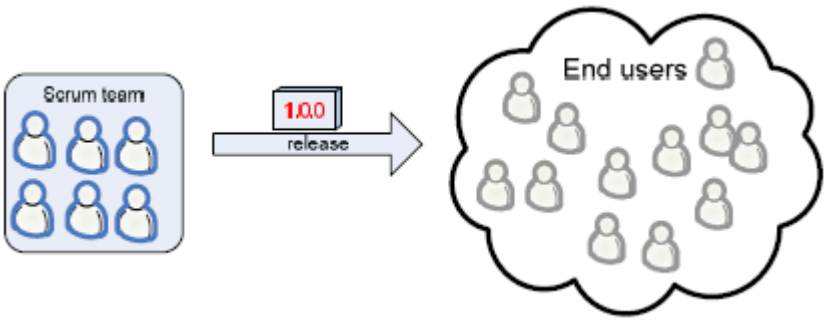
Perinteisissä menetelmissä testaajalla ei ole näkyvyyttä yksikkö ja integrointitestaukseen kehitysympäristössä. Testaaja joutuu tekemään yksikkötestauksen black box -menetelmällä varmistaakseen, että ohjelmoijan testaamat osat toimivat.

Taulukko 4. Testaustasot ja testaus

	Perinteinen	**Ketterä**
Yksikkö testaus	*	TDD +Tiimi + testaaja
Integrointi kehitysympäristössä	*	TDD +Tiimi + testaaja
Integrointi (Stage)	Test team	*
Systeemitestaus (Stage)	Test team	*
Hyväksymistestaus	Asiakas *	ATDD +Tiimi + testaaja
Release (Production)	Test team	*
Production	*	*

* = testaustaso, jonka näkyvyys on varmistettava testausstrategiassa

Ketterissä menetelmissä testaaja on tiimin jäsen ja näkyvyys yksikkötestaukseen on parempi. Tiimi joutuu oppimaan release-prosessin tarkemmin alusta loppuun asti. Harvemmin tiimin jäsenet tietävät mitä testaustiimissä on tehty. Nyt kaikki vastuu testauksesta siirtyy koko tiimille. Perinteisessä projektissa hyväksymistestaus on asiakkaan suorittama testaus ennen julkaisua. Ketterillä menetelmillä tehdyssä projektissa vastuu hyväksymistestistä on tiimillä.



Kuva 15. Uusi toiminnallisuus tuotantoon (Kniberg 2007)

Scrum ideaalimallissa uusi valmis toiminto on valmis julkaistavaksi sprintin lopussa. Todellisuudessa näin ei valitettavasti ole, tarvitaan paljon toimenpiteitä (etenkin testausta) ennen kuin toiminto on julkaistu menestyksellisesti.

4.4.1 Miten testaus poikkeaa eri testaustasoilla?

Stage on tässä tutkimuksessa integrointiympäristö joka ainakin periaatteessa pitäisi olla samanlainen kuin tuotantoympäristö. Stagellä ajetaan automatisoidut testit, mutta usein periatteena on, että testejä ei ajeta tuotannossa.

Taulukko 5. Testijoukot eri testaustasoilla

Testaustaso	Testit
Yksikkö testaus	– automatisoidut yksikkötestit - voidaan käyttää yksikkötesteissä virhetilanteiden simulointia – voidaan pakottaa metodeja testattaviin toimintoihin
Integrointi kehitysympäristössä	– automatisoidut yksikkötestit - voidaan käyttää yksikkötestien virhetilanteiden simulointia – voidaan pakottaa metodeja testattaviin toimintoihin - integrointitestit käyttäen mahdollisesti Mick objekteja

	- komponentin testattavuutta lisäävät testit - automatisoidut acceptance testit - testaus käsin (funktionaaliset testit, käytettävyys, ym.)
Integrointi (Stage)	– automatisoidut yksikkötestit – integrointitestit oikeita komponentteja vastaan – automatisoidut acceptance testit – testaus käsin (funktionaaliset testit, käytettävyys, ym.)
Systeemitestaus (Stage)	– koko järjestelmä testataan oikeita komponentteja vastaan – kehitystä varten rakennetut lisäosat on poistettu
Hyväksymistestaus (Stage)	– ennen tämä vaihe oli asiakaan vastuulla, nyt tiimin – onko rakennettu oikea tuote?
Julkaisu (Tuotanto)	– onko testit varmasti poistettu?
Tuotanto	– halutaanko seurata jotain?

Testausstrategiassa pitäisi suunnitella miten ohjelman testaus etenee tuotantoon. Miten testijoukkoja (Test Suite) ajetaan ja hallitaan ohjelmiston kehitysprosessin aikana?

Testidatan hallinta on tehtävä niin, että automatisoidut testit löytävät virheet. Mukana pitää olla myös negatiivinen testaus.

4.5 Miten testausta kannattaa kehittää tulevaisuutta varten?

Testausstrategia pitää olla kunnossa, tehty yhdessä tiimin kanssa (Grispin 2009; 91).

Testauksen hallinta on testijoukkojen hallintaa. Kun tuotetta kehitetään ohjelmointiympäristössä, ajetaan kaikki yksikkötestit aina ennen uutta julkaisua. Regressiotestijoukkoa kasvatetaan järjestelmällisesti. On tärkeää, että testit voidaan ajaa nopeasti. Testijoukkoja pitää olla monenlaisia, kun yksikkötestit on ajettu kehitysympäristössä, tarvitaan kunnolliset integrointitestit, jotka ajetaan muiden tiimien toimittamia komponentteja vasten. Testien kehittäminen kannattaa aloittaa tekemällä aloitustesti joukko (intake test), kun aloitustesti on ajettu onnistuneesti, ajetaan savutestijoukko (smoke test) (Tietotekniikan liitto 2009). Intake Test on aloitustestin erikoismuoto, jolla päätellään, onko komponentti tai järjestelmä valmis tarkempaan testaukseen. Aloitustesti suoritetaan yleensä testivaiheen alussa. Smoke Test on komponentin tai järjestelmän päätoiminnallisuuden kattava kaikista määritellyistä/suunnitelluista testitapauksista valittu osajoukko, jolla varmistetaan, että kaikkein

kriittisimmät ohjelman toiminnot toimivat, mutta pienempiin yksityiskohtiin ei kiinnitetä huomiota. Päivittäinen koonti ja aloitustesti (savutesti) ovat teollisuudenalan parhaita käytäntöjä.

Kannattaa huomioida, että yksikkötestit eivät ole aloitustestejä, yksikkötestit testaavat vain komponentin sisäistä rakennetta.

Taulukko 6. Vesiputousmalli toimii kun vaatimukset ovat selvillä

Esitutkimus	Määrittely	Suunnittelu	Toteutus	Integrointi ja Testaus	Käyttöönotto ja Ylläpito

Tiimin pitää huomioida edistyminen joka iteraatiokerralla ja objektiivisesti arvioida edistymisen arvo. Alussa aloitetaan toteutuksesta (koodaaminen), tehdään integrointi ja siihen liittyvä testaus. Joka iteraatikerralla käydään kuitenkin periaatteessa läpi vesiputousmallin kaikki vaiheet. Kun tiimi saavuttaa korkeamman kypsyystason huomataan, että voidaan ja halutaan kiinnittää enemmän aikaa määrittelyyn, esitutkimusvaiheeseen ja käyttöönoton helpottamiseen.

4.6 Mitä testattavuuden kannalta tärkeitä ominaisuuksia komponenttitason testauksessa voi olla mukana?

Tyypillisen komponentin testattavuus on huono. Ohjelmistoja rakennetaan komponenteista ja työn nopeuttamiseksi halutaan luoda uudelleenkäytettäviä komponentteja. Gross on kirjassaan esitellyt kuinka erilaisiin komponentteihin voidaan rakentaa testattavuutta merkittävästi parantavia ominaisuuksia joustavasti (Build-in Contract Testing) ja ottaen huomioon myös uudelleenkäytön helppous. Testirakenteet käyttävät testaamiseen rajapintoja tai testikomponentteja. (Gross 2005, 121-178.)

Ei ole realistista olettaa, että voidaan julkaista ohjelmisto jossa ei ole yhtään virhettä. Gross esittelee myös rakenteen (Build-in Quality-of-Service Runtime Monitoring), jonka avulla voidaan seurata ohjelman osien toimintaa integroinnin ja julkaisun aikana. (Gross 2005, 121-178.)

Kun ohjelmisto kehitetään ohjelmistoperhekonseptia (Product Family) käyttäen, kannattaa käyttää aikaa komponenttien testauksen suunnitteluun. (Gross 2005, 242-254.)

5 Pohdinta

Seuraavassa tehdään yhteenveto onnistuneen testauksen ominaisuuksista sekä hahmotellaan sitä, mitä etua ketteriin testausmenetelmiin siirtymisestä on, sekä minkälaisia tekijöitä menetelmän käyttöönotossa ja testauksessa kannattaa kiinnittää huomiota. Lopussa esitetään ehdotuksia jatkotutkimukseksi.

5.1 Testauksen onnistumisen kannalta keskeisiä tekijöitä

Onnistuneen testauksen näkökulmasta tärkeitä tekijöitä ovat testauksen mukaan ottaminen jo suunnitteluvaiheessa, selkeä testausstrategia, hyvin määritellyt ja sisäistetyt tavoitteet, testauksen automatisoinnin ja laadun korostaminen, oikein kohdentunut automaattisen ja manuaalisen testauksen kokonaisuus ja ketterien menetelmien kokonaisvaltainen käyttöönotto.

Testauksen epäonnistumisten tai puutteellisuuksien sekä uuden tiimin tarkastelun perusteella onnistuneen testauksen kannalta keskeistä on myös testauksen mieltäminen palvelutoimintona, kokeneen testausta ja ohjelmistokehitystä tuntevan tiimin johtajan tai valmentajan intensiivinen mukana oleminen, eri testaustekniikoiden tarkoituksenmukainen käyttö, erilaisten testijoukkojen (test suite) oikeanlainen hallinta eri testaustasoilla, ammattitaitoinen projekti- ja testaushenkilöstö sekä tiimin säännöllinen kommunikointi.

Testausympäristön perustoimivuus on testauksen automatisoinnin tekemisen kannalta tärkein asia ja testauksen ja kehityksen hallinta suunnittelusta tuotantoon erilaisissa ympäristöissä (kehitys, integrointi, tuotanto ym.) kriittisin tekijä. Testausta automatisoidessa aineisto-ohjatut ja avainsanaohjatut työkalut ovat toimivia ratkaisuja.

Empiiristen tutkimusten perusteella testivetoinen kehittäminen (Test-Driven-Development) näyttäisi vaikuttavan testauksen ja ohjelmien laatuominaisuuksiin: virheet vähenevät ja ohjelman ylläpidettävyys, ymmärrettävyys ja uudelleenkäytettävyys paranevat.

5.2 Ketterät menetelmät

Siirtyminen käyttämään ketteriä menetelmiä tuo organisaation, tiimin ja yksilöiden kannalta työhön paljon hyvää. Tiimi saa kehittää toimintojaan yhdessä. Kauan korjaamatta olleet bugit,

jotka ovat haitanneet tiimiä toimimasta tehokkaasti, voidaan korjata. Testaaja pääsee vaikuttamaan testaukseen jo ennen kun vaatimusmäärittelyä on tehty ja jo ennen kun yksikkötason koodausta tehdään. Tiimi voi kehittää yhdessä toimintojaan, myös laatutoimintojaan, testitapausten suunnittelu ja katselmoinnit tehdään yhdessä. Tiimin jäsenet voivat lisätä Tehtäväluetteloon saa kuka vaan lisätä uusia toimintoja (esim. puuttuva testien automatisointi), quality depth (testit joita ei ole pystytty ajamaan). Koodaus aloitetaan ennen kuin vaatimusmäärittely on valmis. Ketteriä menetelmiä käyttävällä tiimillä on suora yhteys asiakkaaseen ja siihen kuinka asiakas käyttää sovellusta. Tekninen velka (technical depth) ja laatuvelka (quality depth) tuodaan näkyväksi. Usein tiimit siirtyvät käyttämään ketteriä menetelmiä ennen kuin koko organisaatio siirtyy ketteriin menetelmiin.

Toimiakseen kunnolla ketterien menetelmien käyttöönotto pitäisi tulla organisaation johdon taholta, koska suuren organisaation kaikkien tiimien tulisi toimia yhteen. Organisaation kannalta on tärkeää, että tiimien välinen kommunikointi toimii ja tiimit voivat tehdä työtään odottamatta toisen tiimin suoritusta tai joutumatta tekemään välillä töitä monessa tiimissä. Ketterän tiimin muodostamisessa on sääntönä pitää tiimit pieninä, koska pienet tiimit toimivat tehokkaammin (Sutherland 2003). Kun useimmissa organisaatioissa on enemmän kuin kahdeksan työntekijää, tiimejä paljon ja tiimien välinen kommunikointi on tärkeää. Tilanteessa, jossa kaikki tiimit refactoroivat koodia, integrointitestaus on todella tärkeää.

5.3 Toimiva kehitysympäristö ja automaatio avainasemassa

Ketteriä menetelmiä käyttävä tiimi kehittää ohjelmaa ominaisuus kerrallaan ja näin ei tuhlata resursseja turhien toimintojen kehittämiseen. Halutaan vastata nopeasti muutoksiin. Kaiken perusta on toimiva kehitysympäristö, julkaisuprosessin tunteminen ja nopea julkaisusykli.

Ohjelmiston laatua ja automatisointia korostetaan erittäin paljon ketteriä menetelmiä käyttävässä prosessimallissa. Ketterä ohjelmistokehitys voi käyttää testivetoista ohjelmistokehitysmallia. On myös mahdollista kehittää testejä vähitellen, korkeammalle priorisoidut testit lisätään ensin, ja testejä lisätään hallitusti kun ohjelmistokehitysprosessi etenee. Automaatio on avain ketterillä menetelmillä tehtävään ohjelmistokehitykseen. Tarvitaan automatisoituja testejä testaamaan ohjelmistokoodia, kehitettävien komponenttien integrointia ja ohjelmiston kehitysympäristöä. Jotta voidaan kehittää tuotetta, on ensin rakennettava automatisoitu jatkuva integraatio (continuous integration) ympäristö. On myös

hyvä huomioida, että tarvitsemme erilaisia testejä eri testitasoilla (Katso kappale 4.4 Miten ketterissä menetelmissä otetaan huomioon eri testaustasot?).

Kehitysympäristön pitää sallia automatisoitujen testien kehittäminen, ajaminen ja myös käsin tehtävä testaus. Ketteriä menetelmiä käyttävä tiimin on hyvä tuntea julkaisuprosessi, jotta uudet ominaisuudet saadaan julkaistua tuotantoon nopeasti.

5.4 Innovatiivisyys

Laatukulttuurin kehittäminen on avainasemassa jotta organisaatio voi toimia kilpailukykyisesti ja innovatiivisesti. Kypsyystasoltaan korkealla tasolla oleva tiimi pystyy vastaamaan muutoksiin nopeasti, siksi tutkimukseen on otettu mukaan CMMI kypsyysmallin tarkastelu. Mallin mukaan ei voida siirtyä innovatiivisen organisaation tasolle jos ei ole edetty portaittain. Seuraavalle tasolle eteneminen edellyttää edellisen tason käytäntöjen omaksumista. CMMI kypsyysmalli sopii hyvin käytettäväksi ketteriä menetelmiä käyttävälle tiimille, joka haluaa kehittää tiimin kypsyystasoa, koska mallin avulla voidaan ohjata kehitystä myös prosessi kerrallaan (Cepeda, S. 2005). CMMI mallin mukaan kypsyys on kuitenkin ennenkaikkea organisaation mittari. CMMI mallin mukaan kaikkien tiimien on oltava samalla kehitystasolla jotta organisaatio saavuttaisi tietyn kypsyystason (Katso kappale 2.2.1 Miten laatua voidaan rakentaa ja mitata?).

5.5 Testattavuus

Testaus on kallista, sen takia testausprosessia kannattakin kehittää tulevaisuutta ajatellen. Testauksen kehittämisen hyödyt tulevat ehkä esiin vasta seuraavassa projektissa. Mitä aikaisemmin ohjelmistokehitysprojektissa pyydetään testattavuuden kannalta tärkeitä ominaisuuksia, sitä helpompi ne on ottaa mukaan projektisuunnitelmaan ja budjetoida ne. Jos niitä ei budjetoida ja oteta mukaan aikatauluun, niitä ei luultavasti saada. (Kaner, Bach, Bettichord 2002, 161). Ketterän tiimin ei tarvitse budjetoida testattavuuden kannalta tärkeiden ominaisuuksien kehittämistä, vaan yhdessä Product Ownerin kanssa lisätään puuttuvat ominaisuudet back logiin ja priorisoidaan.

Ketteriä menetelmiä käyttävä tiimi voi kehittää testattavuuden kannalta tärkeitä ominaisuuksia ohjelmiin samaan aikaan kun tiimi kehittää ohjelmaa.

5.6 Testausautomaatio ketterissä menetelmissä

Manuaalinen testaaminen on käsityötä ja tulevaisuudessa manuaalisen testauksen osuus vähenee. Manuaalista testausta voi verrata käsityöläisverstaaseen, kun taas testausautomaatiota verrataan tehtaaseen. Tehtaan rakentaminen vaatii valtavia investointeja ja tehtaan rakentaminenkin pitää osata. Samoin on automaation luomisen laita: kyseessä on haastava ohjelmankehitysprojekti. (Systeemityö 2005-1.)

Kun ketteriä menetelmiä käyttävä tiimi rakentaa sovellusta komponenteista, pitää sovelluksen testausautomaatio rakentaa huolellisesti. Komponentin testauksen ja rakentamisen kehitystyö on hyvä aloittaa pilottiprojektilla ja kehittää kestäviä testejä. Komponentit pitää rakentaa alusta asti testattaviksi. Sovelluksen arkkitehtuuri ratkaisee paljon: helpoimmin onnistutaan silloin, kun ohjelman sisäiset ja ulkoiset rajapinnat ovat harkittuja ja vakaita. Ohjelmistojen arkkitehtuuri kannattaa rakentaa suoraan automatisoitavaksi. Tämä mahdollistaa laajempien ja parhaimmillaan helppokäyttöisempien automaatiorakenteiden luomisen. Sovellusten ja järjestelmien entistä laajempi keskinäinen yhteistoiminta asettaa uusia vaatimuksia rajapintojen kuvaamiselle. Järjestelmätason testaus tehdään myös ohjelmointirajapinnan läpi, käyttäen erilaisia skriptejä. Rajapinnan tulisi olla testaamisen kannalta merkityksellinen, eli sellainen, että sen kautta voidaan tarkkailla ja vaikuttaa juuri testauksen kohteena oleviin toiminnallisuuksiin. (Systeemityö 2005-1.)

5.7 Testausautomaatioon siirtyminen hallitusti

Ennen testiautomaation käyttöönottoa kannattaa tehdä perusteellinen menetelmien arviointi ja pilotointiprojekti. Tätä lähestymistä on lähes poikkeuksetta pidetty hyvänä sen tarjotessa taloudellisesti riskittömän ja testattavan järjestelmän omia erityispiirteitä huomioivan käynnistysvaiheen (Systeemityö 2005-1.). Ketteriä menetelmiä on hyvä käyttää harkiten ja niin, että organisaation kaikki tiimit hyötyvät uusista menetelmistä.

Työkaluvalintoja ei voi tehdä kevyesti. Henkilökunnan kouluttautuminen käyttämään jotakin tiettyä työkalua vie aikaa (esim. TDD ja muut työkalut, joita käytetään). Työkaluja ei voida joka vuosi vaihtaa, vaikka mielenkiintoinen väline löytyisi (Systeemityö 2005-1.). Ketterillä menetelmillä voidaan työkaluja kokeilla joustavasti ja voidaan edetä kohti oppivaa organisaatiota.

5.8 Jatkotutkimuskaavailut

ATDD (Acceptance-Test-Driven-Development) eli hyväksymistestivetoisen kehityksen ja DDD (Domain-Driven-Design)sovellusalueen mallintamisen tutkiminen jäivät sivuosaan tutkimuksessa. Nämä menetelmät näyttäisivät olevan nouseva kehitystrendi ketterillä menetelmillä tehtävässä ohjelmistokehityksessä ja testauksessa. Kummatkin tarjoavat mielenkiintoisia jatkotutkimuskaavailuja.

Ketterien menetelmien yleistyminen on luonut tarpeen uusille ketteriä menetelmiä paremmin palvelevalle arkkitehtuureille. Suunnitteluvetoisia menetelmiä (Personal Software Process, Team Software Process, Rational Unified Process) käyttävien prosessien suosimat arkkitehtuurimallit, joihin tässä tutkimuksessa viitataan, ovat liian raskaita ketterillä menetelmillä rakennettaville ohjelmistoille.

Lähteet

Aro, A. 2002: Yritän vain hoitaa omaa tehtävääni. Edita, Helsinki.

Boehm, B. & Basili, V.R. 2001. Software Defect Reduction Top 10 List. Luettavissa: http://ieeexplore.ieee.org/iel5/2/19363/00962984.pdf. Luettu: 20.9.2009.

Cepeda, S. 2005. CMMI – Staged or Continuous? Cepeda Systems and Software Analysis, Inc. Luettavissa: http://www.sei.cmu.edu/library/assets/cepeda-cmmi.pdf. Luettu: 10.9.2009.

Deloitte Oy, 2005. Konsultoinnin laadunvarmistus Deloitte Oy:ssä. Luottamuksellinen (materiaali saatu konsultointi kurssia varten) 2005.

Electric Cloud, 2009. Making the Business Case for Build and Test Automation. Webcast sponsored by Electric Cloud. Esitetty: 28.7.2009.

Erämetsä, T. 2003: Myönteinen muutos. Tammi. Helsinki. Vammalan kirjapaino Oy, Vammala.

Fenton, N. & Pfleeger, S. 1997. Software Metrics - A Rigorous and Practical Approach (second edition); International Thomson Computer Press, 1997.

Grispin, L. & Gregory, J. 2008. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley.

Gross, H-G. 2005. Component-based software testing with UML. Springler.

Hartman, A. 2006. Adaptation of Model-Based Testing in Industry. Luettavissa: http://www.cs.tut.fi/tapahtumat/testaus06/ Luettu: 18.10.2009.

Heckel, R. & Lohmann, M. 2003. Towards Model-Driven Testing, Electronic Notes in Theoretical Computer Science 82 No. 6 2003. Luettavissa: http://www.elsevier.nl/locate/entcs/volume82.html. Luettu: 20.3.2006.

Hendrickson, E. 1999. The Difference Between Test Automation Success and Failure - a tale of two automation projects, presented at the STARWest 1998 conference. Luettavissa: http://testobsessed.com/wordpress/wp-content/uploads/2007/01/dbtasaf.pdf. Luettu: 20.9.2009.

Huhtamäki, J. 2005. Hypermedian ohjelmointi- kurssin luentomateriaali. Tampereen teknillinen yliopisto. Luettavissa: http://matriisi.ee.tut.fi/hmopetus/hm-ohj/2005/pruju/hmohj05-132-143.pdf. Luettu: 3.10.2009.

Humphrey, W. 2003. Introduction to the team software process. Addison-Wesley. Reading.

Huttunen, J. 2006. Ketterän ohjelmistokehitysmenetelmän määrittely, vertailu ja käyttäjäkysely. Diplomityö, Teknillinen korkeakoulu 2006. Luettavissa: http://lib.tkk.fi/Dipl/2007/urn007665.pdf. Luettu: 12.9.2009.

Itkonen, J. 2005. Software Testing and Quality Assurance, Teknillinen korkeakoulu, Tietotekniikan osasto, 2005. Luettavissa: http://www.soberit.hut.fi/T%2D76.5613/2005/material/T-76.5613_Testing%20Fundamentals_part_I_2005.pdf. Luettu: 20.3.2006.

Itkonen, J. 2004. Testing in Time-Paced Software Development, Teknillinen korkeakoulu, TestausOSY, 6.9.2004. Luettavissa: http://www.pcuf.fi/sytyke/kerhot/testaus/arkisto/Itkonen2004-09-06.pdf. Luettu: 20.3.2006.

Itkonen, J. 2006. Tutkimuksen ja käytännön välisen kuilun partaalla. Systeemityö N:o 4/2006 s. 12-15..

Janzen, D. 2006. An Empirical Evaluation of the Impact of Test-Driven Development on Software Quality. University of Kansas. Luettavissa: http://works.bepress.com/djanzen/4/. Luettu: 20.10.2009.

Kaner, C., Bach, J. & Pettichord, B. 2002. Lessons Learned in Software Testing. John Wiley & Sons, Inc. Hoboken.

Kaner, C. & Bach, J. 2006. Black Box Software Testing. Online Software Testing Course. Florida Institute of Technology. Luettavissa: http://www.testingeducation.org/BBST/index.html. Luettu: 10.8.2009.

Kaner, C. 2004. The Ongoing Revolution in Software Testing. Florida Institute of Technology. Luettavissa: http://www.kaner.com/pdfs/TheOngoingRevolution.pdf. Luettu: 4.10.2009.

Kaner, C 1997. Pitfalls and strategies in automated testing. Computer, 4/1997. Luettavissa: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=585164&isnumber=12687. Luettu: 20.9.2009.

Kniberg, H. 2007. Scrum and XP from the Trenches. Luettavissa: http://www.infoq.com/minibooks/scrum-xp-from-the-trenches. Luettu: 10.8.2009.

Koivukangas, T., Loukusa, V. & Moilanen, M. 2002. Prosessori 11/2002. Testaus- ja testattavuussuunnittelun haasteet. Luettavissa: http://www.prosessori.fi/es02/arkisto/PDF/TESTAUS.PDF . Luettu: 10.8.2009.

Koskela, L. 2009. Scrum: Ketterien menetelmien markkinajohtaja. Luettavissa: http://www.ttlry.fi/ Luettu: 10.8.2009.

Kotter, J. P. 1996: Muutos vaatii johtajuutta. Rastor, Helsinki.

Lyndsay, J. 2003. A Positive View of Negative Testing, Article in a conference produced by Software Quality Engineering, STAREAST 2003. Luettavissa: http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=6920. Luettu: 20.9.2009.

Manolache, L. I. , Kourie D. G. 2002. Software testing using model programs Software - Practice and experience, 2001; 31:1211–1236.

Meszaros, G. 2007. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley. Upper Saddle River (NJ).

Myers, G., Sandler, C., Badgett, T. & Thomas, T. 2004. The Art of Software Testing. Second edition. John Wiley & Sons, Inc. 2004.

Nevalainen, R. 2005. Projektitoiminta. 2/2005 Luettavissa: http://www.pry.fi/images/Projektitoiminta_2_2005_vedos.pdf. Luettu: 10.8.2009.

Ogbeide, T. 2008. Jeffrey Liker: Lean on vastuuta tulevaisuudesta. Net – Fujitsun asiakaslehti 3/2008. Luettavissa: http://www.net-lehti.com/default.aspx?ContentID=188. Luettu: 20.10.2009.

Partanen A.: Olio-ohjelmien testaus. Pro gradu -tutkielma, Kuopion yliopisto, Tietojenkäsittelytieteen laitos, heinäkuu 2003. Luettavissa: http://www.cs.uku.fi/research/Teho/julkaisut.html. Luettu: 20.9.2009.

Poppendieck, T. 2003. Lean software development: an agile toolkit. Addison-Wesley, 2003.

Pohjolainen, P. 2002. Software Testing Tools, erikoistyön osa. PlugIT-Teho, 2002. Luettavissa: http://www.cs.uku.fi/tutkimus/Teho/SoftwareTestingTools.pdf. Luettu: 20.9.2009.

Pohjolainen, P. 2003. Ohjelmiston testauksen automatisointi; Pro gradu -tutkielma, Kuopion yliopisto, 2003. Luettavissa: http://www.cs.uku.fi/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf. Luettu: 20.9.2009.

Proffit, J. 2008. TDD Proven Effective! Or is it? Luettavissa: http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx Luettu: 29.10.2009.

Pöyhönen, E. 2003. Kuinka testaus voi selvitä resurssi ja aikataulupaineissa, TecIT Forum, Helsinki, 30.1.2003. Luettavissa: http://www.pcuf.fi/sytyke/kerhot/testaus/TecIT_Forum-Testauksen_selviäminen.pdf. Luettu: 3.2.2006.

Pöyhönen, E. 2005. Testauksen kehittäminen, Power Point esitys 2005. Luettavissa: http://www.testauskirja.com/slides/Eki3SSymposium.ppt. Luettu: 3.2.2006.

Pyhäjärvi, M. & Sakkinen, M. 2005. Jyväskylän yliopisto, Department of Computer Science and Information Systems, testauskurssi, 2005. Luettavissa: http://www.cs.jyu.fi/~sakkinen/testaus2005/kalvot/8.1_VaariaOletuksia.ppt. Luettu: 20.3.2006.

Robinson, H. 2000. Intelligent Test Automation. The Software Testing & Quality Engineering Magazine, September/October 2000, Luettavissa: http://www.geocities.com/harry_robinson_testing/robinson.pdf. Luettu: 12.02.2006.

Robinson, H. 2005. Model-Based Testing Home Page. Luettavissa: http://www.geocities.com/model_based_testing/. Luettu: 12.10.2009.

Salo, M. 2007. CMMI-malli ja sen soveltaminen ICT-palveluyrityksessä. Opinnäytetyö. Tampereen Ammattikorkeakoulu. Luettavissa: https://oa.doria.fi/handle/10024/11979 Luettu: 10.9.2009.

Sakkinen, M. & Pyhäjärvi, M. 2005. Jyväskylän yliopisto, Department of Computer Science and Information Systems, testauskurssi, 2005. Luettavissa: http://www.cs.jyu.fi/~sakkinen/testaus2005/kalvot/8.1_VaariaOletuksia.ppt. Luettu: 20.3.2006.

SEI, 2009. The Carnegie Mellon Software Engineering Institute website. Luettavissa: http://sei.cmu.edu/acquisition/start/work/index.cfm. Luettu: 10.10.2009.

Silén, T. 2001. Laatu, brandi ja kilpailukyky. WSOY.

Sutherland, J. 2003. SCRUM: Keep Team Size Under 7! Luettavissa: http://jeffsutherland.com/2003/02/scrum-keep-team-size-under-7.html . Luettu: 20.9.2009.

Systeemityö, 2005-1. Testaus Testauksen automatisointi. Systeemityö 2005-1. Luettavissa: http://www.pcuf.fi/sytyke/lehti/stsis.html#2005-1 Luettu: 10.9.2009.

Taina, J. 2004. Ohjelmistojen testaus, luentomateriaali. Testausprosessi, syksy 2004. Helsingin Yliopisto, Tietojenkäsittelytieteen laitos. Luettavissa: http://www.cs.helsinki.fi/u/taina/ohte/s-2004/luennot/Luku02.pdf. Luettu: 20.8.2009.

Taipale, O. 2007. Observations on Software Testing Practice (Havaintoja ohjelmistotestauksen käytännöistä). Lappeenrannan teknillinen yliopisto. Lappeenranta. Luettavissa: https://oa.doria.fi/handle/10024/31126. Luettu: 20.9.2009.

Testing Experience, 2009. Application Security Fundamentals. Luettavissa: http://www.testingexperience.com/testingexperience02_09.pdf. Luettu: 20.10.2009.

Tevanlinna, A. , Taina, J. & Kauppinen, R. 2004. Product family testing: a survey. University of Helsinki, Helsinki. Luettavissa: http://portal.acm.org/citation.cfm?id=979766 Luettu: 12.9.2009.

Tietotekniikan liitto 2009. ISTQP sanasto. Luettavissa: http://www.ttlry.fi/yhdistykset/osaamisyhteisot/fistb/glossary-sanasto/. Luettu: 12..2009.

Tieturi 2003. Testitapausten suunnittelu -kurssi, kurssimateriaali. Tieturi 2003.

Toroi, T. 2009. Testing Component-Based Systems - Towards Conformance Testing and Better Interoperability. (Komponenttipohjaisten tietojärjestelmien testaus – kohti määrityksen mukaisuuden testausta ja parempaa järjestelmien välistä yhteentoimivuutta) University of Kuopio. Luettavissa: http://www.uku.fi/vaitokset/2009/isbn978-951-781-994-7.pdf. Luettu: 13.9.2009.

Turun yliopisto 2007. Ohjelmistotuotanto. Laatujärjestelmät: toiminnan (prosessin) kehittäminen. Luettavissa: http://staff.cs.utu.fi/kurssit/ohjelmistotuotanto/kevat07/laatujarjestelmat.pdf. Luettu: 8.9.2009.

Van Cauwenberghe, P. & Tung, P. 2009. Seminaariesitys: The Toyota Way Management principles to sustain Lean and Agile. Scandinavian Agile Conference 15.11.2009. Tiivistelmä luettavissa: http://www.agilecoach.net/coach-tools/the-toyota-way/ Luettu: 20.10.2009.

Ward, A. 2007. Lean Product and Process Development. Lean Enterprises Inst Inc. 2007.

Virkanen, H. 2002. Ohjelmistojen testaus ja virheenjäljitys, Pro gradu -tutkielma. Kuopion yliopisto, 2002. Luettavissa: http://www.cs.uku.fi/tutkimus/Teho/graduvirkanen.pdf. Luettu: 30.9.2009.