TAMPEREEN TEKNILLINEN YLIOPISTO

PETTERI PALOMÄKI

TESTAUS OHJELMISTOTUOTANNON OSANA

Diplomityö

Tarkastaja: professori Hannu Jaakkola

Tarkastaja ja aihe hyväksytty tuotantotalouden ja rakentamisen tiedekunnan kokouksessa 9. marraskuuta 2016

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Johtamisen ja tietotekniikan koulutusohjelma
PALOMÄKI, PETTERI: Testaus ohjelmistotuotannon osana
Diplomityö, 46 sivua
Kesäkuu 2017
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu Jaakkola
Avainsanat: Ohjelmistotestaus, ohjelmistotuotanto

Testaus on kriittinen vaihe ohjelmistotuotannossa, jolla varmistetaan, että ohjelmistot ovat toimivia ja laadukkaita. Testauksessa löytynyt virhe on paljon helpompaa ja edullisempaa korjata, kuin vasta valmiissa ohjelmistossa ilmenevä ongelma. Näin ollen testaus on kannattavaa sekä taloudellisesti että yrityksen maineen kannalta.

Diplomityön tarkoituksena on tutustua ohjelmistotestaukseen akateemiselta näkökannalta. Tavoitteena on tutustua ohjelmistotestausta käsittelevään kirjallisuuteen ja tutkimuksiin ja esittää näiden avulla selkeä kokonaisuus tästä ohjelmistotuotannon osasta.

Diplomityössä tutustutaan aluksi ohjelmistotuotantoon yleisesti ja huomioidaan, miten testaus on mukana sen erilaisissa menetelmissä. Sen jälkeen perehdytään itse ohjelmistotestaukseen käymällä läpi testauksen periaatteita, sen menetelmiä, työvälineitä ja suunnittelua.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
PALOMÄKI, PETTERI: Testing as a Part of Software Engineering
Master of Science Thesis, 46 pages
June 2017
Major: Software engineering
Examiner: Professor Hannu Jaakkola
Keywords: Software Testing, Software Engineering

Testing is a critical part of software engineering. It aims to ensure, that the programs work and have high quality. An error which is found in testing is much easier and cheaper to fix than error that is found from the finished program. This means that testing benefits not only the company financially but also it's reputation

The intent of this master's thesis is to look into the software testing from an academic point of view. The goal is to study the literature and studies about software testing and use these to represent a clear picture about this part of the software engineering.

First this master's thesis looks into software engineering as a whole and looks how testing is a part of different methods used in software engineering. After that this thesis focuses on software testing by looking into testing principles, methods, tools and planning.

ALKUSANAT

Opiskelujeni aikana näin ohjelmistotestauksen tärkeänä ja mielenkiintoisena osana ohjelmistotuotantoa. Itse testaus jäi kuitenkin usein pieneksi osaksi kulloinkin käytävää aihetta ja sama tuntui toistuvan useissa ohjelmistotuotantoa käsittelevissä teoksissa. Halusin perehtyä ohjelmistotestaukseen enemmän ja lopulta päätin, että diplomityössäni voisin tutkia aihetta lisää.

Haluan kiittää diplomityön ohjaajaani professori Hannu Jaakkolaa hänen arvokkaista neuvoistaan ja palautteestaan tätä työtä tehdessäni. Lisäksi haluan kiittää vaimoani Susannaa kärsivällisyydestä ja kissojamme Tiberiusta ja Merliniä, jotka suostuivat edes välillä pysymään poissa näppäimistöltä.

Porissa 25.6.2017

Petteri Palomäki

SISÄLLYS

1 JOHDANTO....1
2 OHJELMISTOTUOTANTO....4
2.1 Vesiputousmalli....5
2.2 V-malli....6
2.3 Iteratiivisuus....7
2.4 RUP....8
2.5 Ketterät menetelmät....9
2.6 Testivetoinen kehitys....11
3 TESTAAMINEN YLEISESTI....13
3.1 Staattinen ja dynaaminen testaus....13
3.2 Etsittävät viat....13
3.3 Tarkastukset....14
3.4 Validointi ja verifiointi....15
3.5 Testausprosessi....16
3.6 Testausprosessi organisaatiossa....17
4 OHJELMISTOTESTAUS....20
4.1 Testauksen periaatteet....21
4.2 Testauksen tasot....22
4.2.1 Yksikkötestaus....22
4.2.2 Integrointitestaus....24
4.2.3 Järjestelmätestaus....26
4.2.4 Hyväksymistestaus....28
4.3 Testauksen tasot kypsyyden perusteella....30
5 OHJELMISTOTESTAUKSEN MENETELMÄT....32
5.1 Musta laatikko....32
5.2 Valkoinen laatikko....33
5.3 Harmaa laatikko....33
5.4 Regressiotestaus....33
5.5 Käytettävyystestaus....34
5.6 Kuormitustestaus....34
5.7 Savutestaus....34
5.8 Käyttäjätestaus....35
5.9 Alfa- ja Beta-testaus....35
5.9.1 Alfa-testaus....35
5.9.2 Beta-testaus....36
5.10 Ad hoc-testaus....36
5.11 Mallipohjainen testaus....36
5.12 Virheiden kylväminen....36

5.13 Mutaatiotestaus....37
6 OHJELMISTOTESTAUKSEN SUUNNITTELU....38
6.1 Suunnitteludokumentti....39
6.2 Testitapaus....39
6.3 Riskianalyysi....40
6.4 Testauksen lopetuksen määrittäminen....40
6.5 Ohjelmistotestauksen työvälineet....40
6.5.1 Kehitysympäristöt....41
6.5.2 Virheenjäljitin....41
6.5.3 Testausautomaatio....41
6.5.4 Vikatietokanta....42
6.5.5 Tynkä....42
6.5.6 Analysaattori....42
6.5.7 Dokumenttipohja....42
7 YHTEENVETO....44
LÄHTEET....45

KUVALUETTELO

Esimerkki tavallisesta ohjelmiston rakenteesta (Kasurinen 2013).....4
Projektin eteneminen vesiputousmallissa (Kasurinen 2013).....5
Projektin eteneminen V-mallissa (Kasurinen 2013).....6
Projektin eteneminen RUP-mallissa (Kasurinen 2013).....9
Projektin osien eteneminen Scrum-menetelmässä (Kasurinen 2013).....10
Projektin eteneminen testivetoisessa kehityksessä (Sommerville 2011).....12
Tarkastukset ja testaus osana ohjelmistokehitystä (Sommerville 2011).....14
Esimerkki perinteisestä testausprosessista (Sommerville 2011).....16
Yksinkertaistettu esimerkki testauksen läpiviennistä organisaatiossa (Ammann & Offutt 2008).....18
IEEE Computer Society jakaa ohjelmistotestauksen kuuteen osaan (IEEE Computer Society 2014).....20
Yksikkötestauksen kohteena on uusi komponentti järjestelmään (Kasurinen 2013).....23
Integrointitestauksessa yhdistetään valmistuneita komponentteja (Kasurinen 2013).....25
Järjestelmätestauksen kohteena on koko järjestelmä (Kasurinen 2013).....27
Esimerkki hyväksymistestauksen läpiviennistä (Sommerville 2011).....28
Musta laatikko ei kerro mitä testin aikana ohjelmistossa tapahtuu (Sommerville 2011) 32
Testauksen suunnittelussa huomioitavia tekijöitä (Haikala & Mikkonen 2011).....38

TERMIT JA NIIDEN MÄÄRITELMÄT

Agile Alliance	Voittoa tavoittelematon järjestö, joka edistää ketterien ohjelmistokehitysmenetelmien käyttöä maailmanlaajuisesti.
FiSTB	Finnish Software Testing Board on ISTQB:n Suomen paikallisjärjestö.
Hyönteismyrkkyparadoksi	Testaus menettää tehoaan, jos testejä ei pidetä ajan tasalla. Termi on vertaus sille, miten hyönteiset tulevat immuuneiksi hyönteismyrkylle ajan kuluessa.
IEEE Computer Society	Organisaatio, joka edistää tietotekniikan teoriaa ja käytäntöä.
ISO/IEC 29119	Kansainvälisesti hyväksytty standardi, joka määrittelee ohjelmistotestausta.
ISTQB	International Software Testing Qualifications Board on kansainvälinen ohjelmistotestauksen sertifiointiorganisaatio.
UML	UML on standardoitu mallinnuskieli, joka on kehitetty ohjelmistokehitystä ja erityisesti ohjelmiston suunnittelua varten.

1 JOHDANTO

Ohjelmistotestaus on yksi tärkeä osa ohjelmistotuotannon kokonaisuutta. Testaamiseen kuuluu yllättävän laaja määrä erilaisia työtehtäviä, jopa verrattuna varsinaiseen ohjelmointityöhön. Usein ohjelmistotestaus esitellään kirjallisuudessa hyvin irrallisena osa-alueena, josta lukija saa helposti käsityksen, että ohjelmistotestaus on jotain, mikä tapahtuu varsinaisen ohjelmistotyön jälkeen. Näin ei kuitenkaan ole, vaan onnistunut testaus kulkee käsi kädessä muiden työvaiheiden kanssa läpi koko ohjelmiston kehitysprosessin. Tämän diplomityön tarkoituksena on tutustua ohjelmistotestausta käsittelevään kirjallisuuteen ja tutkimuksiin ja koostaa näiden avulla kokonaisuus tästä ohjelmistotuotannon osasta. Diplomityön yhtenä tavoitteena on esitellä lukijalle ohjelmistotestauksen menetelmiä, työkaluja ja testauksen suunnittelua. Toinen tavoite osa on esittää, miten testaus näkyy osana ohjelmistotuotannon kokonaisuutta.

Ohjelmistotestauksen käsikirjan mukaan ohjelmistotestaus on ”työtä, jota tehdään sen varmistamiseksi, että toteutettavasta ohjelmistotuotteesta tulee toivotun kaltainen ja että kaikki valmiiksi saadut ominaisuudet varmasti toimivat niin kuin oli tarkoitus.” (Kasurinen 2013, s. 10) Puhekielessä testauksella tarkoitetaan yleensä mitä tahansa kokeilemista. Ohjelmistotuotannossa testaus määritellään yleensä suunnitelluksi virheiden etsimiseksi ohjelmistosta tai sen osasta. (Haikala & Mikkonen 2011)

Vielä 1950-luvulla tietokoneiden ohjelmat olivat hyvin yksinkertaisia ja nykyisenlaista ohjelmistotestausta ei tarvittu. Ohjelman testaamiseen riitti usein se, että matemaatikon todettiin saavan saman tuloksen, kuin mitä ohjelman tuloste ilmoitti. 1960-luvun alkupuolella tietokoneiden tehokkuus oli kuitenkin jo saavuttanut sen pisteen, että ohjelmia oli alettava myös testaamaan. Vuosikymmenen loppupuolella ohjelmistokehityksessä alkoi jo olla vaikeuksia ohjelmien monimutkaisuuden vuoksi. Saksassa otettiin käyttöön termi ”ohjelmistokriisi” kuvaamaan uutta ongelmaa ohjelmien tekemisessä, joka oli syntynyt tietokoneiden suoritustehon valtavasta kasvusta. Suoritusteho mahdollisti entistä suurempia ohjelmistoja, joissa oli paljon erilaisia ominaisuuksia. Ohjelmistoprojekteille tämä tarkoitti koko ajan kasvavaa määrää tarkastettavia asioita. Oltiin tilanteessa, jossa yksittäinen ohjelmoija ei enää välttämättä pystynyt hallitsemaan tai edes muistamaan koko ohjelmiston kaikkia erilaisia yksityiskohtia. Luonnollisesti ohjelmistojen paisumisen myötä myös itse ohjelmistoprojektit kasvoivat suuremmiksi ja entistä vaikeammin hallittaviksi kokonaisuuksiksi. Tästä seurasi nykypäivänäkin ohjelmistotuotannolle tutut ongelmat budjetin ja määräaikojen ylittämisestä ja seurauksena oli huo-

nosti optimoituja ja heikkolaatuisia ohjelmistoja, jotka eivät täyttäneet niille asetettuja vaatimuksia. Tuolloin ohjelmistoteollisuus joutui ensimmäistä kertaa tunnustamaan nämä ongelmat ja alkamaan kehittää niihin ratkaisuja. (Kasurinen 2013)

Ohjelmistokriisin seurauksena alkoi syntyä käsitys ohjelmistotuotannosta nykyaikaisemmassa muodossa ja samalla luotiin myös pohja ohjelmistotestaukselle. Koska ohjelmistoista tuli monipuolisempia ja samalla monimutkaisempia, oli ominaisuuksien testaaminen myös otettava paremmin huomioon. Alettiin siis laatia suunnitelmia siitä, että mitä asioita testataan ja missä kehitysvaiheessa niiden testaaminen suoritetaan. Tämä johti aluksi tarkastuslistojen käyttöönottoon, joiden avulla seurattiin, että mitä asioita oli jo kehityksen edetessä testattu. Ohjelmistotestauksen kehittyessä menetelmät ja työkalut ovat kehittyneet sen mukana, mutta testaussuunnitelmat ja tarkastuslistat ovat edelleen osa testausta, samoin kuin useat ohjelmistokriisin seurauksena löydetyt ongelmat, jotka on edelleen huomioitava. (Kasurinen 2013)

Ohjelmiston testaaminen on tärkeä osa sen kehittämistä, sillä virhe saattaa aiheuttaa vakaviakin seurauksia. Ongelmien vakavuus voi vaihdella suuresti kohteena olevasta ohjelmistosta riippuen. Virhe sairaalan laitteessa saattaa aiheuttaa potilaan menehtymisen, mutta virhe videopelissä pakottaa pahimmillaankin pelaajan vain aloittamaan pelaamisen alusta. (Vesterholm & Kyppö 2010) Testauksen päätavoite on tarkastaa, että luotu ohjelmisto on toteutettu suunnitelmien mukaisesti ja se täyttää asiakkaan tarpeet. Testaaminen on siis myös jatkuvaa vertailua, jossa aikaansaatua ohjelmistoa tai sen osaa vertaillaan siihen, mitä on ollut tarkoitus tehdä. Näin voidaan tunnistaa kohdat, jotka poikkeavat tehdystä suunnitelmasta. (Kasurinen 2013)

Testaus todetaan päättyneeksi yleensä siinä vaiheessa, kun ohjelmistosta ei enää löydetä merkittäviä virheitä ja se täyttää sille asetetut muut vaatimukset. Testauksen jälkeen ohjelmiston eli halutun tuotteen pitäisi siis olla valmis otettavaksi käyttöön. Testauksen jälkeen ohjelmistoon saatetaan vielä tehdä joitain muutoksia esimerkiksi asiakasympäristöön asentamista varten, mutta pääasiassa ohjelmisto on valmis käyttöönottoon ja siirtyy asiakkaalle ja ylläpitoon. (Kasurinen 2013) Vaikka testauksessa ei enää löydettäisi virheitä ei se kuitenkaan tarkoita, etteikö ohjelmistossa niitä yhä olisi.

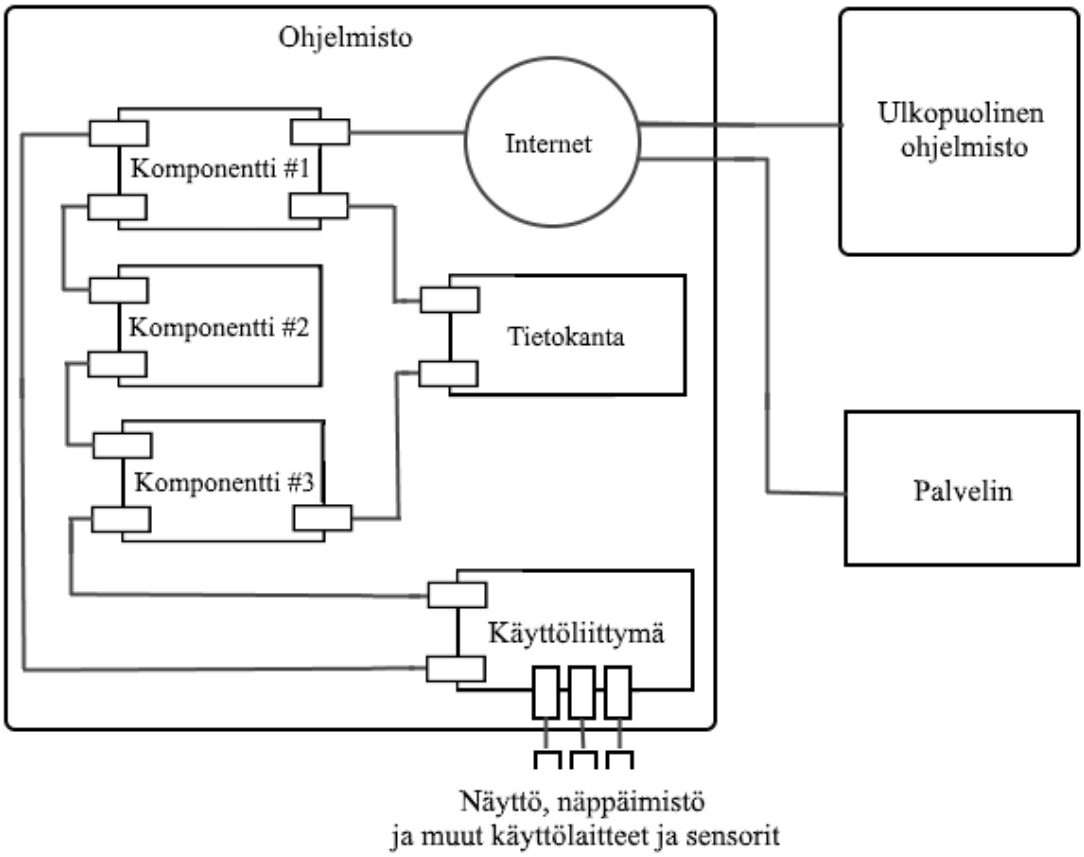
Testauksella on mahdollista osoittaa vain, että ohjelmistossa on virheitä, sillä täysin virheettömäksi ei voida todistaa edes pientä ohjelmaa. Käytännössä testauksella pystytään kattamaan vain pieni osa kaikista mahdollisista ohjelmiston toiminnoista. Hyvin testatuissa ohjelmistoissa kyse on siitä, että mahdolliset virheet ovat niin pieniä tai harvinaisia, että ne eivät enää haittaa ohjelmiston käyttöä. (Haikala & Mikkonen 2011)

Testauksesta ja sen hallinnoinnista on syytä huomata, että ohjelmistotestaukseen liittyy paljon muutakin, kuin ohjelmiston kokeilemista ja löytyvien virheiden korjaamista. Kokonaisuuteen liittyy myös organisaation määrittelemää politiikkaa, erilaisia työtehtäviä, työvälineitä, lakien asettamia vaatimuksia, standardeja ja jopa erilaisia kulttuureja. (Kasurinen 2013)

Tämän diplomityön toisessa luvussa käsitellään ohjelmistotuotantoa kokonaisuudessaan ja sen menetelmiä. Erilaisia kehitysmenetelmiä läpikäydessä on hyvä kiinnittää huomiota siihen, miten testaus näkyy kunkin menetelmän kohdalla. Kolmas luku toimii pohjustuksena testaukselle yleisesti. Neljäs luku siirtyy varsinaiseen ohjelmistotestaukseen ja viides luku esittelee sen erilaisia menetelmiä. Diplomityön kuudes luku käsittelee ohjelmistotestauksen suunnittelua ja esittelee samalla ohjelmistotestauksessa käytettäviä työvälineitä, sillä niiden tunteminen on tärkeä osa suunnittelua tehtäessä. Lopuksi suoritetaan aiheesta yhteenveto ja pohditaan mahdollisia aiheita jatkotutkimukselle.

2 OHJELMISTOTUOTANTO

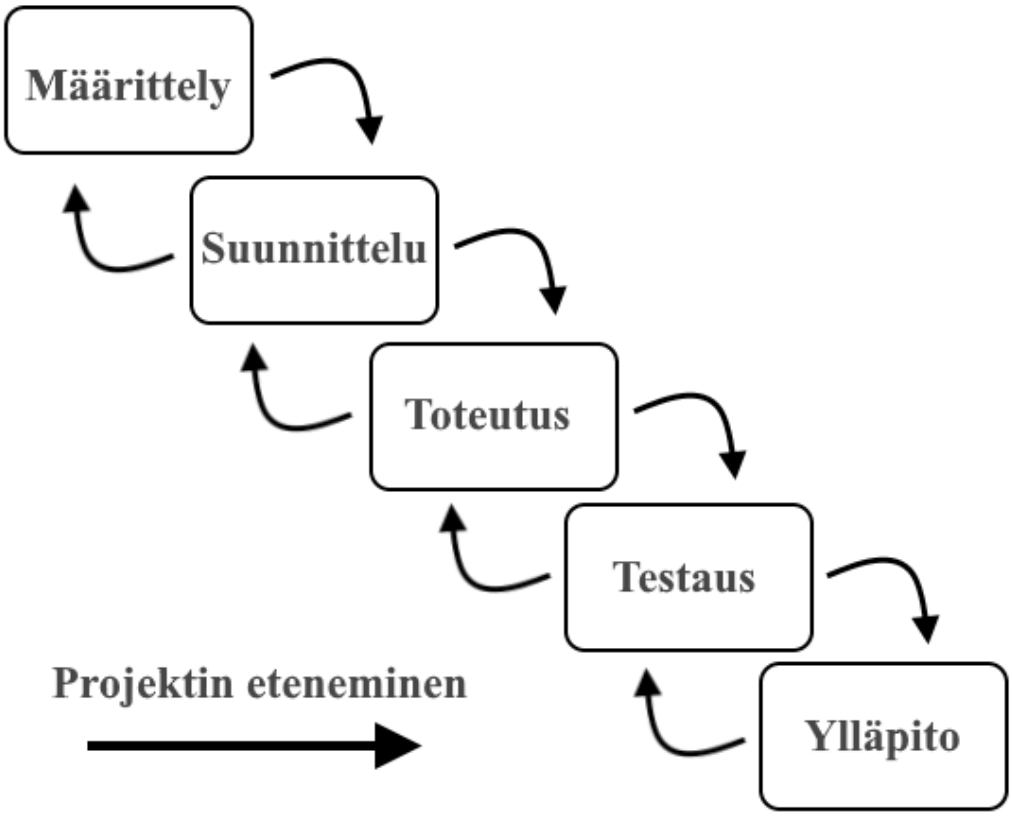
Ohjelmistotuotanto tarkoittaa tietokoneohjelmistojen rakentamisessa käytettyjä tekniikoita, työvälineitä, menetelmiä ja periaatteita. ”Ohjelmistotuotannon tarkoitukseksi voidaan ajatella asiakkaan kohtuulliset odotukset täyttävien tietokoneohjelmien tuottaminen siten, että kehityskustannukset ja aikataulu ovat ennustettavissa riittävällä tarkkuudella.” (Haikala & Mikkonen 2011, s. 12) Tämä tarkoittaa, että lähes kaikki ohjelmiston tekemiseksi suoritettavat vaiheet ovat osa ohjelmistotuotantoa asiakkaan tarpeiden määrittelystä ohjelmiston testaukseen. (Haikala & Mikkonen 2011) Kuvassa 1 on esitetty yksinkertainen esimerkki tavallisesta ohjelmiston rakenteesta.



Kuva 1: Esimerkki tavallisesta ohjelmiston rakenteesta (Kasurinen 2013)

2.1 Vesiputousmalli

Yksi yleinen mielikuva testauksesta on sen oleminen yhtenä työvaiheena perinteisessä projektityön vesiputousmallissa, kuten nähdään kuvassa 2. Kyseisessä mallissa projekti etenee vaiheittain siten, että ohjelmiston tekeminen aloitetaan vaatimusten määrittelyllä, jossa kootaan saatavilla olevat tiedot, kuten tiedossa olevat asiakastarpeet ja kannattavuuslaskelmat. Kun taustatutkimus on suoritettu, kootaan aineiston perusteella lista vaatimuksia, jotka ohjelman on täytettävä. Näitä vaatimuksia hyödyntäen muodostetaan suunnitelma ohjelmiston rakenteesta, toiminnallisista vaatimuksista ja itse projektin läpiviennistä. Tämän jälkeen toteutetaan ohjelmisto suunnitelmaa noudattaen, jonka jälkeen se siirtyy testauksen kautta käyttöön ja ylläpitoon. Vesiputousmallissa ajatuksena on se, että toiminta etenee aina askeleesta seuraavaan. Mikäli ongelmia ilmenee, voidaan palata edelliseen askeleeseen, mutta tavoitteena on, että näin ei jouduttaisi tekemään. (Kasurinen 2013)



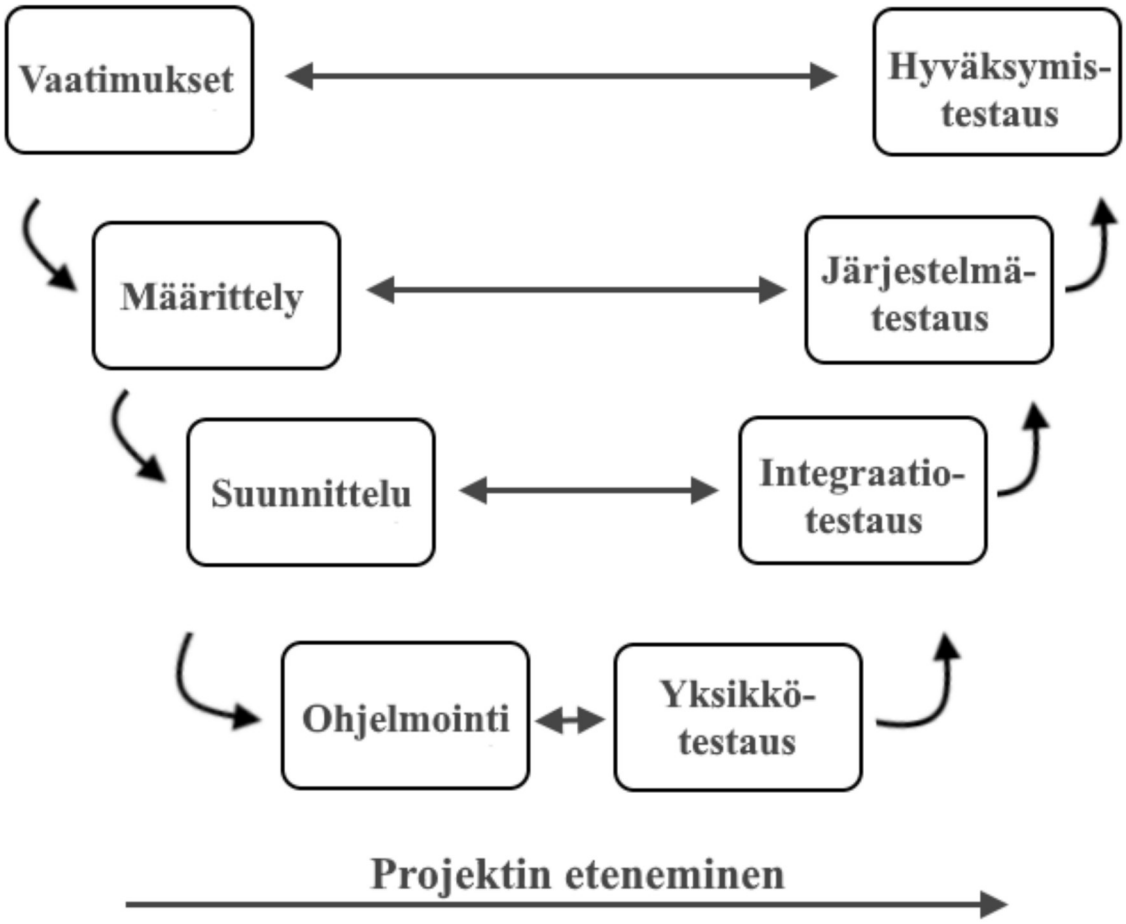
Kuva 2: Projektin eteneminen vesiputousmallissa (Kasurinen 2013)

Jo alkuperäisessä vesiputousmallissa oli mukana iterointi eli työvaiheiden toistaminen tarvittaessa ja sitä pidettiin tärkeänä ominaisuutena. (Royce 1970) Teollisuudessa malli kuitenkin usein yksinkertaistuu malliksi, josta iterointi puuttuu ja tämä on saanut mallin vaikuttamaan vanhanaikaiselta ja toimimattomalta. Uudempia projektimalleja tarkastel-

lessa voidaan usein kuitenkin huomata, että myös osa niistä koostuu sarjasta toisiaan seuraavia lyhyitä vesiputouksia. Ohjelmiston kehitystyön ja testauksen suhdetta havainnollistetaan usein V-mallilla, jossa testauksen suunnittelu tapahtuu testaustasoa vastaavalla suunnittelutasolla. Vastaavasti tulokset todetaan oikeiksi vertaamalla niitä vastaavan tason dokumentteihin. (Haikala & Mikkonen 2011)

2.2 V-malli

V-mallissa toteutus toimii samankaltaisesti, kuin vesiputousmallissa, mutta testaus kulkee mukana jokaisessa vaiheessa sen sijaan, että se olisi yksi erillinen vaiheensa. V-mallissa jokaiselle vaiheelle on merkitty omat testauskategoriansa, jotka näkyvät kuvassa 3. Tämä tarkoittaa sitä, että määrittelyn paikkansapitävyys tarkastetaan järjestelmätestauksella, suunnitelmien toteutuminen integrointitestauksella ja ohjelmointityö pääsääntöisesti yksikkötestauksella. Lisäksi järjestelmän yleiset vaatimukset tarkastetaan erillisellä hyväksymistestauksella. Jos ohjelma läpäisee kaikki testauksen tasot, se on valmis otettavaksi käyttöön ja siirtyy ylläpitoon. (Kasurinen 2013)



Kuva 3: Projektin eteneminen V-mallissa (Kasurinen 2013)

V-malli siis määrittelee ohjelmistotuotannon testaukselle kolme päävaihetta, jotka ovat yksikkötestaus, integrointitestaus ja järjestelmätestaus. Näitä kutsutaan testaustasoiksi, koska testausta tehdään V-mallin eri tasoilla. Yksikkötestauksessa testitapaus kohdistuu yhteen ohjelmiston osaan. Integrointitestauksessa kohteena on muutama osa ja niiden välinen toiminta. Järjestelmätestauksessa testaus kattaa jo koko ohjelmiston, mutta testaustyö yleensä suoritetaan vielä erillisessä testiympäristössä. Varsinaisen ohjelmistokehityksen jälkeen tulee vielä hyväksymistestaus, jossa testaus tehdään varsinaisessa kohdeympäristössä tai sen mahdollisimman tarkassa simulaatiossa. (Kasurinen 2013)

2.3 Iteratiivisuus

Ohjelmistotuotannon käytännöt teos määrittelee perinteisten tuotekehitysmallien, kuten vesiputousmallin, ongelmiksi erityisesti kuusi tekijää, jotka ovat seuraavat. (Haikala & Mikkonen 2011)

1. Oppimisen tehokas hyödyntäminen
2. Riskien hallinnan monimutkaisuus ja riskien ennalta-arvaamattomuus
3. Testauksen myöhäinen aloitus
4. Vaatimusten tarkentuminen ja myöhäinen asiakaspalaute
5. Itsepetos
6. Kärsimättömät asiakkaat

Oppimisen tehokas hyödyntäminen on haasteellista, koska kehityksen syklit ovat niin pitkiä, että kun päästäisiin hyödyntämään uudelleen aikaisemmin opittua, niin asiat ovat saattaneet jo unohtua ja mahdollisesti myös osa työntekijöistäkin jo vaihtunut.

Riskien hallinnan monimutkaisuudella ja riskien ennalta-arvaamattomuudella tarkoitetaan sitä, että perinteisissä kehitysmenetelmissä osa riskeistä paljastuu vasta testausvaiheessa, joten niihin saattaa olla myöhäistä tai ainakin hyvin kallista reagoida.

Testauksen myöhäinen aloitus todettiin ongelmaksi jo edeltävässä kohdassa, mutta lisäksi tästä aiheutuu se, että testaajat eivät pysty valmistautumaan testaukseen ennen varsinaista testausvaihetta. Testauksella on myös varattu esimerkiksi vesiputousmallissa paljon vähemmän aikaa, kuin sitä yleensä käytännössä tarvitaan, sillä virheiden korjaamiseen ja ohjelmiston uudelleen testaamiseen kuluu väistämättä aikaa.

Vaatimusten tarkentuminen ja myöhäinen asiakaspalaute muodostavat myös ongelman, sillä kehityssyklin ollessa hyvin pitkä ehtivät ohjelmiston vaatimukset muuttua ja tähän voidaan pahimmillaan reagoida vasta, kun on jo tuotettu valmiiksi vanhentunut ohjelmisto. Pahimmillaan tämä seikka saattaa vielä selvitä vasta valmiista ohjelmistosta saadusta asiakaspalautteesta.

Itsepetos kuvaa projektin tilannetta, jossa aikataulun on helppo julistaa pitävän, koska suunnitelmaan merkityt valmistumispäivämäärät ovat pitävät, mutta kuten aikaisemmissa kohdissa on todettu, niin on todennäköistä, että testausvaiheessa paljastuvat viat osoittavat, että projektin osat eivät itse asiassa ole kunnolla

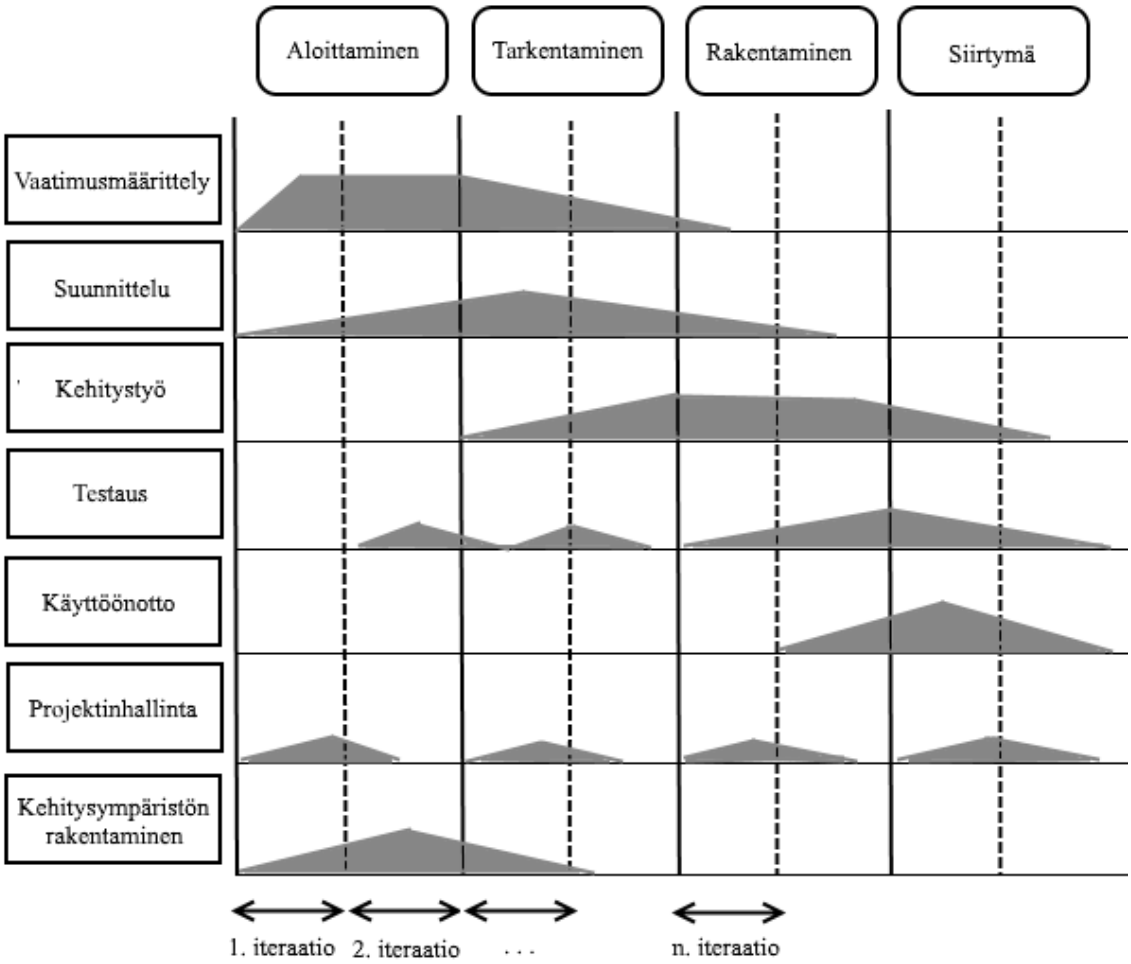
valmistuneet. Jotain on siis saatu valmiiksi aikataulussa, mutta sen laatu voi olla aivan toinen asia.

Kärsimättömät asiakkaat ovat ongelma, johon on helppo samaistua. Kehitysajan ollessa pitkä kestää myös kauan ennen kuin asiakas näkee mitään tuloksia. Samalla tapaa myös projektin työntekijöiden usko omaan tuotteeseen voi olla koetuksella, kun tiedostetaan, että varmuutta onnistumisesta ei saada pitkään aikaan.

Nämä ongelmat ilmenevät yksinkertaisessakin kehityssyklissä, joten ratkaisuksi niille on kehitetty iteratiivinen kehitysmenetelmä, jossa ydinajatuksena on lyhentää kehityssyklin pituutta. Iteratiivisessa kehityksessä sykli voi olla esimerkiksi kuukausi, kun vesiputousmallissa syklin pituus voi olla vuosikin. Vaatimukset jaetaan siis näihin lyhyisiin iteraatioihin ja jokaisen iteraation jälkeen pystytään periaatteessa esittelemään asiakkaille jotain tuloksia. Iteratiivisella ohjelmistokehityksellä on tietysti myös heikkoutensa ja testauksen näkökulmasta se on hieman jopa ironinen, sillä ongelmia aiheuttaa säännöllinen testaus. Testauksen määrä kasvaa jokaisen lisätyn syklin myötä, joten siihen käytetään myös koko ajan paljon aikaa. Tämä edellyttää testauksen automatisointia mahdollisimman tehokkaasti. Ongelmia voi seurata myös vaatimusten jatkuvasta lisääntymisestä, jolloin projektilla ja erityisesti sen dokumentaatiolla voi olla vaikeuksia pysyä perässä. (Haikala & Mikkonen 2011)

2.4 RUP

RUP-mallissa on pyritty hyödyntämään ohjelmistotuotannossa hyväksi havaittuja tapoja, kuten UML-kaavioiden avulla toteutettua visuaalista suunnittelua, vaatimustenhallintaa ja laadunvalvontaa, joka toteutetaan testaamalla ohjelmistoa läpi koko projektin. (Kasurinen 2013) RUP on hyvin laaja kokonaisuus, joka muistuttaa hieman vesiputousmallia, mutta kaikkia vesiputouksen vaiheita tehdään eri painotuksin kaikissa RUP-mallin neljässä eri vaiheessa. Kuvasta 4 nähdään, että nämä vaiheet ovat aloittaminen, tarkentaminen, rakentaminen ja siirtymä. Kussakin vaiheessa taas käydään läpi sitä koskien sen vaatimusmäärittely, suunnittelu, varsinainen kehitystyö ja testaus. Aloittamisessa määritellään ohjelmiston ominaisuudet ja tehdään alustavat suunnitelmat ja arvioidaan riskit. Näin saadaan myös arvio siitä, onko projekti mahdollinen. Tarkentamisessa täydennetään suunnitelmia ja luodaan kuvaus ohjelmiston arkkitehtuurista ja tehdään myös alustava käyttöohje. Rakentamisvaiheessa edellisten vaiheiden suunnitelmia täydennetään ja suoritetaan varsinainen ohjelmiston toteutus. Siirtymässä ohjelmisto voidaan ottaa käyttöön ja sen kuvaukset ja käyttöohjeet saadaan täydennettyä lopullisiksi. Lisäksi mallissa resurssoidaan aikaa kehitysympäristön rakentamiseen, projektinhallintaan ja valmiin ohjelmiston käyttöönottoon. (Haikala & Mikkonen 2011)



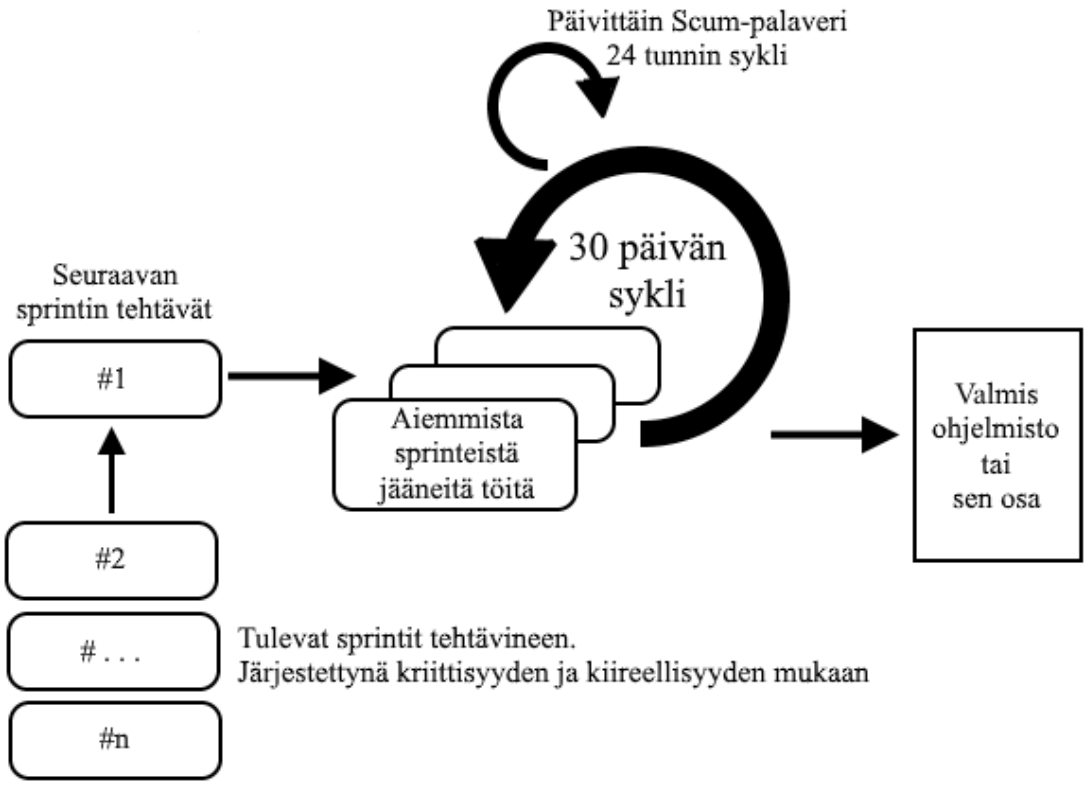
Kuva 4: Projektin eteneminen RUP-mallissa (Kasurinen 2013)

2.5 Ketterät menetelmät

Raskaat kehitysmenetelmät, joissa paljon painoarvoa on hallinnossa ja byrokratiassa, ovat luonnollisena seurauksena synnyttäneet vastapainokseen kevyempiä menetelmiä. 2000-luvun alussa luotiin pohja niin kutsutuille ketterille menetelmille ja niitä edistämään perustettiin Agile Alliance -järjestö. Järjestö on julkaissut manifestin ketterien menetelmien arvoista, joita voidaan tulkita niin, että prosessi, työvälineet, dokumentaatio ja niin edelleen ovat tarpeettomia. (Agile Alliance 2001) Vaikka manifesti tunnustaakin nämä arvokkaiksi ohjelmistokehityksen kannalta, se toteaa, että tärkeintä on tyytyväinen asiakas ja toimiva ohjelmisto. (Haikala & Mikkonen 2011) Suunnittelulähtöiset menetelmät kuten vesiputousmalli ja RUP pyrkivät varautumaan kaikkiin ongelmiin, kun taas ketterien menetelmien tarkoituksena on pystyä reagoimaan mahdollisimman nopeasti ilmaantuviin ongelmiin. Rajanveto näiden raskaiden ja ketterien kehitysmenetelmien välillä ei kuitenkaan ole aina täysin selvää. Esimerkiksi RUP sisältää useita ketterien me-

netelmien ominaisuuksia, vaikka sitä ei luokitella ketteräksi. Yleensä ketteriksi luokitellaan ne menetelmät, joissa korostuu kommunikaatio, yhteistyö asiakkaan kanssa, sekä ohjelmiston toiminnallisuudet enemmän kuin kattava etukäteen suunnittelu tai sopimuksien perusteella ohjelmiston määritteleminen. (Kasurinen 2013)

Yksi suosittu ketterä menetelmä on nimeltään Scrum, jonka merkittävin etu on sen yksinkertaisuus. Scrum:in perusperiaatteet on helppo käydä läpi nopeasti. Sen yksinkertaisuus on kuitenkin osittain harhaa, sillä vaikutelma selittyy osin sillä, että Scrum ei ota kantaa kaikkiin ohjelmiston elinkaareen liittyviin tehtäviin. Se ei myöskään käsittele käytettäviä kehitysmenetelmiä tai työvälineitä. Projektinhallintamenetelmän sijasta Scrum on siis pikemminkin tapa organisoida ohjelmistoprojektin iteraatiot. Scrum ei siis yksinään riitä koko ohjelmistoprojektin läpiviemiseksi, vaan vaatii avukseen, myös muita menetelmiä. (Haikala & Mikkonen 2011) Scrum on parhaimmillaan organisaatioissa, joissa tehdään tiivistä yhteistyötä ja joissa kommunikaatio on tehokasta asiakkaalle ja organisaation sisällä. Kommunikaatio korostuu erityisesti Scrum-palavereissa, joilla olisi tarkoitus aloittaa teoriassa jokainen projektin työpäivä. Lyhyissä palavereissa, on tarkoituksena keskustella vastaan tulleista ongelmista ja jakaa seuraavat tehtävät kullekin projektin jäsenelle. (Kasurinen 2013)



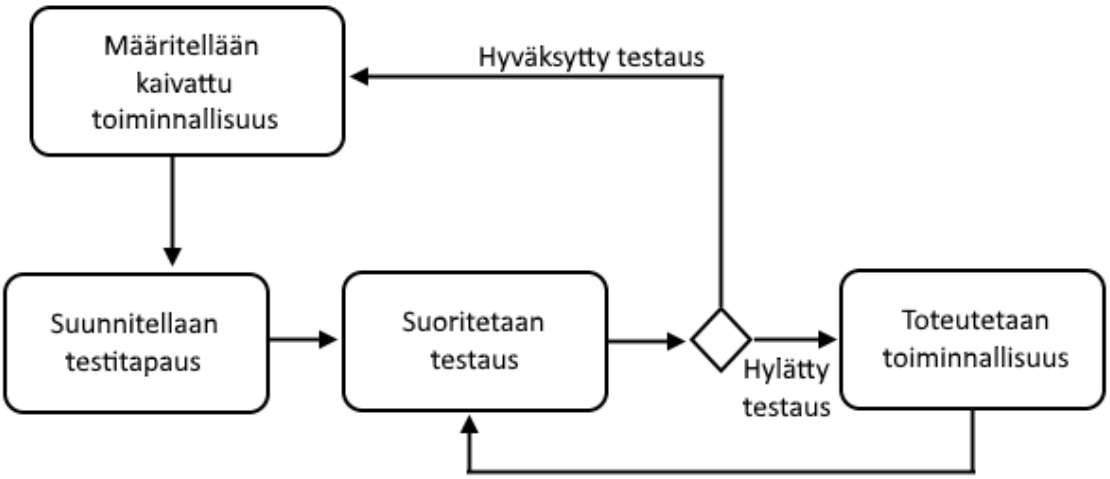
Kuva 5: Projektin osien eteneminen Scrum-menetelmässä (Kasurinen 2013)

Ohjelmiston määrittely suoritetaan kulloinkin parhaiten sopivaa menetelmää, joka voi olla esimerkiksi markkinatutkimus tai perinteisesti asiakasta haastattelemalla. Määrittelyn ja suunnittelun jälkeen ohjelmiston toteuttaminen jaetaan lyhyisiin iteratiivisiin osiin, kuten kuvassa 5. Näitä osia kutsutaan sprinteiksi. Scrum -menetelmässä ohjelmistoa kehitetään näissä lyhyissä jaksoissa, jotka voivat olla esimerkiksi muutaman päivän tai vaikka viikon mittaisia. Yhden jakson aikana on tarkoitus luoda toteuttaa tietty määrä ominaisuuksia ja varmistaa testauksella, että ne toimivat. Kun kaikki jaksoihin jaetut tehtävät on suoritettu, voidaan valmistunut ohjelmisto tai sen osa toimittaa eteenpäin. (Kasurinen 2013)

Usein ketterät ohjelmistokehitysmenetelmät määrittelevät, että laatu rakennetaan mukaan ohjelmistoon kehitystyön aikana, eikä sitä jätetä testauksen mukana viimeiseksi työvaiheeksi. Asiakkaan tarpeiden asettamien laatuvaatimusten toteuttaminen ei voi onnistua kunnolla liimaamalla niitä ohjelmiston päälle jälkikäteen. Ohjelmiston ja sen testauksen laadun tulisi lähteä jo itse kehitystyöstä, eikä sitä sovi jättää vain testaajien tai vielä pahempaa asiakkaan vastuulle. Mitä enemmän ja mitä aikaisemmassa vaiheessa testausta tehdään, niin sitä vähemmän ongelmia on projektin myöhäisemmissä vaiheissa. Kehitystiimin ja sen jäsenten on tärkeää alusta saakka sisäistää oman koodinsa testaamisen tärkeys. Ohjelmiston testaus on hyvä aloittaa heti projektin alusta alkaen myös siksi, että testien koodin kirjoittaminen ja ylläpito muodostuu paremmin tavaksi. (Kuha 2008)

2.6 Testivetoinen kehitys

On olemassa myös ohjelmistokehitysmalli, jossa testaus määrää projektin etenemisestä. Tällöin on kyseessä testivetoinen kehitys eli englanniksi Test Driven Development. Tässä mallissa projektin läpivienti on suunniteltu niin, että se hyödyntää mahdollisimman lyhyitä iteraatioita ja korostaa, että kerrallaan tehdään vain pieniä lisäyksiä ohjelmistoon. Kun ohjelmiston määrittely ja suunnittelu on tehty, niin luodaan ensiksi testitapaus, jonka perusteella aletaan vasta luoda itse komponenttia, jota se koskee. Kun testitapaus on valmis, tehdään ohjelmistoa niin pitkälle, että se kykenee suorittamaan kyseisen testitapauksen. Tämän jälkeen lähdekoodia voidaan vielä parantaa, jotta se täyttää paremmin projektin muut vaatimukset. Tämän jälkeen voidaan taas edetä projektissa suunnittelemalla uusi testitapaus. (Kasurinen 2013) Menetelmän etenemistä havainnollistetaan kuvassa 6. Testivetoinen kehitys luotiin aluksi ketteriä menetelmiä ajatellen, mutta sitä on mahdollista hyödyntää myös muiden ohjelmistokehitysmenetelmien kanssa. (Sommerville 2011)



Kuva 6: Projektin eteneminen testivetoisessa kehityksessä (Sommerville 2011)

Testivetoisessa kehityksessä toteutettavien muutosten määrä pidetään niin pienenä, että pystytään aina helposti todistamaan järjestelmän toimivan yhä oikein. Malli siis luottaa siihen, että ohjelmistoa varten ei tarvitse luoda kattavaa suunnitelmaan, vaan sitä voidaan kasvattaa pala palalta. Koska järjestelmä on suunnitelmansa osalta jatkuvasti ajan tasalla, ei projektissa tarvitse yhdistää eri kehitysversioita tai suorittaa erillistä regressiotestausta. Teknisen toteutuksen suhteen testivetoinen kehitys kannustaa organisaatiota kehittämään komponentteja, joita on helppo käyttää uudelleen useissa projekteissa. (Kasurinen 2013)

Testivetoisessa kehityksessä on luonnollisesti myös heikkoutensa. Se luottaa kaikista eniten yksikkötason testaukseen, joten järjestelmätasolla voi kuitenkin tulla ongelmia, vaikka ohjelmistoon lisätään uusia komponentteja vain vähän kerrallaan. Lisäksi, jos testitapaus on suunniteltu huonosti, niin se saattaa antaa virheellisen käsityksen ohjelmiston laadusta, joka saattaa paljastua vasta liian myöhään. (Kasurinen 2013)

3 TESTAAMINEN YLEISESTI

Testaus on välttämätöntä toimivan ohjelmiston luomiseksi, mutta se on myös tärkeä työvaihe ohjelmistoprojektin kannattavuuden kannalta. Huolellinen testaaminen tuo yritykselle suhteessa paremman katteen, kuin jos testaus suoritettaisiin heikosti. Ero korostuu lisää, jos yritys saa huonon maineen toimimattomien ohjelmistojen takia. ”Virheen korjaaminen ohjelmaa suunnitellessa maksaa 1-2 prosenttia siitä, mitä se maksaisi julkaisun jälkeen tehtynä. Tämän vuoksi testaamiseen panostaminen ohjelman kehityksen aikana on tärkeää ja jopa suorastaan taloudellisesti kannattavaa.” (Kasurinen 2013, s. 12) Virheiden etsiminen koodista on tyypillinen ajatus ohjelmistojen testaamisesta, mutta on myös syytä tietää mitä ja miten tätä etsintää tehdään.

3.1 Staattinen ja dynaaminen testaus

Staattinen testaaminen on ohjelmiston testaamista ilman, että ohjelmistoa varsinaisesti käytetään. Tällä tarkoitetaan esimerkiksi ohjelmiston lähdekoodin lukemista manuaalisesti tai erilaisilla analysaattoreilla. Tällä pyritään tekemään perustavanlaatuista tarkastusta ennen varsinaista testausta. Staattinen testaus voidaan aloittaa jo hyvin aikaisessa vaiheessa, ennen kuin ainuttakaan komponenttia on varsinaisesti valmiina. Staattisen testauksen oleellinen ominaisuus on sen edullisuus. Aikaisessa vaiheessa löydetyt viat ovat paljon halvempia korjata. (Kasurinen 2013)

Dynaaminen testaus on staattisen testauksen vastakohta. Dynaamisessa testauksessa ohjelmistoa käytetään testausta suoritettaessa. Useimmat testausmenetelmät kuten yksikkötestaus ja integrointitestaus kuuluvat dynaamisiin menetelmiin. Staattisten ja dynaamisten testausmenetelmien ero on siis varsin selvä. (Kasurinen 2013)

3.2 Etsittävät viat

Ohjelmistoista löytyvät erilaiset vikatilanteet jaetaan International Software Testing Qualifications Board -organisaation mukaan seuraaviin ryhmiin. (ISTQB 2011)

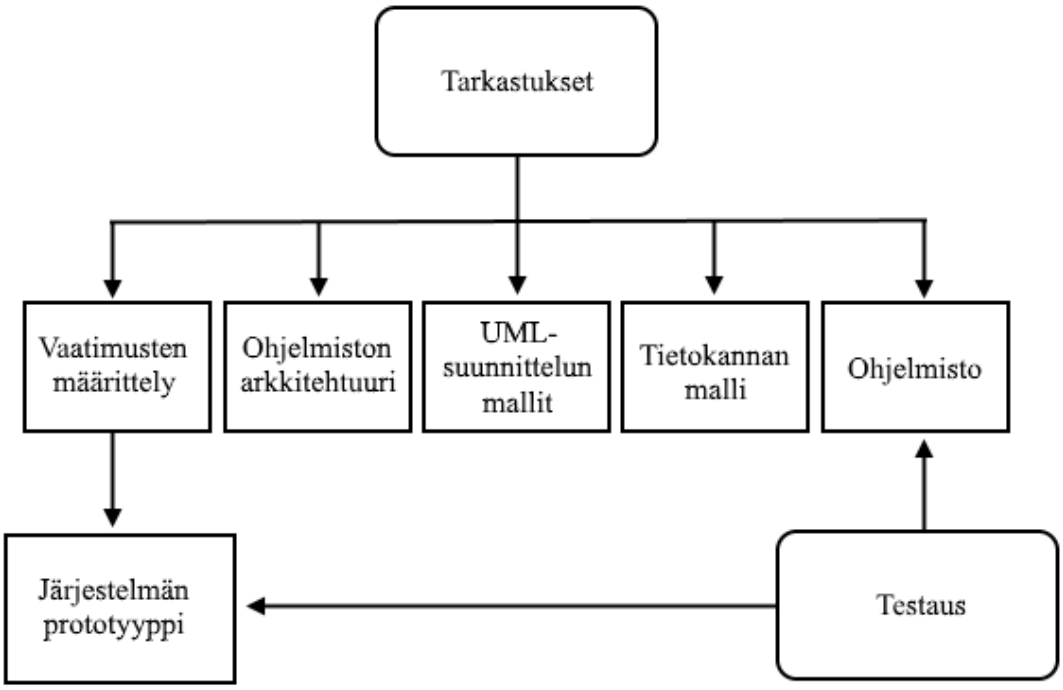
Virhe tai erehdys on ohjelmiston käyttäjän aikaansaama poikkeus, joka aiheuttaa vian ohjelmistoon, lähdekoodiin tai dokumentaatioon. Tästä virheestä voi seurata vikatilanne, joka estää ohjelmistoa toimimasta oikealla tavalla. Englannin kielessä tästä käytetään termiä ”mistake”.

Vika on ohjelmistossa tai dokumentaatiossa oleva poikkeus, joka saattaa estää ohjelmistoa toimimasta oikein ja aiheuttaa näin häiriön. Puhekielessä viasta käytetään usein nimitystä ”bugi” ja englannin kielessä käytetään termiä ”error”.

Häiriö on tilanne, jossa ohjelmisto toimii toisin, kuin on tarkoitettu. Ohjelmistossa olevien vikojen takia häiriöitä voivat aiheuttaa myös muut järjestelmän osat kuten käytetty laitteisto, vaaralliset mittarit, magneettikenttä, sähkökenttä, huonepöly tai mikä tahansa muu asia, joka haittaa järjestelmän toimintaa. Englanniksi käytetään termiä ”failure”.

3.3 Tarkastukset

Tarkastukset ovat staattisia menetelmiä, jotka analysoivat ja tutkivat järjestelmän vaatimukset, suunnittelumallin, ohjelmiston lähdekoodin ja suunnitellut testit. Kuvassa 7 olevista nuolista nähdään, miten tarkastukset liittyvät ohjelmistokehityksen vaiheisiin. Tarkastukset keskittyvät usein lähdekoodiin, mutta periaatteessa mikä tahansa ohjelmistoprojektin osa, jota pystytään lukemaan voi olla kohteena. Tarkastusta tekevällä henkilöllä on oltava riittävä tietämys tarkastettavasta järjestelmästä. Hänen on siis lähtökohtaisesti vähintään ymmärrettävä ohjelmointikieltä, jolla lähdekoodia on kirjoitettu. (Sommerville 2011)



Kuva 7: Tarkastukset ja testaus osana ohjelmistokehitystä (Sommerville 2011)

Tarkastuksella on tiettyjä etuja testaukseen verrattuna. Virheet voivat jäädä huomaamatta testauksessa tilanteissa, joissa ne peittyvät toisten virheiden alle. On vaikeaa tietää johtuvatko odottamattomat tulokset ohjelmistossa uudesta virheestä vai jostain aikaisemmasta virheestä, joka on tullut esiin vasta toisen virheen korjaamisen jälkeen. Tarkastuksessa useiden virheiden välisellä vuorovaikutuksella ei ole kuitenkaan merkitystä, koska lähdekoodia tarkastellaan rivi riviltä. Tarkastus voi siis löytää kerralla myös useampia virheitä, kuin testaus. Keskeneräistä järjestelmää on mahdollista tarkastaa ilman minkäänlaisten apuvälineiden koodaamista, toisin kuin testauksessa. Tarkastus on siis tältä osin edullisempaa. Tarkastuksessa pystytään myös huomioimaan myös sellaisia ohjelmiston ominaisuuksia, joita testaukseen ei voida sisällyttää. Tällaisia ovat esimerkiksi ylläpidettävyys, siirrettävyys ja standardien noudattaminen. (Sommerville 2011) Kun puhutaan ulkopuolisten ammattilaisten tai ohjelmiston käyttäjien suorittamista tarkastuksista ohjelmiston määrittelyyn, kuvaukseen tai toteutukseen, niin tällöin käytetään myös termiä läpikäynti. (Pohjonen 2002)

Ohjelmiston tarkastaminen on hyvin vanha menetelmä ja siitä on tehty paljon tutkimuksia vuosien saatossa, jotka kertovat sen olevan testausta tehokkaampi löytämään isomman määrän virheitä. Totuus kuitenkin on, että tarkastus ei käytännössä pysty korvaamaan ohjelmistotestausta. Tarkastukset eivät kuitenkaan ole kovin kykeneviä löytämään virheitä, jotka johtuvat ohjelmiston komponenttien odottamattomista vuorovaikutuksista tai ongelmista ajoituksen tai järjestelmän suorituskyvyn kanssa. Käytännössä tarkastus on myös niin paljon työläämpää ja hitaampaa, että se vaatii suuren joukon työvoimaa. (Sommerville 2011)

3.4 Validointi ja verifiointi

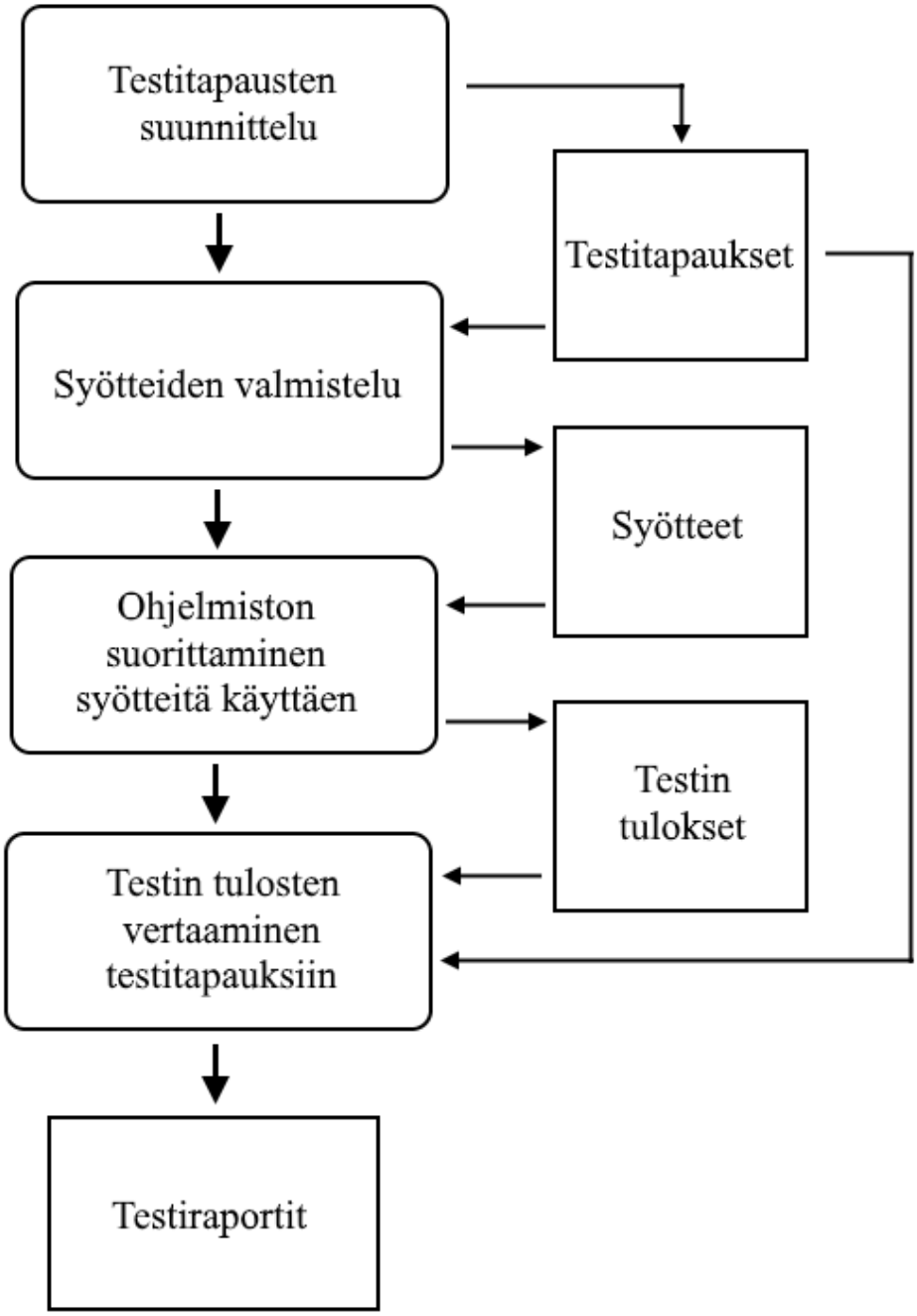
Ian Sommerville esittelee validoinnin ja verifioinnin eron kirjassaan Software Engineering näin. (Sommerville 2011)

> Validointi kysyy, teemmekö oikeaa tuotetta.
>
> Verifiointi kysyy, teemmekö tuotetta oikein

Validointi ja verifiointi pyrkivät siis siihen, että ohjelmistoa kehitetään oikein. Nämä prosessit alkavat heti ensimmäisten määrittelyjen synnyttyä. Verifioinnin tarkoituksena on varmistaa, että ohjelmisto täyttää sen toiminnalliset ja ei-toiminnalliset vaatimukset. Validointi sen sijaan on yleisempi prosessissa, jonka tarkoituksena on varmistaa, että ohjelmisto täyttää asiakkaan odotukset. Sekä validointi että verifiointi pyrkivät siis varmistamaan, että ohjelmisto on tarpeeksi hyvä sille suunniteltuun tarkoitukseensa. Tähän vaikuttavia tekijöitä ovat esimerkiksi ohjelmiston kohteen kriittisyys, käyttäjien odotukset ja markkinoinnin vaatimukset. (Sommerville 2011)

3.5 Testausprosessi

Kuva 8 esittää yksinkertaistetun mallin perinteisestä ohjelmistotestauksen prosessista. Testitapauksissa määritellään testeihin menevät syötteet ja niistä odotettavat tulokset. Testien suorittaminen on myös mahdollista automatisoida. Testausohjelmisto voi verrata odotettuja tuloksia todellisuudessa saatuihin ja ilmoittaa poikkeamista.



Kuva 8: Esimerkki perinteisestä testausprosessista (Sommerville 2011)

Ohjelmistolle suoritettavan ensimmäisen testitapauksen ei tarvitse olla monimutkainen, sillä yksi sen tärkeimpiä tarkoituksia on testata myös itse testauksen toimivuutta. Ensimmäisellä testitapauksella voidaan samalla varmistua, että testauksen apuna käytetyt luokkakirjastot ja sovelluskehykset toimivat myös oikein. (Kuha 2008)

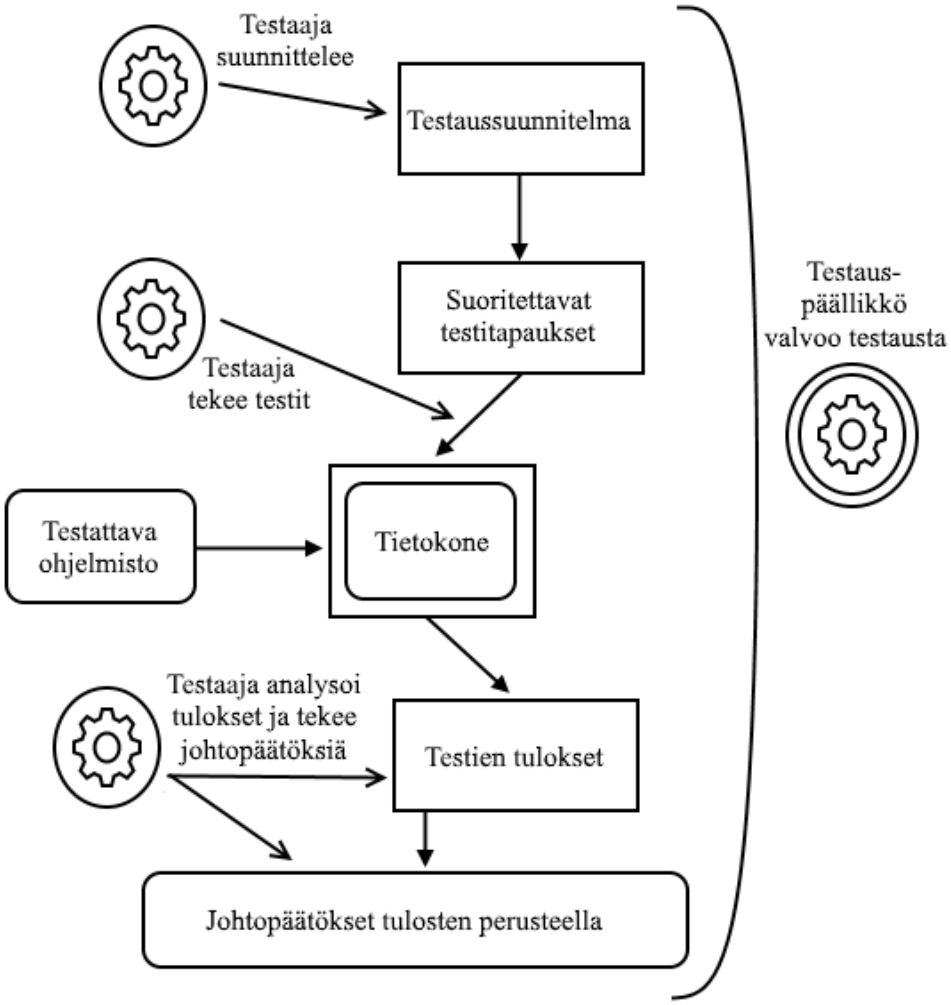
Ensimmäisiä testauskohteita voi olla esimerkiksi varmistaa, että ohjelmiston mahdollinen tietokanta toimii. Tämä kannattaa tehdä aikaisessa vaiheessa, koska myöhemmin voi kulua paljon aikaa sen määrittelyssä. Testauksessa katsotaan siis, että tietokantatason komponenttien perusasetukset ja määrittelyt ovat kunnossa. Käytännössä testauksessa yleensä tallennetaan ensin jotain tietokantaan ja tämän jälkeen luetaan kyseinen sama tieto sieltä. On helpompaa tehdä yksi osa ohjelmistosta kerrallaan, suorittaa sille testaus ja siirtyä seuraavaan osaan vasta, kun edellinen on todettu toimivaksi. (Kuha 2008)

Jo suoritettuja testauksia on myös hyvä pitää tallessa ja ylläpitää, jossa milloin tahansa ohjelmiston kehitystyön aikana ne voidaan suorittaa uudelleen ja varmistaa, että ohjelmisto toimii edelleen halutulla tavalla. Tämä tuo myös varmuutta siihen, että viimeisin muutos ohjelmiston koodiin ei luonut ongelmia muihin koodin osiin. Koko ohjelmiston tekemisen jälkeen muistissa saattaa olla satoja tai jopa tuhansia tallennettuja testitapauksia. (Kuha 2008)

3.6 Testausprosessi organisaatiossa

Testaus ei ole pelkästään ohjelmiston käyttämistä ja toimintojen tarkastamista. Testaukseen liittyy tämän lisäksi myös suunnittelu, analysointi, raportointi ja tiedonhallinta. Testauksen eri vaiheissa keskitytään eri kohteisiin ja erilaisilla menetelmillä ja lähestymistavoilla pyritään löytämään ohjelmiston viat mahdollisimman tehokkaasti ja luotettavasti. (Kasurinen 2013)

Yksi yksinkertainen esimerkki testausprosessin läpiviennistä organisaatiossa löytyy kuvasta 9. Käytännössä testauksessa on kuitenkin yleensä mukana ainakin kolme tahoa, jotka ovat testauksesta vastaava esimies eli testauspäällikkö, ohjelmistokehittäjät ja ohjelmistotestaajat. Kaksi jälkimmäistä ryhmää ovat tosin usein samoja henkilöitä.



Kuva 9: Yksinkertaistettu esimerkki testauksen läpiviennistä organisaatiossa (Ammann & Offutt 2008)

Ohjelmistoprojektin edetessä testaus etenee projektin sisällä yleensä karkeasti esitettynä seuraavasti.

Ohjelmistoprojektin aluksi testauspäällikkö, kehittäjät ja testaajat tekevät testaussuunnitelman ja luovat ensimmäiset suoritettavat testitapaukset tämän suunnitelman perusteella. Lisäksi testausryhmä jakaa tulevat tehtävät.

Seuraavaksi kehittäjät toteutettuaan järjestelmän erilaisia komponentteja suorittavat niille yksikkötestauksen.

Ohjelmistoprojektin näin edetessä kehittäjät ja testaajat suunnittelevat sitä varten lisää testitapauksia ja päivittävät laadittua testaussuunnitelmaa aina tarpeen vaatiessa.

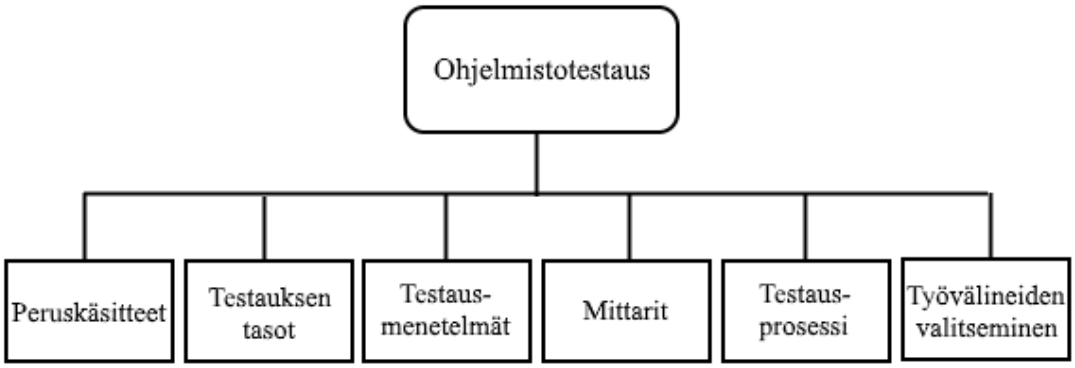
Testaajat suorittavat laadittuja testitapauksia valmiille komponenteille ja tekevät raportin epäonnistuneista testeistä. Näiden raporttien pohjalta ohjelmistokehittäjät jatkavat työskentelyä komponenttien parissa korjaten aikaisempia ja kehittäen uusia.

Komponenttien valmistumisen myötä, niitä aletaan integroida yhteen ja testaajat suorittavat näille integrointitestauksen, jälleen raportoiden löytyneistä ongelmista. Kun kaikki komponentit ovat valmiita, integroitu ja niiden yhdessä toimiminen testattu, siirrytään järjestelmätestaukseen.

Järjestelmätestauksessa suoritetaan testausta nimensä mukaisesti koko järjestelmälle eli ohjelmiston kokonaisuudelle. Kun merkittäviä ongelmia ei enää löydytä, siirrytään hyväksymistestaukseen. Hyväksymistestauksessa asiakas tai muu ohjelmiston vastaanottaja toimii hyväksyjänä ja tarkastaa, että ohjelmisto täyttää sovitut vaatimukset. Tämän jälkeen ohjelmisto voidaan todeta valmiiksi. (Kasurinen 2013)

4 OHJELMISTOTESTAUS

Ohjelmistotestaus voidaan teoksen Software Engineering Body of Knowledge mukaan jakaa kuuteen ryhmään, kuten kuvassa 10. Nämä ryhmät ovat peruskäsitteet, testauksen tasot, testausmenetelmät, mittarit, testausprosessi ja työvälineiden valitseminen. (IEEE Computer Society 2014)



Kuva 10: IEEE Computer Society jakaa ohjelmistotestauksen kuuteen osaan (IEEE Computer Society 2014)

Peruskäsitteisiin kuuluvat yksinkertaiset perusteet testauksen tekemiseksi. Niihin kuuluvat testaukseen liittyvä sanasto, testauksen avaintekijät ja testauksen suhde ohjelmistotuotannon muihin osa-alueisiin, kuten laadunvalvontaan ja sertifiointiin.

Testauksen tasot ovat testauksen kohteita ja tavoitteita. Kohteita ovat V-mallista tutut yksikkötestaus, integrointitestaus ja järjestelmätestaus. Tavoitteissa on useita erilaisia testauksia, joita ovat muun muassa hyväksymis-, kuormitus-, käytettävyys-, alfa- ja beta-testaus.

Testausmenetelmät ovat erilaisia lähestymistapoja testaukseen, jotka pyrkivät mahdollisimman järjestelmällisesti löytämään syötteet, jotka aiheuttavat vääränlaisia tuloksia. Nämä menetelmät ovat siis lähtökohtia, joiden perusteella testitapauksia lähdetään luomaan. Näitä voivat olla itse koodi, siitä löytyvät viat, ohjelmiston käyttö tai ohjelmoijan kokemus ja intuitio.

Mittarit liitetään yleensä ohjelmiston laadunvalvontaan, mutta niitä voidaan käyttää myös ohjelmistotestauksen mittaamiseen ja kehittämiseen. Mittareita voidaan hyödyntää myös testauksen suunnittelussa. Tällaisia ovat esimerkiksi ohjelmiston kokoa ja sen osien välisen kommunikaation määrää tarkkailevat mittarit.

Testausprosessissa nämä eri ohjelmistotestauksen ryhmät yhdistetään yhdeksi kokonaisuudeksi. Testausprosessi tukee ohjelmistotestausta ja ohjaa testaajien työskentelyä sen suunnittelusta aina tulosten tulkitsemiseen. Prosessin tavoitteena on mahdollistaa, että itse testauksen tavoitteet voidaan saavuttaa mahdollisimman kustannustehokkaasti.

Työvälineiden valitseminen on tärkeä vaihe ohjelmistotestausta suunniteltaessa, koska se vaikuttaa suuresti testauksen tehokkuuteen ja onnistumiseen. Työvälineiden valitseminen riippuu monesta seikasta, kuten kehitystyössä tehdyistä valinnoista ja arvioinnin tavoitteista. Yleensä yksittäinen työväline ei pysty kattamaan kaikkia tarpeita, vaan niitä on valittava useita.

4.1 Testauksen periaatteet

International Software Testing Qualifications Board -organisaatio ja sen suomalainen paikallisjärjestö Finnish Software Testing Board kiteyttävät ohjelmistotestauksen perusteet sertifikaatissaan seitsemään periaatteeseen. Nämä periaatteet ovat seuraavat. (ISTQB 2011; FiSTB 2010)

Periaate 1. Testaus osoittaa vikojen olemassaolon. Ohjelmistotestauksen tarkoituksena on vähentää vikojen todennäköisyyttä. Periaatteissakin siis lähdetään siitä oletuksesta, että täysin virheetöntä ohjelmistoa ei ole olemassa.

Periaate 2. Täydellinen testaus on mahdotonta. Ohjelmistotestauksessa pitäisi tämän vuoksi keskittyä riskien kartoittamiseen ja testauksen osien priorisointiin.

Periaate 3. Aikainen testaus. Ohjelmistotestaus on aloitettava mahdollisimman aikaisessa vaiheessa ohjelmistotuotantoa. Ohjelmistotestauksen pitää kuitenkin ottaa huomioon myös projektin vaatimukset ja suunnitelmat.

Periaate 4. Vikojen kasaantuminen. Ohjelmistotestauksessa pitäisi painottaa niitä komponentteja, joissa todennäköisimmin on virheitä. Nämä komponentit pitäisi tunnistaa ajoissa ja keskittää niihin erityisesti huomiota testauksen aikana.

Periaate 5. Hyönteismyrkkyparadoksi. Ohjelmistotestaukseen käytettyjä erilaisia testitapauksia tulee päivittää ja kehittää projektin edetessä. Mikäli näin ei tehdä, päädytään tilanteessa, jossa vain näiden alkuperäisten tapausten huomioimat virheet löydetään ja muut ongelmat jäävät kokonaan huomiotta.

Periaate 6. Testaus on tilanneriippuvaista. Ohjelmistotestausta tehdään eri tavoilla tapauksista ja projekteista riippuen. Sen takia prosessin on oltava riittävän joustava, jotta kaikki vaihtoehdot saadaan toteutettua ilman, että prosessia muutetaan jokaista erilaista vaihetta varten.

Periaate 7. Virheettömyyden harhaluulo. Ohjelmistotestauksen vaiheessa löydetyt ja korjatut viat eivät pelasta järjestelmää, joka on käyttökelvoton. Mikään laadunvalvonnan tai testauksen määrä ei ole riittävä, jos järjestelmä on suunniteltu väärin ja se ei toteuta sille projektin alussa asetettuja vaatimuksia.

4.2 Testauksen tasot

Kehitystyön aikana ohjelmistoa voidaan testata kolmella eri tasolla. (Sommerville 2011)

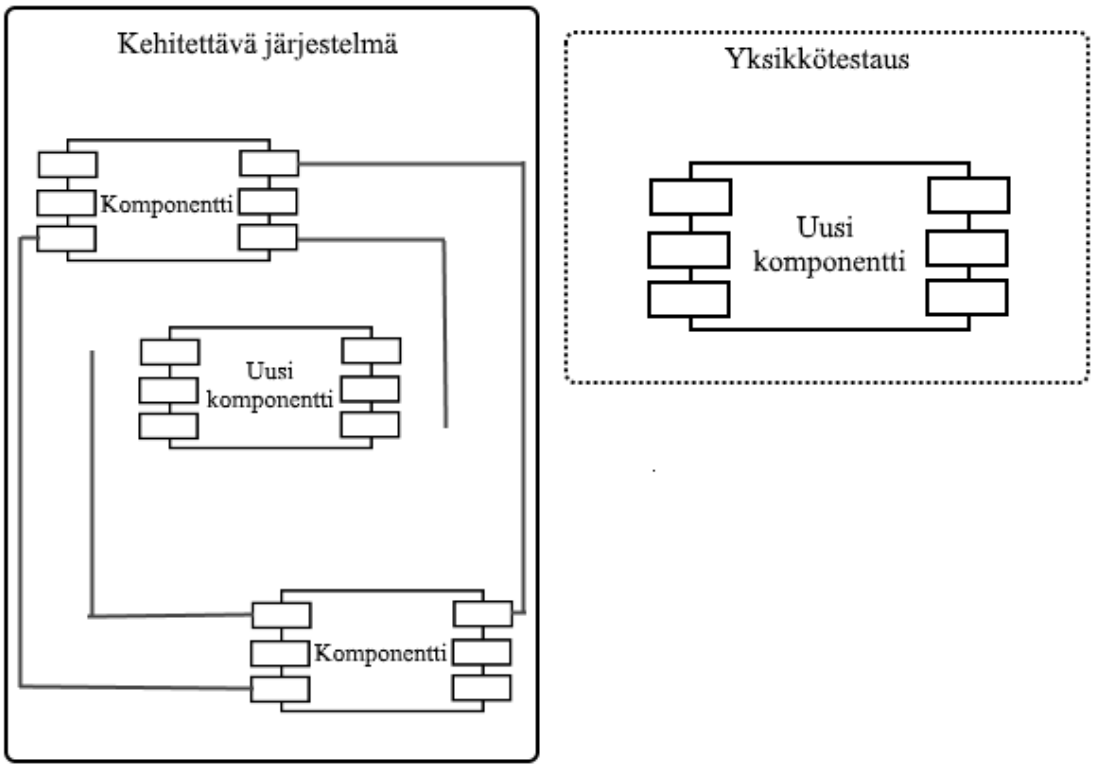
1. Yksikkötestauksella, joka kohdistuu yksittäisiin komponentteihin.
2. Integrointitestauksella, jossa keskitytään kahden tai useamman komponentin toimintaan yhdessä.
3. Järjestelmätestauksella, jossa yhtä tai useampaa järjestelmän osaa testataan yhtenä kokonaisuutena.

V-mallia noudattaen myös varsinaisen kehitystyön jälkeen tuleva hyväksymistestaus voidaan katsoa yhdeksi testauksen tasoksi. (Sommerville 2011)

4.2.1 Yksikkötestaus

Yksikkötestaus tarkoittaa testaustyötä, jossa yhden yksittäisen moduulin tai komponentin toimintaa tarkastellaan toteutuksen valmistuttua, kuten kuvassa 11. Testauksen suorittaa tyypillisesti komponentin toteuttanut henkilö. Tarkoituksena on varmistaa, että uusi komponentti tai muutos olemassa olevaan järjestelmään toimii ainakin periaatteessa ja se noudattaa ohjelmiston suunnittelussa tehtyjä dokumentteja. Testausta suorittaessa katsotaan, että tehty muutos kääntyy virheettä ja komponentti reagoi syötteisiin oikein. (Kasurinen 2013) Testaus kohdistuu siis ohjelmiston pienimpiin osiin, kuten metodeihin ja luokkiin. Testauksessa näitä komponentteja tulisi kutsua erilaisilla parametreilla, jotta saadaan katettua kaikki komponentin ominaisuudet. Tämän vuoksi on testattava kaikki komponenttiin liittyvät operaatiot ja simuloitava kaikki tapahtumat, jotka muuttavat komponentin tilaa. Testaus on kallista ja vie aikaa, joten suoritettavan testauksen on

oltava mahdollisimman tehokasta. Testauksen tulisi siis osoittaa, että kun komponenttia käytetään oikein, se myös toimii oikein. Lisäksi testauksen tulee paljastaa komponentin viat. Tarvitaan siis kahdenlaisia yksikkötestejä. Ensimmäisessä testityypissä simuloidaan komponentin normaalia toimintaa ja toisessa testityypissä komponenttia käytetään odottamattomalla tavalla. Jälkimmäisessä tarkoituksena on osoittaa, että komponentin saadessa virheellistä dataa se ei käsittele sitä, kuin se olisi oikein tai aiheuta kaatumista. Yksikkötestausta kannattaa automatisoida niin pitkälle, kuin mahdollista. (Sommerville 2011)



Kuva 11: Yksikkötestauksen kohteena on uusi komponentti järjestelmään (Kasurinen 2013)

Yksikkötestaukseen voi myös sisältyä vakaustestausta, rakenteellista testausta ja toiminnallisten tai ei-toiminnallisten piirteiden testausta. Tällaisia voivat olla esimerkiksi testit muistivuotojen varalta. Käytännössä yksikkötestausta tehdään usein ohjelmiston kehitysympäristössä hyödyntäen siihen yhdistettyä yksikkötestauksen apuvälinettä kuten virheenjäljitintä. Testaaja on usein kyseisen komponentin kirjoittanut ohjelmoija. Mahdolliset viat korjataan usein samassa yhteydessä ja raportointi virheistä saattaa olla muodollista. (FiSTB 2010)

Yksittäinen komponentti ei kuitenkaan aina pysty toimimaan itsenäisesti. Jos esimerkiksi komponentin tarkoituksena on hakea tietoa erillisestä tietokannasta, ei sen toimivuutta

voida testata ilman jonkinlaista apuvälinettä. Näitä apuvälineitä kutsutaan testipedeiksi. Testipetiin voi kuulua erilaisia ohjelman ympäristöä simuloivia osia, joita ovat testiajurit ja tyngät. Tyngät tai tynkämoduulit korvaavat ne komponentin toiminnan testaamiseen tarvittavat muut komponentit, joita ei ole vielä luotu. Testiajurit taas mahdollistavat komponentin toteuttamien palveluiden kutsumisen ja tulosten tarkastelun. Joskus pelkät tyngät eivät riitä, vaan testaamisen mahdollistamiseksi on luotava kokonaisia testiolioita, joiden rakenne imitoi tarkemmin viereisten komponenttien sisäistä toimintaa. Näitä olioita kutsutaan mock-olioiksi. (Haikala & Mikkonen 2011)

Koodi, jolla yksikkötestausta suoritetaan luo yleensä useita olioita testattavan luokan perusteella ja antaa niille ennalta määrättyjä syötteitä ja tarkastaa, että niistä seuraavat tulokset vastaavat suunniteltua. Testaus ja testitapausten toteuttaminen koodaamalla auttaa hahmottamaan varsinaista ohjelmiston lähdekoodia ja näin osaltaan auttaa parantamaan ohjelmistotyön laatua. (Kuha 2008)

Jotta testikomponenteista ei tulisi liian isoja ja monimutkaisia, niiden kannattaa keskittyä testaamaan vain rajattua määrää ominaisuuksia. Testikomponenttien koodin kirjoittamisessa pätevät samat säännöt, kuin ohjelmiston varsinaisen lähdekoodin tekemisessäkin. Testauskoodia täytyy pystyä samalla tavalla ylläpitämään mahdollisia tulevia testitapauksia varten tai jos on tarpeen suorittaa samoja testejä uudelleen myöhemmin. (Kuha 2008)

4.2.2 Integrointitestaus

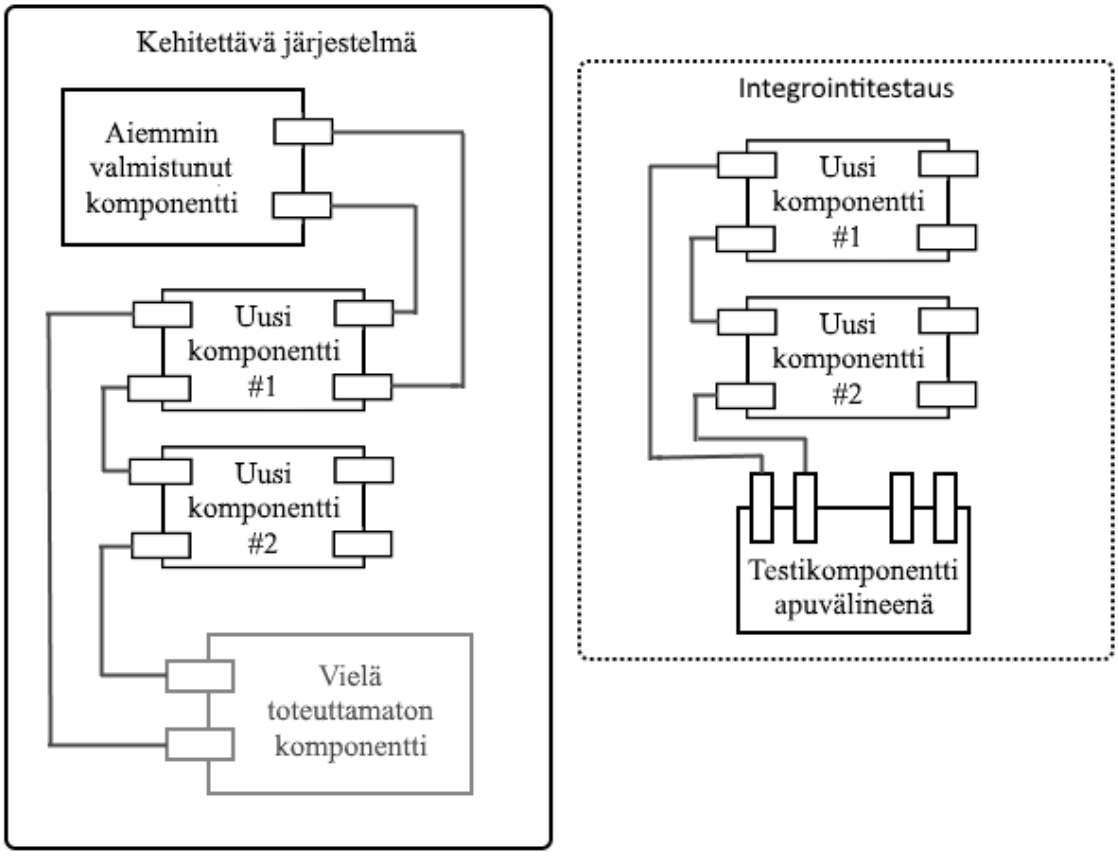
Integrointitestaus on yksikkötestauksen jälkeen seuraava työvaihe, mutta käytännössä sitä tehdään myös rinnakkain yksikkötestauksen kanssa. Integrointitestauksessa yhdistellään valmistuneita komponentteja ja kiinnitetään erityisesti huomiota siihen, että ne toimivat myös yhtenä kokonaisuutena, kuten kuvassa 12. Testauksen painopiste on siis komponenttien rajapintojen toimivuuden tarkastelussa. Saatuja tuloksia verrataan yleensä ohjelmiston teknisiin määrittelydokumentteihin. (Haikala & Mikkonen 2011)

Rajapinnoilla syntyvät ongelmat ovat yksi yleinen virheiden muoto monimutkaisemmassa järjestelmässä. Tällaisia ongelmia syntyy yleensä seuraavissa tapauksissa.

- Komponentti kutsuu väärää komponenttia, joka saattaa johtua esimerkiksi siitä, että komponentit on yhdistetty väärin tai kutsussa on parametreja väärä määrä tai väärässä järjestyksessä.
- Komponentti tulkitsee väärin kutsumaansa komponenttia, jonka seurauksena kutsuttu komponentti näyttää toimivan odottamattomalla tavalla. Tämä voi aiheutua esimerkiksi, jos kutsun parametrit ovat väärässä järjestyksessä.

- Jos järjestelmää on tarkoitus käyttää reaaliajassa, niin ongelmia voi syntyä myös, jos komponentit toimivat eri nopeuksilla tai eri aikaan.

Viallisia rajapintoja on kuitenkin vaikea testata, koska ongelmat saattavat ilmetä joskus vain hyvin epätavallisissa tilanteissa. (Sommerville 2011) Jos uusi komponentti sisältää liitäntöjä, joita ei ohjelmistossa vielä ole, niin yksikkötestauksen tapaan niihin liittyvien komponenttien testaamiseen on kehitettävä sopivat apuvälineet. Testitapaukset ovat myös laajempia, kuin yksikkötestauksessa, koska komponentteja on enemmän. (Kasurinen 2013)



Kuva 12: Integrointitestauksessa yhdistetään valmistuneita komponentteja (Kasurinen 2013)

Integrointitestauksen tasoja voi myös olla useampi kuin yksi ja ne jaetaan yleensä komponentti-integrointitestaukseen, jossa testataan komponenttien välistä vuorovaikutusta ja järjestelmäintegrointitestaukseen, jossa testataan eri järjestelmien välistä vuorovaikutusta. Kahta eri järjestelmää yhdessä testatessa voi kuitenkin syntyä riskejä, jos organisaatio ei pysty kontrolloimaan järjestelmien välisestä rajapinnasta kuin toista puolta. Mitä suuremmaksi integroitava kokonaisuus kasvaa, sitä vaikeammaksi sen hallitseminen muuttuu ja vikojen löytäminen hankaloituu. Integrointitestauksessa testaajien tulisi keskittyä meneillään olevaan integraatioon, eikä komponenttien mahdollisiin toiminnalli-

suuksiin, sillä ne on testattu jo yksikkötestauksessa. Ihanteellisessa tilanteessa testaajat ovat myös päässeet vaikuttamaan integrointitestauksen suunnitteluun. Integrointitestauksessa voidaan käyttää sekä toiminnallisia että rakenteellisia lähestymistapoja. (FiSTB 2010)

Integraatiotestauksen yhteydessä saatetaan testata myös muita komponentteihin liittyviä asioita. Tällaisia voi olla esimerkiksi transaktiomääritysten toiminta, kun useampi komponentti kutsuu niitä tietokantakäsittelijän yli. Samalla myös komponenttien määrittely voidaan tarkastaa toimivaksi. (Kuha 2008)

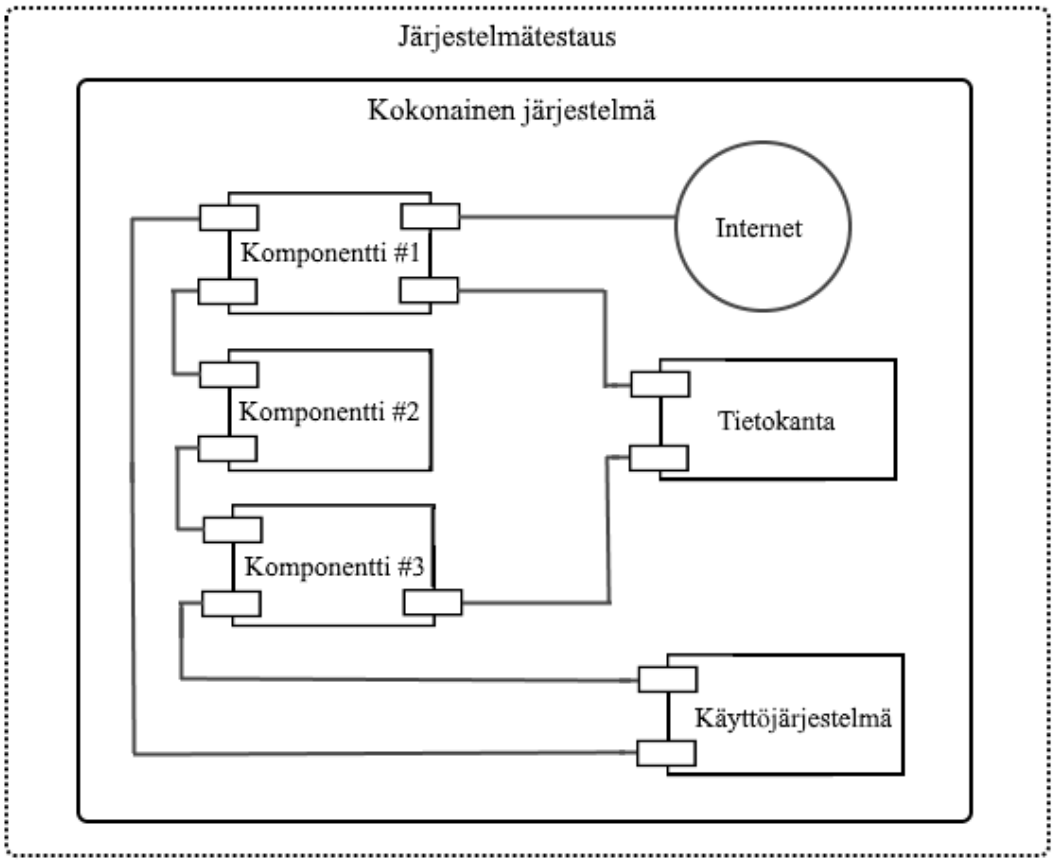
Jotta riittävän laajaa integraatiotestausta voidaan tehdä, tarvitaan usein valmista dataa tiettyjä testitapauksia varten. Usein ohjelmiston ensimmäisen sisäinen integraatiotesti vaatii paljon työtä, koska komponenttien väliset rajapintojen kytkennät pitää määritellä. Tämän jälkeen tulevat testaukset ovat kuitenkin helpompia ja nopeampia, koska testiympäristö on valmiina. (Kuha 2008)

4.2.3 Järjestelmätestaus

Järjestelmätestaukseen siirrytään kun komponentit ovat ensin käyneet läpi testauksen aikaisemmat tasot. Järjestelmätestaus on nimensä mukaisesti testausta, jota tehdään kokonaiselle järjestelmälle, kuten kuvassa 13. Se ei tarkoita itsessään mitään tiettyä testaustapaa, vaan se on yleisnimike kaikelle testaukselle, joka kohdistuu kokonaiseen järjestelmään eli ohjelmistoon, jossa ei ole enää mukana aiemmilla tasoilla käytettyjä testipetejä. Tavoitteena on varmistaa, että ohjelmisto toteuttaa sille asetetut tavoitteet ja toimii kokonaisuutena. Vaikka kyse on koko ohjelmiston testaamisesta, niin virheitä etsitään edelleen myös yksittäisistä komponenteista. Järjestelmätestaus suoritetaan testiympäristössä, ei varsinaisessa lopullisessa kohdeympäristössä. (Kasurinen 2013) Testiympäristön täytyy kuitenkin vastata lopullista käyttö- tai tuotantoympäristöä mahdollisimman tarkasti, jotta voidaan minimoida ympäristöjen eroista johtuvat virheet. (FiSTB 2010)

Järjestelmätestaus tarkastaa, että komponentit ovat yhteensopivia, vuorovaikuttavat keskenään oikein ja lähettävät oikeaa dataa oikeaan aikaan. Tätä tehdään luonnollisesti myös jo yksikkötestauksen aikana. Järjestelmätestauksessa kuitenkin suoritetaan muutamia toimenpiteitä ensimmäistä kertaa. Ensinnäkin uudelleenkäytettävät komponentit, jotka on kehitetty erillään ja mahdolliset kolmansien osapuolien kehittämät komponentit yhdistetään ohjelmistoprojektissa kehitettyjen uusien komponenttien kanssa. Testaus kohdistuu siis nimenomaan tähän kokonaan uuteen järjestelmään. Toisekseen myös projektin sisäisesti eri kehittäjäryhmien tekemät komponentit yhdistetään tässä vaiheessa. Järjestelmätestausta tehdään siis selkeästi enemmän kollektiivisesti, kuin yksilötasolla. Joissakin yrityksissä järjestelmätestauksen saattaa suorittaa myös kokonaan erillinen testausryhmä. (Sommerville 2011)

Järjestelmätestaukseen voidaan myös sisällyttää erilaisten ei-toiminnallisten ominaisuuksien testauksia, joihin kuuluu muun muassa kuormitustestit, luotettavuustestit, asennustestit ja käytettävyystestit. Kuormitustestaus tutkii kuinka hyvin järjestelmä selviää oletetusta ja sitä suuremmasta kuormituksesta. Luotettavuustestaus tutkii kuinka hyvin järjestelmä toipuu virhetilanteista ja kuinka pitkään järjestelmä toimii itsenäisesti ilman ongelmia. Asennustestaus tutkii, että onnistuuko järjestelmän asentaminen odotusten mukaisesti. Käytettävyystestaus tutkii miten hyvin järjestelmän suunnitellut käyttäjät saavat hyödynnettyä toteutusta toiminnassaan. (Haikala & Mikkonen 2011)



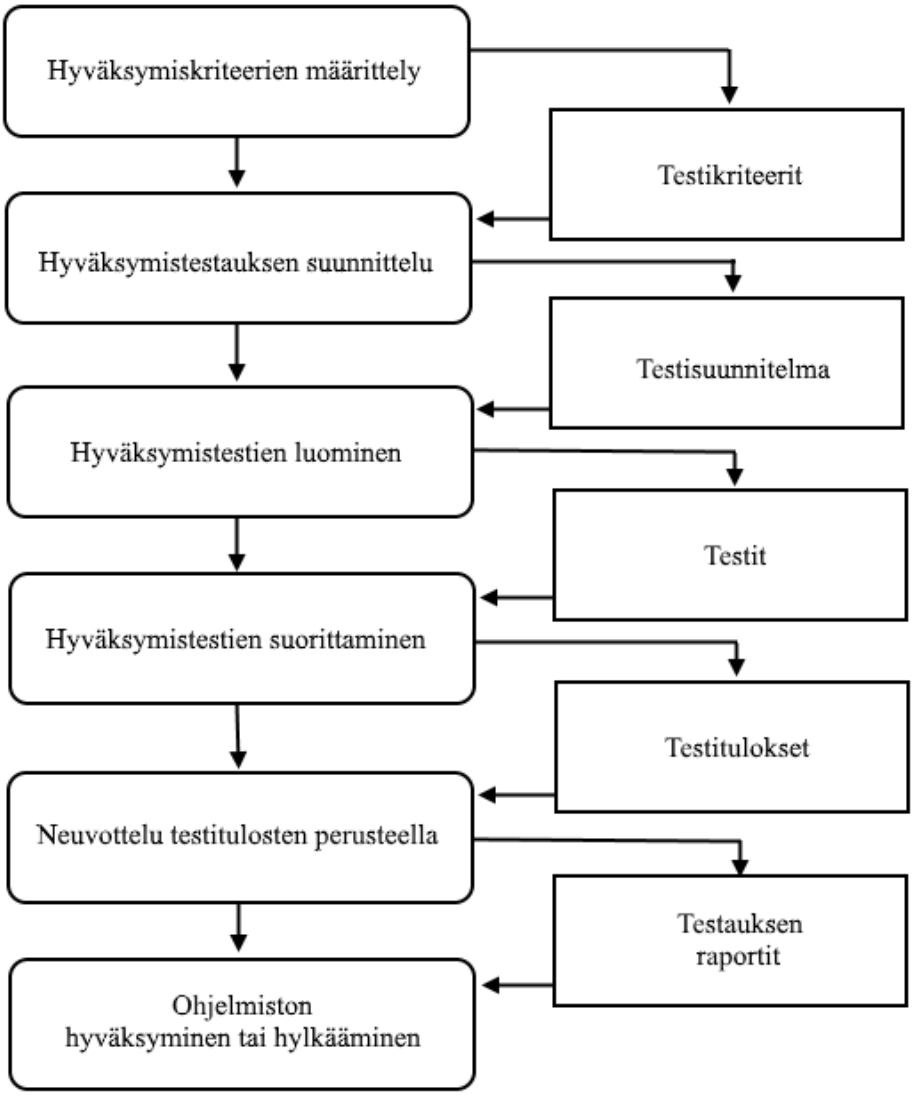
Kuva 13: Järjestelmätestauksen kohteena on koko järjestelmä (Kasurinen 2013)

Järjestelmätestausta muistuttaa myös julkaisutestaus, joka on testausprosessi, jossa testataan ohjelmiston tiettyä versiota, jota ollaan julkaisemassa käytettäväksi kehitysorganisaation ulkopuolella. Yleensä tämä tarkoittaa julkaisemista asiakkaille ja käyttäjille. Monimutkaisemmissa projekteissa kyseessä voi olla myös julkaisu projektin muiden tiimien käytettäväksi. Julkaisutestaus eroaa järjestelmätestauksesta sillä, että se tulisi hoitaa erillisen testausryhmän toimesta ja tarkastaa, että ohjelmisto täyttää riittävän hyvin

sille annetut tavoitteet, jotta se voidaan antaa ulkopuoliseen käyttöön. (Sommerville 2011)

4.2.4 Hyväksymistestaus

Hyväksymistestauksen tärkeimpänä tavoitteena on osoittaa ohjelmiston olevan riittävän laadukas ja että se täyttää sille asetetut vaatimukset. Esimerkki hyväksymistestauksen läpiviennistä nähdään kuvassa 14. Hyväksymistestaus on aidosti testausta, joka tehdään yleensä vasta ohjelmiston valmistumisen jälkeen. Kyseessä on järjestelmän virallinen tarkastus, jossa asiakas hyväksyy ohjelmiston eli tuotteen ja tämän jälkeen se siirtyy asiakkaan omaisuudeksi. (Kasurinen 2013)



Kuva 14: Esimerkki hyväksymistestauksen läpiviennistä (Sommerville 2011)

Hyväksymistestaus suoritetaan usein ohjelmiston varsinaisessa lopullisessa käyttöympäristössä. Hyvin oleellista ohjelmistotuotannon kannalta on se, että yleensä hyväksymistestauksen jälkeen ohjelmiston kehityksenaikainen korjausvelvoite ei ole enää voimassa ja mahdolliset tulevat muutokset ovat yleensä erillistä tilausta. (Kasurinen 2013)

Hyväksymiskriteerien määrittelyn tulisi olla osa tehtyä tilaussopimusta, joten tämä vaihe on tehtävä hyvin aikaisessa vaiheessa projektia. Käytännössä on kuitenkin vaikeaa määritellä kriteerejä näin aikaisessa vaiheessa. Yksityiskohtaisia vaatimusmäärittelyjä ei ole välttämättä vielä tehty ja tarpeet saattavat myös muuttua kehitystyön aikana. (Sommerville 2011)

Hyväksymistestauksen suunnittelussa tulee ottaa huomioon käytettävissä olevat resurssit, aika ja budjetti. Lisäksi on suunniteltava hyväksymistestauksen aikataulu ja missä järjestyksessä järjestelmän osat testataan. Huomioon on otettava myös riskit eli mahdollisesti eteen tulevat ongelmat ja miten ne voidaan minimoida. Tällaisia voivat esimerkiksi olla järjestelmän kaatuminen tai riittämätön suorituskyky. (Sommerville 2011)

Kun hyväksymiskriteerit on tiedossa ja testaus suunniteltu, voidaan luoda varsinaiset testit. Hyväksymistestien tulisi kohdistua sekä ohjelmiston toiminnallisiin, että ei-toiminnallisiin ominaisuuksiin. Ihanteellisessa tilanteessa testit kattavat kokonaan kohteena olevan ohjelmiston. Käytännössä tätä on kuitenkin vaikea toteuttaa objektiivisesti, koska joissain tilanteissa voi nousta erimielisyyksiä siitä, osoittaako testi kyseisen kriteerin toteutumisen riittävän kattavasti. (Sommerville 2011)

Sovitut hyväksymistestit suoritetaan ohjelmistolle. Parhaassa tapauksessa tämä tehdään varsinaisessa kohdeympäristössä, mutta tämä voi olla joissain tapauksissa epäkäytännöllistä tai jopa mahdotonta. Tämän vuoksi myös hyväksymistesteille saatetaan joutua luomaan oma testiympäristönsä. Hyväksymistestausta on hankala automatisoida täysin, sillä useat testit vaativat käyttäjän toimenpiteitä, jotta testi saadaan suoritettua loppuun määritysten mukaisella tavalla. Joskus käyttäjiä joudutaan kouluttamaan näitä testejä varten. (Sommerville 2011)

On epätodennäköistä, että kaikki määritellyt hyväksymistestit sujuvat täysin ongelmitta. Näissä tapauksessa ohjelmiston kehittäneen organisaation edustaja ja asiakas neuvottelevat testitulosten perusteella, onko hyväksymistestaus onnistunut riittävän laajoilta osin eli voidaanko se ottaa asiakkaalle käyttöön. Tässä vaiheessa neuvotellaan myös kehittäjän mahdollisista toimenpiteistä hyväksymistestien epäonnistuneiden kohteiden suhteen. (Sommerville 2011)

Kehittäjä ja asiakas päättävät yhdessä onko ohjelmisto riittävän onnistunut hyväksyttäväksi käyttöön. Jos näin ei ole, niin kehitystyö jatkuu löytyneiden ongelmien korjaamiseksi ja näiden korjausten jälkeen hyväksymistestit suoritetaan uudelleen. (Sommerville 2011)

Haasteena hyväksymistestauksessa on kuitenkin varmistaa, että valitut käyttäjät edustavat riittävän tyypillistä peruskäyttäjää, jotta tulokset eivät vääristyisi. Myös automatisoidun testauksen vaatimukset rajoittavat interaktiivisten ohjelmistojen testausta. Tällaisissa tapauksissa hyväksymistestaukseen saatetaan tarvita tavallista suurempi joukko käyttäjiä. (Sommerville 2011)

Hyväksymistestaus saattaa ajatuksena vaikuttaa yksinkertaiselta. Jos ohjelmisto ei selviä hyväksymistesteistä, niin ohjelmisto ei ole valmis ja asiakkaan ei tarvitse maksaa siitä. Käytännössä tilanne on kuitenkin monimutkaisempi. Ohjelmiston tilanneet asiakkaat haluavat kuitenkin päästä käyttämään tarvitsemaansa tuotetta mahdollisimman nopeasti, joten on myös heidän etunsa, että hyväksymistestaus suoritetaan joustavalla tavalla. Asiakkaalla saattaa olla esimerkiksi uutta laitteistoa ja jopa työvoimaa, joka vain odottaa, että he pääsisivät työskentelemään uuden ohjelmiston kanssa. Keskeneräisenkin ohjelmiston hyväksyminen käyttöön saattaa tulla edullisemmaksi, kuin odottaa jonkin vähemmän kriittisen korjauksen toteutumista. Tästä syystä ohjelmisto saatetaan usein hyväksyä tietyin ehdoin neuvottelujen tuloksena. Asiakas ottaa ohjelmiston käyttöön ja maksaa siitä kehitysorganisaatiolle, kunhan se sitoutuu korjaamaan havaitut ongelmat tietyssä aikataulussa. (Sommerville 2011)

4.3 Testauksen tasot kypsyyden perusteella

Toinen tapa luokitella testauksen tasot on tehdä se organisaation testausprosessin kypsyyden mukaan. Tasot on jaettu sen mukaan, mikä on testauksen tavoite. (Ammann & Offutt 2008)

- Taso 0. Testauksella ja virheiden etsimisellä ei ole eroa
- Taso 1. Testauksen tarkoitus on näyttää, että ohjelmisto toimii
- Taso 2. Testauksen tarkoitus on näyttää, että ohjelmisto ei toimi
- Taso 3. Testauksen tarkoituksena ei ole osoittaa mitään tiettyä, vaan vähentää ohjelmiston käyttämisestä muodostuvaa riskiä
- Taso 4. Testaus on mielentila, joka auttaa ohjelmistokehittäjiä tekemään laadukkaampia ohjelmistoja.

Testauksen tasolla 0 testaus on käytännössä vain virheiden etsimistä. Tällä tasolla ei vielä tehdä eroa virheen ja ohjelmiston väärän toiminnan välillä, eikä siksi juuri auta kehittämään luotettavaa tai turvallista ohjelmistoa.

Testauksen tasolla 1 testauksen tarkoituksena on osoittaa, että ohjelmisto toimii. Tämä on selvä parannus edelliseltä tasolta, mutta omaa edelleen oleellisen heikkouden. On helppoa osoittaa, että ohjelmisto toimii tietyissä tilanteissa, mutta on käytännössä mahdotonta osoittaa, että ohjelmisto toimii oikein aivan kaikissa tapauksissa. Jos testi osoittaa, että ohjelmistossa ei ole virheitä, niin sen perusteella ei voi tietää, onko kyseessä huono testi vai hyvä ohjelmisto.

Testauksen tasolla 2 testauksen tarkoituksena on osoittaa, että ohjelmistossa on virheitä. Virheiden löytäminen on helpompaa, kuin osoittaa, että ohjelmisto toimii täydellisesti. Tämä laittaa kuitenkin testaustyön ja kehitystyön vastakkain, sillä testaajan rooli yrittää osoittaa, että kehittäjän rooli on hoidettu epätyydyttävästi. Tasolla 2 säilyy kuitenkin edellisen tason ongelma eli mistä voidaan tietää, onko ohjelmisto kunnossa silloin, kun testaus ei löydä enää virheitä.

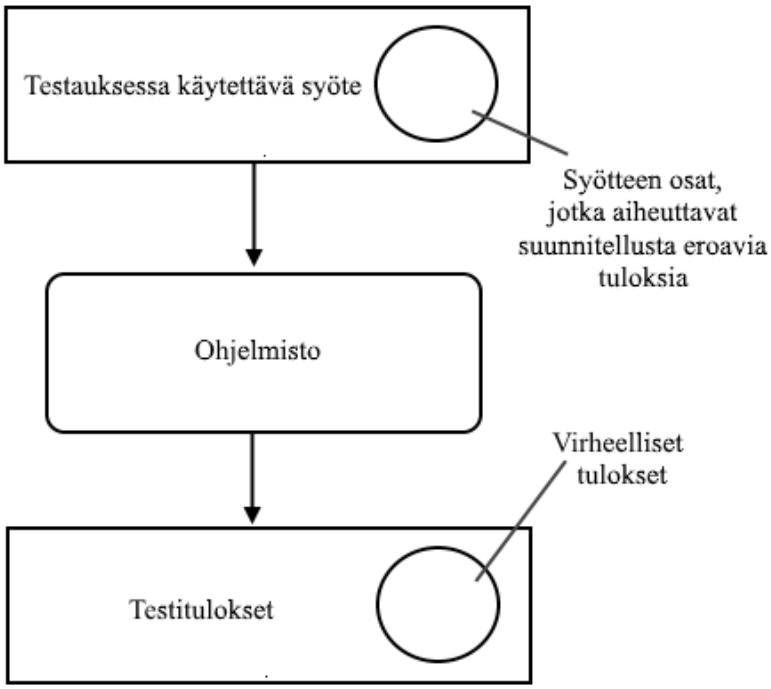
Testauksen tasolla 3 testauksessa hyväksytään totuus, että testauksella voidaan vain osoittaa, että ohjelmisto sisältää virheitä. Kun testauksessa ja kehityksessä hyväksytään, että ohjelmistojen käyttämiseen liittyy aina riskejä, niin molemmissa vaiheissa voidaan keskittyä näiden riskien minimoimiseen.

Testauksen tasolla 4 testaus aletaan mieltää toiminnaksi, jonka tarkoituksena on parantaa ohjelmiston laadukkuutta. Tähän on useita eri tapoja ja hyvien testitapausten luominen on näistä vain yksi. Samaan tapaan kuin tekstinkäsittelyssä automaattinen oikeinkirjoitus ei pelkästään auta kirjoittajaa löytämään virheitä, vaan se myös parhaimmillaan voi opettaa kirjoittamaan paremmin, niin tällä tasolla testauksen tarkoituksena ei ole pelkästään parantaa ohjelmiston laatua, vaan opettaa ohjelmistokehittäjää tuottamaan laadukkaampaa ohjelmistoa. (Ammann & Offutt 2008)

5 OHJELMISTOTESTAUKSEN MENETELMÄT

5.1 Musta laatikko

Testausmenetelmistä puhuttaessa ensimmäisenä tarkoitetaan yleensä menetelmää, jota kutsutaan mustaksi laatikoksi. Menetelmässä ohjelmistoa testataan antamalla sille syötteitä ja seuraamalla, mitä tuloksia saadaan. Yleensä syötteet on määritelty tarkkaan etukäteen testauksen suunnittelussa. Menetelmä on siis yksinkertainen ja sitä onkin mahdollista käyttää heti, kun ohjelmistosta on valmiina jokin toiminnallisuuden sisältävä komponentti. (Kasurinen 2013)



Kuva 15: Musta laatikko ei kerro mitä testin aikana ohjelmistossa tapahtuu (Sommerville 2011)

Mustaa laatikkoa kutsutaan joskus myös ulkoiseksi testaukseksi. Tässä menetelmässä on tärkeää huomata, että vaikka sillä voidaan paikallistaa ohjelmiston vialliset komponentit, niin se ei aina kuitenkaan paljasta syytä näihin virheisiin. Tämä johtuu siitä, että menetelmä ei kiinnitä huomiota siihen, mitä ohjelmistossa tapahtuu testauksen aikana. Tätä havainnollistetaan kuvassa 15. Nämä syyt täytyy selvittää valkoisen laatikon testauksel-

la. Molemmissa menetelmissä on kuitenkin tärkeää, että testauksessa käytetään sekä oikeita, että virheellisiä syötteitä. (Pohjonen 2002)

5.2 Valkoinen laatikko

Valkoinen laatikko on pidemmälle viety testausmenetelmä verrattuna mustaan laatikkoon. Kun mustassa laatikossa tarkkaillaan vain ohjelmistoon sisään menevää syötettä ja siitä seuraavaa tulosta, niin valkoisessa laatikossa tarkkaillaan myös, mitä ohjelmisto itse asiassa tekee näiden kahden vaiheen välillä. Testaaja siis tarkkailee, mitä ohjelmiston sisällä testauksen aikana tapahtuu. Näin nähdään, miten syötettä käsitellään ohjelmiston sisällä ja mahdollinen ongelma on mahdollista paikallistaa lähdekoodin tasolle asti. Näin tiedetään myös, että saatu tulos ei ole vain sattumaa. Tätä testausmenetelmää käyttävän henkilön on kuitenkin tunnettava ohjelmisto riittävän tarkasti, jotta hän voi luotettavasti sanoa onko ohjelmisto toiminut oikein vai väärin. (Kasurinen 2013) Valkoisen laatikon testausta kutsutaan usein myös sisäiseksi testaukseksi tai lasilaatikkotestaukseksi.

5.3 Harmaa laatikko

Harmaa laatikko on loogisesti testausmenetelmä, joka yhdistää mustan ja valkoisen laatikon menetelmiä. Tarkoituksena on luonnollisesti yhdistää molempien testausmenetelmien parhaat puolet. Menetelmä testaa yhtä aikaa vaatimusten täyttymistä ja lähdekoodin oikeaa suorittamista. Harmaa laatikko sopii tapauksiin, joissa testausta ei voida suorittaa puhtaasti valkoisen laatikon menetelmällä, mutta testattavan ohjelmiston yksittäisiä komponentteja saadaan kuitenkin tutkittua. (Kasurinen 2013)

5.4 Regressiotestaus

Regressiotestaus ei ole varsinainen testausmenetelmä, vaan sillä tarkoitetaan uudelleentestaamista yleisesti. Regressiotestaus on kyseessä esimerkiksi tilanteessa, jossa aiemmissa testeissä toimivaksi todettua järjestelmää muokataan jostain syystä ja suoritetaan uudet testaukset, jotta voidaan varmistua, että järjestelmä toimii yhä oikein. Kyseessä voi myös olla, että ohjelmistokehitys on saavuttanut tärkeän välietapin ja halutaan testata järjestelmä vielä uudelleen lävitse, jotta voidaan olla varmoja välietapin onnistumisesta. Regressiotestauksen lähtöajatuksena on, että virheitä löytyy todennäköisimmin järjestelmään lisättävistä uusista komponenteista tai komponenteista, jotka hyödyntävät näitä uusia komponentteja. Regressiotestauksen näkökulmasta siis, jos jotain järjestelmän osaa on muutettu, niin järjestelmää pitää ajatella kokonaan uutena järjestelmänä. Regressiotestauksella voidaan myös osoittaa, että kun versiohallinnan yhteydessä useampia kehityshaaroja yhdistetään toisiinsa, niin aiemmista versioista korjatut virheet eivät ole yhdistyksen myötä palanneet. (Kasurinen 2013)

5.5 Käytettävyystestaus

Käytettävyystestauksessa painopiste on ohjelmiston käyttöliittymän toimivuudessa ja intuitiivisuudessa. Käytettävyystestausta ei kuitenkaan varsinaisesti ole sidottu nimenomaan järjestelmätestaukseen. Iso osa käyttöliittymän suunnittelusta tehdään jo suunnitteluvaiheessa, joten samalla myös testaamista voidaan suorittaa yksinkertaisilla prototyypeillä, joissa ei ole muuta toiminnallisuutta kuin käyttöliittymän luonnos. Käytettävyystestaus pääsääntöisesti tarkoittaa kuitenkin testaustyötä, jolla tarkastetaan tehdyn järjestelmän käyttöliittymä ja varmistetaan, että se on suunniteltu oikein. (Kasurinen 2013)

5.6 Kuormitustestaus

Kuormitustestauksessa ohjelmisto laitetaan toimimaan sen suunnitellulla käyttäjämäärällä esimerkiksi luomalla virtuaalisia käyttäjiä, jotka suorittavat testitapauksia, jotka simuloivat normaalia ohjelmiston käyttöä. Kuormitustestauksella on kolme keskeistä tehtävää. Tunnistaa järjestelmän pullonkaulat, selvittää millaiseen maksimikapasiteettiin järjestelmä pystyy ja tarkastella sitä miten testattava laite selviytyy normaaleista käyttöolosuhteista. Pullonkauloilla tarkoitetaan mahdollisesti ongelmia aiheuttavia kohtia, kuten palvelimen välisten yhteyksien kykyä selvitä raskaasta ja jatkuvasta liikenteestä, tapahtumien vasteaikoja suorittimen ja muistin ollessa kuormitettuna tai levyjen luku- ja kirjoitusnopeuksien riittävyyttä. Mikäli järjestelmä ei kykene suoriutumaan toiminnoista välttävästi, kaatuu tai lakkaa toimimasta raskaalla kuormalla tai ei muutoin saavuta sille asetettuja laatuvaatimuksia, voidaan todeta kuormitustestauksen löytäneen korjausta vaativia virheitä. Kevyttä kuormitustestausta, jossa varmistetaan ohjelmistolla asetettujen minimimääritysten toteutuminen, kutsutaan joskus myös erikseen suorituskykytestaukseksi. Rasitustestaus taas on kuormitustestauksen pidemmälle viety muoto, jossa ohjelmistoa testataan suunniteltua suuremmilla käyttäjämäärillä tai raskaammalla kuormituksella. Tarkoituksena on löytää erilaisia lukkiutumisia ja muistivuotoja, jotka eivät välttämättä löytyisi normaalissa kuormituksessa. Näin voidaan löytää ohjelmiston ehdoton yläraja normaalille kuormitukselle. Kuormitustestaus vaatii paljon työtä ja erilaisia syötteitä, joten sen testaus on usein automatisoitu. Joissain tapauksissa kuormitustestaus saatetaan myös suorittaa päästämällä rajattu joukko käyttäjiä kokeilemaan ohjelmistoa. (Kasurinen 2013)

5.7 Savutestaus

Savutestaus on yleinen termi ohjelmiston testaamiselle yksinkertaisilla peruskokeilla, joilla varmistetaan, että perusasiat toimivat. Käytännössä tämä tarkoittaa perinteistä tarkastuslistaa, jossa tapaukset ovat hyvin yksinkertaisia, kuten ”käynnistyykö ohjelma”.

Savutestit ovat siis eräänlainen testauksen perustaso, joka suoritetaan ennen hienostuneemman testauksen aloittamista. (Kasurinen 2013)

5.8 Käyttäjätestaus

Käyttäjätestaus on testausvaihe jossa käyttäjät tai asiakkaat antavat palautetta tehtävästä ohjelmistosta. Tämä voidaan toteuttaa virallisena testauksena esimerkiksi ulkoistettuna ulkopuoliselle testausryhmälle tai kyseessä voi olla epämuodollisempi testaus, jossa käyttäjät pääsevät kokeilemaan uutta ohjelmistoa ja kertomaan mielipiteensä siitä. Ian Sommervillen mukaan käyttäjätestaus on hyvin tärkeää, vaikka kattavat järjestelmä- ja julkaisutestaukset olisikin suoritettu. Syynä tähän on, että käyttäjien oma työympäristö vaikuttaa suuresti ohjelmiston luotettavuuteen, suorituskykyyn ja käsitykseen ohjelmiston laadusta. On käytännössä joskus lähes mahdotonta organisaation suorittaa testausta samanlaisessa ympäristössä, kuin kohdeympäristö tulee olemaan. Esimerkiksi sairaalaan tehtyä ohjelmistoa käytetään kliinisessä ympäristössä, jossa tapahtuu paljon useita ohjelmiston käyttöä häiritseviä asioita, kuten potilaiden hätätapauksia ja keskeytyksiä omaisilta. Nämä kaikki vaikuttavat ohjelmiston käyttöön, mutta kehitysorganisaatio ei pysty realistisesti ottamaan näitä huomioon testausympäristössään. (Sommerville 2011)

5.9 Alfa- ja Beta-testaus

Alfa- ja beta-testausta tehdään pääasiassa sellaisille ohjelmistoille, jotka on suunniteltu yksittäiseksi kokonaisuudeksi. Tällaisia ohjelmistoja ovat esimerkiksi toimisto-ohjelmistot ja erilaiset videopelit. Termit ovat vakiintuneita näiden alojen ohjelmistotaloissa, mutta ovat harvinaisempia niiden ulkopuolella. (Kasurinen 2013) Kaupallisia ohjelmistoja tekevät organisaatiot haluavat yleensä saada palautetta potentiaalisilta käyttäjiltä ennen kuin tuote tulee laajaan kaupalliseen levitykseen. Alfatestaus suoritetaan useimmiten organisaation omissa tiloissa, mutta sitä eivät yleensä kuitenkaan tee ohjelmistoa kehittäneet henkilöt. Beta-testauksen taas tekevät kiinnostuneet asiakkaat omissa kodeissaan. (FiSTB 2010) Alfa-testausta saatetaan kuitenkin hyödyntää muissakin tapauksissa, kuin yksittäisissä tuotteissa. Myös jotkut ketterät kehitysmenetelmät kannattavat käyttäjien hyödyntämistä kehitystyössä ja että heidät pitäisi huomioida luotaessa testitapauksia. Testien tulokset hyväksytään vasta, kun ne miellyttävät käyttäjiä, joten tällaisessa mallissa ei tarvita erillistä hyväksymistestausta. (Sommerville 2011)

5.9.1 Alfa-testaus

Alfa-testauksessa käyttäjät ja ohjelmistokehittäjät tekevät yhteistyötä ohjelmiston testauksen suorittamiseksi. Käyttäjät voivat usein löytää ohjelmistosta tekijöitä, joita kehittäjät itse eivät ole tulleet huomioineeksi. Kehittäjät joutuvat työskentelemään ohjelmiston määrittelyn ja suunnittelun antamista näkökulmista ja eivät siksi pysty aina huo-

mioimaan kaikkia käytännönseikkoja. Tästä syystä käyttäjätestaus hyödyllistä ja näin kehitystyö täyttää paremmin käyttäjien varsinaiset tarpeet. Alfa-testausta käytetään erityisesti videopelejä kehitettäessä. Näiden ohjelmistojen käyttäjät eli pelaajat ovat kiinnostuneita olemaan mukana testauksessa, koska prosessi antaa heille tilaisuuden tutustua tuotteeseen etukäteen ja mahdollisesti hyötyä tästä tiedosta pelatessaan, kun tuote virallisesti julkaistaan. Jos taas kyse on jostain muusta kaupallisesta ohjelmistosta kuin peleistä, kuten esimerkiksi pienyrityksen taloushallintaohjelmistosta, niin tulevat käyttäjät tahtovat osallistua testaukseen, koska se vähentää heidän riskiään, että uudessa tuotteessa on heille haitallisia puutteellisia ominaisuuksia. (Sommerville 2011)

5.9.2 Beta-testaus

Beta-testauksessa kenties tarkoituksellakin keskeneräinen ohjelmisto julkaistaan käyttäjien arvioitavaksi. Beta-testaajiksi saatetaan valita suljettu ryhmä käyttäjiä tai ohjelmisto julkaistaan täysin vapaasti kaikkien halukkaiden kokeiltavaksi. Beta-testausta käytetään enimmäkseen tuotteissa, joita tullaan käyttämään useissa eri ympäristöissä. Ohjelmistokehittäjän on mahdotonta ottaa testauksessa huomioon ja luoda kaikki erilaiset käyttöympäristöt, joissa asiakkaat saattaisivat ohjelmistoa käyttää. Beta-testaus helpottaa siksi löytämään eroja esimerkiksi eri käyttöjärjestelmien ja tietokonejärjestelmien yhdistelmien kanssa. Beta-testaus saattaa toimia samalla myös osana tuotteen markkinointia. (Sommerville 2011)

5.10 Ad hoc-testaus

Ad hoc-testauksella tarkoitetaan suunnittelematonta ja dokumentoimatonta testausta. Kyseessä ei siis ole varsinaisesti mikään yksittäinen testaustapa, vaan yleistermi tällaiselle menettelylle. Yksi tai useampi henkilö siis testaa ohjelmistoa ja toteaa sen toimivan ilman varsinaista raportointia asiasta. (Kasurinen 2013)

5.11 Mallipohjainen testaus

Mallipohjainen testaus on lähestymistapa testaukseen, jossa testaus suunnitellaan niiden mallien perusteella, jotka ohjelmistosta on tehty sen suunnitteluvaiheessa. Testitapaukset luodaan sen mukaan, että ne käyvät läpi esimerkiksi UML-kuvauksilla toteutetut ominaisuudet ja jos testauksen tulokset ja suunnittelun mallit täsmäävät, niin voidaan todeta toteutuksen vastaavan suunniteltua. (Kasurinen 2013)

5.12 Virheiden kylväminen

Virheiden kylväminen on menetelmä, jossa virheitä suunnitellusti ja tarkoituksella lisätään ohjelmiston lähdekoodiin. Näin voidaan testata kuinka hyvin ohjelmiston testaamista varten tehty suunnitelma löytää nämä tiedetyt virheet ja tämän perusteella myös

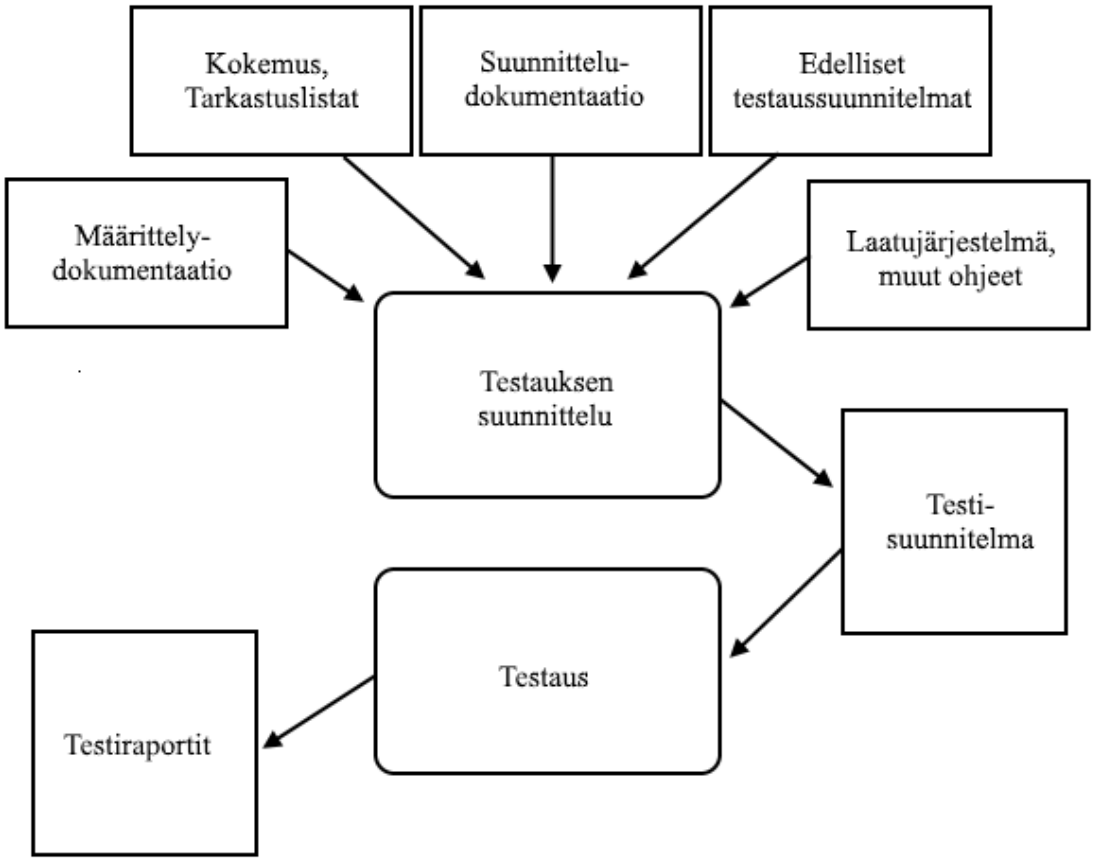
arvioida, että paljonko lähdekoodissa tilastollisesti on oikeita virheitä jäljellä. (Kasurinen 2013)

5.13 Mutaatiotestaus

Mutaatiotestaus on menetelmä, jonka ajatuksena on tarkoituksellisesti muunnella ohjelmiston toimintaa ja tarkastaa, pystyvätkö suunnitellut testitapaukset tunnistamaan viallisen toiminnan. Tehtyjen muutosten pitää kuitenkin olla sellaisia, että vain ohjelmiston toiminta muuttuu, mutta lähdekoodin syntaksi on edelleen oikein. (Kasurinen 2013)

6 OHJELMISTOTESTAUKSEN SUUNNITTELU

Testauksen suunnittelua aloitettaessa on hyvä tiedostaa, että varsinkin isommilla organisaatioilla on usein määritelty testauspolitiikka ja testausstrategia. Testauspolitiikka on ylemmän johdon luoma dokumentti, joka määrittelee, kuka testauksesta on vastuussa ja miksi testausta tehdään. Testauspolitiikka ei ota sen tarkemmin kantaa, miten testaus tulee suorittaa. Testausstrategia on huomattavasti laajemmin testausta määrittelevä dokumentti ja sen laatimiseen on tarvittu testausta ymmärtäviä henkilöitä. ISO/IEC 29119 standardi luettelee testausstrategian laatimisen apuvälineiksi muun muassa projektin laatuvaatimukset ja aikaisempien projektien loppuraportit. Erilaisia testauksen suunnittelussa huomioitavia tekijöitä on esitetty kuvassa 16. (Kasurinen 2013)



Kuva 16: Testauksen suunnittelussa huomioitavia tekijöitä (Haikala & Mikkonen 2011)

Ian Sommerville toteaa kirjassaan Software Engineering testauksella olevan kaksi erillistä tavoitetta ja molempien huomioimisen suunnittelussa olevan tärkeää. Ensimmäinen tavoite on osoittaa kehittäjälle ja asiakkaalle, että ohjelmisto täyttää sille asetetut vaatimukset ja tämän vuoksi testaussuunnitelmassa pitäisi olla vähintään yksi testi jokaista määriteltyä vaatimusta kohden. Toinen tavoite on löytää tilanteet, joissa ohjelmisto toimii virheellisesti. (Sommerville 2011) Ohjelmistotuotannossa kerätään yleensä toteutettavasta ohjelmistosta sille asetetut vaatimukset ja näiden avulla määritellään mitä ohjelmiston pitäisi tehdä. Näiden pohjalta voidaan jo aikaisessa vaiheessa laatia ensimmäisiä testitapauksia. Testitapaukset kuvaavat ohjelmiston toimintaa, kun sitä käytetään tietyllä tavalla. Testausta suorittaessa tuloksia verrataan näihin testitapauksiin. Suunnittelussa voidaan myös ottaa huomioon eri testitapausten riskit ja niiden vakavuus ja näin kohdentaa suunnitelmissa testauksen resursseja enemmän kohteisiin, joissa riski tai sen vakavuus on suurempi. Myös testausympäristön valitseminen tai luominen, jossa ohjelmistoa tullaan testaamaan, on osa suunnittelua. (Kasurinen 2013) Suunnitteluun vaikuttavat myös organisaation testauspolitiikka, testauksen laajuus, rajoitteet, kriittisyys, testattavuus sekä resurssien saatavuus. (FiSTB 2010)

6.1 Suunnitteludokumentti

Projektitasolla ohjelmistotestauksen suunnittelulla tarkoitetaan suunnitteludokumentin luomista, jossa määritellään mitä, missä vaiheessa ja millä menetelmällä ohjelmistoa testataan. Suunnitelma perustuu organisaation testausstrategiaan ja pyrkii noudattamaan organisaation tekemiä linjauksia, mutta poikkeukset voivat olla tarvittaessa mahdollisia. Testaussuunnitelman voi tehdä esimerkiksi projektipäällikkö tai muu testauksesta päättävä henkilö. (Kasurinen 2013)

6.2 Testitapaus

Testitapaus kuvaa yksittäistä tapahtumaa, jolla testataan jokin ohjelmiston toiminnallisuus. Testitapauksessa määritellään kaikki askeleet, joilla kyseinen toiminnallisuus saadaan ohjelmistoa käyttämällä suoritettua. Lisäksi määritellään kaikki ne työvaiheet, jotka itse testin suorittamiseen vaaditaan. Ensimmäiset testitapaukset luodaan normaalisti ohjelmiston määrittelyn pohjalta, jolloin tavoitteena on varmentaa määriteltyjen toiminnallisuuksien toteutuminen ja niiden pysyminen muuttumattomana myös kehitystyön edetessä. Testitapauksia myös lisätään projektin edetessä tarpeen mukaan, jos toiminnallisuuksia lisätään tai ohjelmoija tai testaaja löytää ongelman, jota tähän asti luodut testitapaukset eivät kata. (Kasurinen 2013) Hyviä esimerkkikeinoja testitapauksia luodessa on, että valitaan syötteitä, jotka pakottavat järjestelmän generoimaan kaikki eri virheilmoitukset ja syötteitä, jotka aiheuttavat puskurien ylivuotoja. Toistetaan samoja syötteitä tai syötteiden sarjoja useita kertoja uudelleen ja uudelleen. Pakotetaan järjestelmä ge-

neroimaan virheellisiä tuloksia ja tuloksia, jotka ovat vaatimansa muistin osalta liian suuria tai liian pieniä. (Sommerville 2011)

6.3 Riskianalyysi

Ohjelmiston riskianalyysi ei ole suoranaisesti ohjelmistotestauksen oma työväline, mutta sitä on silti testauksessa mahdollista hyödyntää. Riskianalyysi tehdään yleensä ohjelmiston määrittelyvaiheessa ja siinä arvioidaan riskejä ja riskien todennäköisyyksiä, jotka ovat uhkia ohjelmiston valmistumiselle. Suunniteltaessa ohjelmiston testausta tätä samaa riskianalyysiä voidaan käyttää kartoittamaan, mitkä seikat ovat erityisen tärkeitä testauskohteita. Jos johonkin ohjelmiston osaan liittyy korkea riski, niin sen testaaminen kannattaa suorittaa aikaisessa vaiheessa. Näin saadaan priorisoitua luotuja testitapauksia niiden kohteeseen liittyvän kriittisyyden perusteella. (Kasurinen 2013)

6.4 Testauksen lopetuksen määrittäminen

Osana suunnitelmaa on myös määriteltävä, milloin testaus voidaan lopettaa. Ihanteellisessa tilanteessa testaus lopetetaan vasta, kun ohjelmistosta löytyvät viat ovat niin pieniä ja harvinaisia, ettei niistä koidu taloudellista haittaa enemmän, kuin niiden korjaaminen maksaisi. Vaikka testitapausten valitsemiseen onkin useita menetelmiä, niin käytännössä kaikkia ei ehditä aina toteuttaa. (Kasurinen 2013)

Lopettamiskriteerejä voivat olla esimerkiksi testaukselle varatun ajan päättyminen, kaikkien testitapausten toteuttaminen tai kun testitapauksissa löytyneet virheet on korjattu ja regressiotestaus suoritettu tämän jälkeen. On tärkeää muistaa, että testauksen lopettamiskriteerit eivät ole sama asia, kuin testauksen hyväksymiskriteerit, sillä on aina myös mahdollista, ettei testausta saada suoritettua hyväksytysti lainkaan. (Haikala & Mikkonen 2011)

6.5 Ohjelmistotestauksen työvälineet

Myös käytettävien työvälineiden valitseminen on osa testauksen suunnittelua. Ohjelmistotestausta tehdään paljon samoilla työvälineillä, kuin millä ohjelmistoa tehdäänkin. Työvälineitä on muun muassa testitapausten hallinnointivälineet, yksikkötestauksen työkalut, testausautomaation järjestelmät, vikojen ilmoittamiseen tarkoitettuja järjestelmiä, testitapausten suunnittelu- ja kehitystyökaluja, lähdekoodianalysaattoreita, testiraporttigeneraattoreita ja simulaattoreita. Työvälineiden valinta saattaa riippua esimerkiksi siitä, mille alalle ohjelmistoa ollaan kehittämässä. Teollisuudelle kehitettävä ohjelmisto voi koskea tuotantolinjaa, jota ei ole mahdollista keskeyttää, joten testaaminen on suoritettava erillisessä testiympäristössä. (Kasurinen 2013)

6.5.1 Kehitysympäristöt

Kehitysympäristöt ovat tyypillisiä työvälineitä, joilla voidaan tehdä varsinaisen ohjelmointityön lisäksi myös testausta. Esimerkiksi yksikkötestausta tehdään kehitysympäristössä. Muun muassa C#, Java ja Python ovat sellaisia ohjelmointikieliä, että niillä on sisäänrakennettu tuki yksikkötestaukselle ja näin sen tekeminen kehitysympäristössä on entistä helpompaa. Kehitysympäristöillä tehdään usein myös staattista testausta jo ohjelmointityön aikana, sillä niissä on yleensä erilaisia sisäänrakennettuja ominaisuuksia, jotka tekevät oikeinkirjoituksen ja syntaksin tarkastusta. Kehitysympäristö siis auttaa virheiden havaitsemisessa ja näin estää niitä. (Kasurinen 2013)

6.5.2 Virheenjäljitin

Yksi edellytys ohjelmiston virheiden korjaamiselle on myös se, että virhe pystytään toistamaan. Jos vikatilannetta ei saada aiheutettua uudelleen, voi olla hyvin vaikeaa löytää sen aiheuttanutta virhettä. Toistettavissa olevien virheiden etsimiseen voidaan käyttää myös erillistä virheenjäljitintä, josta käytetään englanniksi termiä ”debugger”. Ennen virheenjäljittimiä koodia tutkittiin usein lisäämällä koodin sekaan ylimääräisiä tulostuslauseita, joiden avulla saatiin paikannettua, missä kohtaa koodia virhe tapahtuu. Tämä on toki edelleenkin käyttökelpoinen menetelmä. Virheenjäljitin on kuitenkin monipuolisempi testaustapa, koska sen avulla ohjelmiston suorittaminen voidaan pysäyttää halutussa kohdassa ja tarvittaessa tarkkailla koodin suoritusta vaikka rivi kerrallaan tuloksia seuraten. (Vesterholm & Kyppö 2010)

6.5.3 Testausautomaatio

Testausautomaatiossa tarkoituksena on ottaa joukko toistuvia tehtäviä ja automatisoida niiden suorittaminen. ”Keskimäärin 10 prosenttia testaustyöstä on automatisoitu siihen pisteeseen asti, että itse työn tekemiseen ei tarvita erillistä testaajaa.” (Kasurinen 2013, s. 12) Testausautomaation ohjelmat ovat yleensä käytännössä organisaation itsensä tekemiä työvälineitä. Varsinkin kun halutaan testata yksittäisten komponenttien toimintaa tai niiden välisiä rajapintoja. Testausautomaatio voi esimerkiksi lähettää komponenteille viestejä, kirjata saadut vastaukset ylös ja verrata niitä odotettuihin tuloksiin. (Kasurinen 2013) Aiemmin määritellyt testitapaukset on syötetty ohjelmaan, joka ajetaan aina, kun kehitettävää ohjelmistoa testataan. Tämä on yleensä nopeampaa kuin manuaalinen testaus varsinkin kun ajetaan samoja testitapauksia uudelleen, jotta varmistutaan, että muutokset ohjelmistoon eivät ole luoneet uusia ongelmia. Testaus ei voi kuitenkaan olla täysin automatisoitu, koska testausautomaatiolla voidaan vain tarkastaa, että ohjelmisto tekee mitä sen pitäisi. On käytännössä mahdotonta käyttää testausautomaatiota järjestelmiin, joiden toiminta riippuu siitä, mitä käyttäjä näkee. Testausautomaatio ei myöskään löydä odottamattomia sivuvaikutuksia, koska sitä ei ole voitu ohjelmoida etsimään sellaista. (Sommerville 2011)

6.5.4 Vikatietokanta

Vikatietokanta on työväline, johon kehittäjät, koekäyttäjät ja testaajat voivat kirjata ylös heille ohjelmistossa vastaan tulleet viat ja ongelmatilanteet. Näin muodostettua tietotokantaa voidaan hyödyntää suunniteltaessa ja seuratessa ohjelmistoon tehtäviä korjauksia. Samalla tietokanta toimii myös testaajien itsensä muistilistana, josta voi tarkastaa, onko kyseisen vian joku löytänyt jo aiemmin. Lisäksi tietokantaa voidaan hyödyntää laadunvalvonnan osoittamisessa ja sen kehitystä testaustyön etenemisen merkkinä. (Kasurinen 2013)

6.5.5 Tynkä

Tyngät ovat yksinkertaisia komponentteja, joita voidaan testauksessa hyödyntää esittämään sellaisia komponentteja, joita ei ole vielä tehty. Näin voidaan testata komponentteja, joiden on toimiakseen kommunikoitava toisten komponenttien kanssa. Tynkiä käytetään pääasiassa yksikkötestauksessa ja integrointitestauksessa. Integrointitestauksen edetessä tynkiä korvataan projektin edetessä varsinaisilla komponenteilla, kunnes tynkiä ei enää tarvita. Myös tyngät vaativat kuitenkin ylläpitoa ja vääränlainen tynkä voi aiheuttaa turhaan virheitä komponentteja testattaessa. (Kasurinen 2013)

6.5.6 Analysaattori

Analysaattorit tutkivat ja tarkastavat järjestelmän toimivuutta tunnettujen ongelmien varalta. Analysaattorit käyvät yleensä läpi joko komponentteja tai lähdekoodia. Analysaattori voi esimerkiksi käydä läpi kirjoitettua lähdekoodia ja varmistaa, että se noudattaa organisaation ohjelmointikäytäntöjä ja että se on syntaksinmukaista. Analysaattori voi myös arvioida suoritustehokkuutta, arvioida kuinka suuren osan testitapaukset kattavat koodista tai tunnistaa tilanteita, joissa kaksi komponenttia voivat jäädä jumiin syötteidensä takia. (Kasurinen 2013)

6.5.7 Dokumenttipohja

Valmiit dokumenttipohjat ovat tärkeä työväline, sillä hyvin hoidetussa testauksessa syntyy paljon erilaisia dokumentteja. Ohjelmistotuotannon käytännöt teos listaa seuraavia esimerkkejä asioista, joita virheen löytymistä koskeva dokumenttipohja voi sisältää. (Haikala & Mikkonen 2011)

1. Kuvaus virheestä
2. Milloin virhe löydettiin
3. Miten virhe löydettiin
4. Kuinka vakava virhe on
5. Milloin virhe on tehty
6. Miten virhe olisi voitu löytää jo aiemmin

7. Miten virhe oltaisiin voitu välttää

Dokumenttipohjat helpottavat ja nopeuttavat dokumenttien tekoa ja säästävät myös aikaa raporttia tehtäessä. Dokumenttipohjan tarkoituksena on varmistaa, että eri asiat muistetaan kirjata ylös ja samalla säästää testaajien aikaa, kun heidän ei tarvitse muotoilla dokumentteja erikseen. Valmis dokumenttipohja on myös helpompi lukea raportteja tutkittaessa ja näin tieto myös löytyy helpommin. (Kasurinen 2013)

7 YHTEENVETO

Kuten aluksi oletettiin, niin ohjelmistotestaus on hyvin laaja kokonaisuus, eikä sen mahduttaminen vain kehitystyön viimeiseksi vaiheeksi ole mahdollista ilman, että työmäärää karsitaan tai sen näkymistä ainakin jossain määrin ohjelmiston laadussa. Testauksessa löytynyt virhe on helpompaa ja edullisempaa korjata, kuin vasta valmiissa ohjelmistossa ilmenevä ongelma. Näin ollen testaus on kannattavaa sekä taloudellisesti että yrityksen maineen kannalta.

Ohjelmistokehitykseen valitut menetelmät vaikuttavat myös testaukseen ja sen suunnitteluun. Testaajan näkökulmasta on helppo ymmärtää halua hyödyntää ketteriä menetelmiä, koska niissä testauksen merkitys on huomioitu useimpia muita menetelmiä perusteellisemmin. Testaukseen käytettävien työvälineiden määrä on suuri, joka korostaa kokemuksen tarvetta testausta suunniteltaessa, jotta jokaiseen testauskohteeseen saadaan valittua mahdollisimman sopivat välineet.

Tämän diplomityön tavoitteena oli muodostaa kirjallisuustutkimuksen pohjalta yhtenäisempi ja laajempi käsitys ohjelmistotestauksesta, kuin millainen siitä yleensä esiintyy osana erilaisia ohjelmistokehityksen malleja. Tämä tavoite on saatu saavutettua, mutta luonnollisesti aihepiirissä riittää vielä muutakin tutkittavaa. Loogisia kohteita jatkotutkimukselle tämän diplomityön jälkeen olisivat erilaiset standardit ja sertifikaatit, jotka liittyvät ohjelmistotestaukseen suoraan tai sen kautta, miten ne vaikuttavat ohjelmistokehitysprosessiin kokonaisuutena.

LÄHTEET

Agile Alliance	Agile Alliance, (2001), Manifesto for Agile Software Development, http://agilemanifesto.org/
Ammann & Offutt	P. Ammann, J. Offutt, (2008), Introduction to software testing, Cambridge University Press, s. 8-10.
FiSTB	Finnish Software Testing Board, (2010), Sertifoitu testaaja perustason sertifikaattisisältö, s.14, s. 23-26, s. 48. http://www.fistb.fi/sites/fistb/files/liitteet/FL%20Syllabus %2020101123_0.pdf.
Haikala & Mikkonen	I. Haikala, T. Mikkonen, (2011), Ohjelmistotuotannon käytännöt, Talentun Media Oy, s. 36-51.
IEEE Computer Society	IEEE Computer Society, (2014), Guide to the Software Engineering Body of Knowledge, s. 82-84, https://www.computer.org/portal/web/swebok.
ISTQB	International Software Testing Qualifications Board, (2011), Certified Tester Foundation Level Syllabus, s. 14. http://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html.
Kasurinen	J. Kasurinen, (2013), Ohjelmistotestauksen käsikirja, Docendo, s. 10-88, s. 110-114, s.148-150.
Kuha	J. Kuha, (2008), Tehokas Java EE-sovellustuotanto, Docendo, s. 21, s. 81-83, s. 119.

Pohjonen R. Pohjonen, (2002), Tietojärjestelmien kehittäminen, Docendo, s. 35-36.

Royce W. Royze, (1970), Managing the Development of Large Software Systems, Proceedings of the IEEE WESCON, s. 1-9.

Sommerville I. Sommerville, (2011), Software Engineering Ninth Edition, Addison-Wesley, s. 29-32, s. 206-230.

Vesterholm & Kyppö M. Vesterholm, J. Kyppö, (2010), Java-ohjelmointi, Talentum Media Oy, s. 340-350.