OULUN SEUDUN
AMMATTIKORKEAKOULU

Teemu Saarinen, Niko Viinikanoja

TESTAUS JA SEN AUTOMATISOINTI

TESTAUS JA SEN AUTOMATISOINTI

Teemu Saarinen, Niko Viinikanoja
Opinnäytetyö
Kevät 2013
Tietojenkäsittelyn koulutusohjelma
Oulun seudun ammattikorkeakoulu

TIIVISTELMÄ

Oulun seudun ammattikorkeakoulu
Tietojenkäsittely

Tekijä(t): Teemu Saarinen, Niko Viinikanoja
Opinnäytetyön nimi: Testaus ja sen automatisointi
Työn ohjaaja(t): Ritva Virkkala
Työn valmistumislukukausi ja -vuosi: Kevät 2013 Sivumäärä: *41*

Tämän opinnäytetyön aiheena on testaus ja sen automatisointi. Työllämme ei ole varsinaista toimeksiantajaa, vaan aihe muotoutui keskustelujen tuloksena ohjaavan opettajamme kanssa. Aihe liittyy kuitenkin olennaisesti opintoihimme, joten meillä oli jo ennestään hieman tietoa kyseisestä asiasta.

Työn tavoitteena oli käsitellä testausta ja sen automatisointia yleisellä tasolla ja myös esitellä automatisoinnissa käytettäviä avoimen lähdekoodin ohjelmia. Tavoitteena oli siis saada aikaan tiivis teoriapainoitteinen teos, jonka avulla asiaan perehtymättömän lukijan on mahdollista saada yleiskäsitys sekä testauksesta yleensä että sen automatisoinnista.

Käytimme työssämme useita tietolähteitä, sillä aineistoa testauksen automatisoinnista löytyy kirjoista ja Internetistä runsaasti. Haasteenamme oli poimia niistä työmme kannalta oleelliset asiat.

Mielestämme saavutimme työlle asetetut tavoitteet. Tutkimme testausta ja sen automatisointia laaja-alaisesti teoreettisesta näkökulmasta. Jatkotoimenpiteitä työssä voisi olla teorian laajentaminen tai syvällisempi lähestyminen testaustyökaluihin.

Asiasanat: Testaus, automatisointi, automaattinen testaus, testaustyökalut

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Business Information Systems

Author(s): Teemu Saarinen, Niko Viinikanoja
Title of thesis: Test automation
Supervisor(s): Ritva Virkkala
Term and year when the thesis was submitted: Spring 2013 Number of pages: *41*

Topic of our thesis is software testing and software testing automation. Our thesis has no employer, but our objective was to find a subject that would be supported by our studies and benefit us after graduation as well.

The aim of our thesis is to give the reader a general idea on what software testing is about and the benefits and downsides of software testing automation. We also listed some of the commonly used Open Source tools that can be used to automate software testing.

There are plenty of literature and websites that contain information about software testing and software testing automation. The real challenge was to find the information that is closely related to our topic and also up to date.

All in all we think we managed to find the right information and succeeded quite well in achieving the goals we set for ourselves and this thesis: We managed to create a compact work that gives the reader a general idea about software testing and software testing automation.

Keywords: Testing, automation, test automation, testing tools

SISÄLLYS

1 JOHDANTO 7
2 OHJELMISTOTESTAUS 8
2.1 Testauksen määritelmä ja kehitys 8
2.2 Ohjelmistovirhe ja sen elinkaari 9
2.3 Testauksen riittävyys 10
3 TESTAUSPROSESSI 12
3.1 Testauksen tasot 13
3.1.1 Yksikkötestaus 13
3.1.2 Integrointitestaus 14
3.1.3 Järjestelmätestaus 14
3.1.4 Hyväksymistestaus 14
3.1.5 Regressiotestaus 15
3.2 Testausmenetelmät 16
3.2.1 Ad hoc 16
3.2.2 Staattinen ja dynaaminen testaus 16
3.2.3 Black Box 17
3.2.4 White Box 18
3.2.5 Grey Box 19
4 TESTAUKSEN AUTOMATISOINTI 20
4.1 Automatisoidun testauksen elinkaari 20
4.2 Automatisoinnin vahvuudet 22
4.3 Automatisoinnin tavoitteet ja edut 24
4.4 Automatisoinnin heikkoudet 26
4.5 Erilaisia tapoja automatisoida testausta 27
4.5.1 Makrojen nauhoitus ja toisto 27
4.5.2 Ohjelmoitavat makrot 27
4.5.3 Täysin ohjelmoitavat automaattiset testaustyökalut 28
4.5.4 Satunnaistestaus 29
5 AVOIN LÄHDEKOODI 31

6 TESTAUSTYÖKALUJA 32
6.1 Nauhoitustyökaluja 32
6.2 Ohjelmistokehyksiä 33
6.3 Suorituskykytestaustyökaluja 34
6.4 Muita testaustyökaluja 34
7 JOHTOPÄÄTÖKSET JA POHDINTA 36
LÄHTEET 38

1 JOHDANTO

Opinnäytetyömme aiheena on testaus ja sen automatisointi. Testauksen automatisointi tarkoittaa jonkin ohjelman tai ohjelman osan testaamista jollain sitä varten kehitetyllä ohjelmalla. Testauksen automatisoinnilla voidaan suorittaa uudestaan jo aiemmin tehtyjä testejä, sekä mahdollisesti automatisoida joitain testaustoimintoja kokonaan.

Opinnäytetyömme lähtökohtana oli ottaa selvää, miten testausta voidaan automatisoida ja missä tapauksissa testauksen automatisointi on kannattavampaa kuin manuaalinen testaus. Tavoitteenamme oli saada aikaan tiivis teoriapainotteinen työ, jonka avulla lukijan on mahdollista saada yleiskäsitys sekä testauksesta yleensä että sen automatisoinnista. Opinnäytetyön alussa käymme läpi ohjelmistotestausta yleisellä tasolla. Kolmannessa kappaleessa aiheena on testausprosessi, jonka jälkeen siirrymme testauksen automatisointiin. Työssä on esitelty myös muutamia automatisoinnissa käytettäviä avoimen lähdekoodin ohjelmia.

Työmme aihe syntyi ohjaavan opettajan kanssa käytyjen keskustelujen perusteella eikä sillä ole varsinaista toimeksiantajaa. Opinnäytetyön aihe liittyy olennaisesti opintoihimme, joten meillä oli jo ennestään pohjatietoa ohjelmistotestauksesta opintojaksoilla saamamme opetuksen perusteella. Työssä käytimme useita tietolähteitä, sillä aineistoa testauksen automatisoinnista on kirjoissa sekä Internetissä runsaasti. Haasteena oli poimia niistä työhömme oleelliset ja ajan tasalla olevat asiat.

Opinnäytetyön aihe on sikäli ajankohtainen, että koko ajan pyritään automatisoimaan enemmän asioita. Kuten yleisesti kaikissa muissakin asioissa, myös automatisoinnissa on omat etunsa ja heikkoutensa. Kuitenkin kustannustehokkuutensa takia automatisointia pyritään lisäämään kaikissa nykypäivän toiminnoissa mahdollisuuksien mukaan.

2 OHJELMISTOTESTAUS

2.1 Testauksen määritelmä ja kehitys

Testaus ei ole ohjelmistonkehityksessä erillinen osa, vaan se on prosessi, joka on mukana jokaisessa ohjelman kehitysvaiheessa. Testaus on iso osa ohjelmistokehitystä ja yleensä testaaminen tapahtuu useammalla kuin yhdellä tasolla. Puhekielessä testaus-termillä voidaan tarkoittaa miltei kaikkea erilaista kokeilemista. Perinteisesti testauksen tarkoitus on suunnitelmallisesti etsiä ohjelmasta virheitä suorittamalla ohjelmaa tai jotain sen pienempää osaa. Erityisesti tuotekehityksessä tapahtuva testaus ja sen myötä löydettyjen virheiden korjaaminen voi viedä tuotekehityksen kokonaisbudjetista jopa 80 %. Kustannuksista johtuen testauksella on lähes mahdotonta löytää kaikkia olemassa olevia virheitä. (Haikala & Märijärvi 2006, 40, 283.)

Ohjelmistoja on testattu niin kauan kuin ohjelmistoja on tehty. Käsitteenä testaaminen on kehittynyt ajan myötä. Testaustekniikoiden kehitystä ja tutkimusta ovat ohjanneet testauksen kohteiden ja itse testauksen määritelmän kehittyminen. David Gelperin ja Bill Hetzel jaottelivat testaustekniikoiden kehittymisen artikkelissaan “The growth of software testing” viiteen eri vaiheeseen:

Ensimmäinen vaihe oli aika ennen vuotta 1956, jolloin testausta ei oltu eritelty vaan se oli osa debuggausprosessia. Vaihetta voidaan pitää ohjelmistotestauksen esiasteena.

Toinen vaihe oli aika vuosien 1957 ja 1978 välillä, jolloin testauksen päämääränä oli varmistaa, että ohjelma vastaa sille asetettuja vaatimuksia.

Kolmas vaihe oli aika vuosien 1979 ja 1982 välillä, jolloin testauksen pääpaino siirtyi virheiden löytämiseen ohjelmasta.

Neljäs vaihe oli aika vuosien 1983 ja 1987 välillä, jolloin testauksella pyrittiin löytämään virheitä myös vaatimuksista ja suunnittelusta itse ohjelman lisäksi.

Viides vaihe on aika vuoden 1988 jälkeen, jolloin testauksella pyritään ennaltaehkäisemään virheitä vaatimuksissa, suunnittelussa ja ohjelman toteutuksessa. (Luo 2013, hakupäivä 9.1.2013; Gelperin & Hetzel 1988, hakupäivä 9.1.2013.)

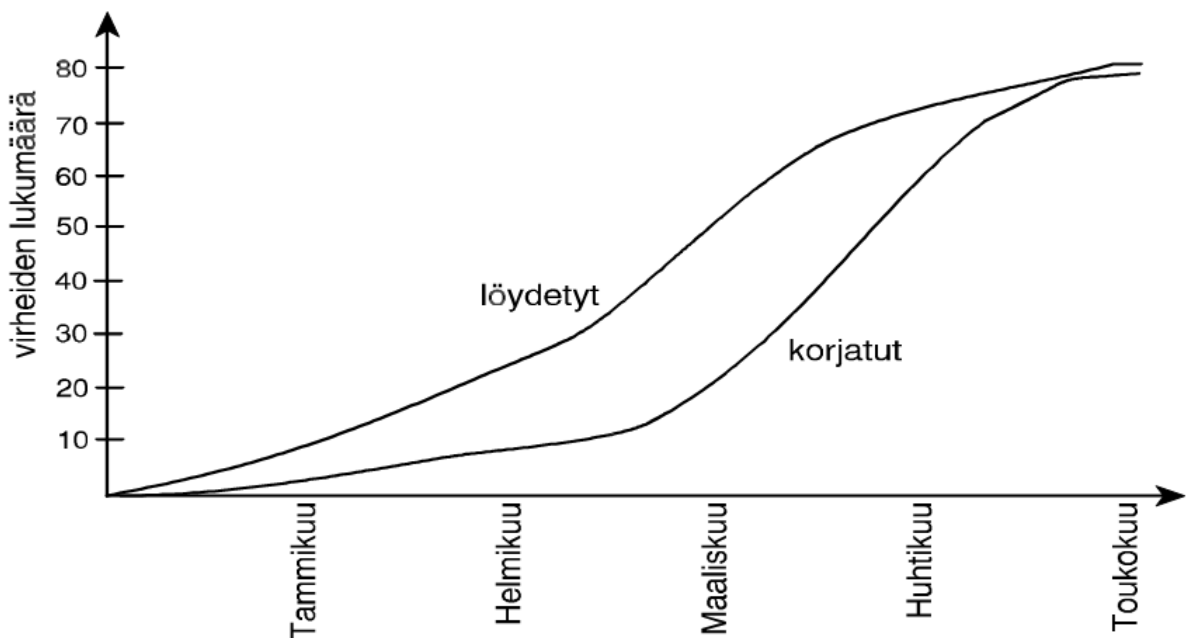
2.2 Ohjelmistovirhe ja sen elinkaari

Testauksen avulla mahdollistetaan ohjelmassa olevien virheiden osoittaminen. Ohjelmistovirheen vakavuus voi olla jokin pieni käyttäjää ärsyttävä yksityiskohta tai vaihtoehtoisesti koko ohjelman käytön estävä virhe. Ohjelman virheettömyyttä testauksella ei voida osoittaa missään olosuhteissa. Testauksessa virhe (error, bug) tarkoittaa poikkeamaa spesifikaatiosta. Yleensä testauksessa käytettäviä spesifikaatioita ovat toiminnallinen ja tekninen määrittely. Tämän vuoksi on tärkeää, että spesifikaatiot dokumentoidaan tarkasti, jotta asiakkaan (ohjelman tilaaja) ja toimittajan tulkintojen erimielisyyksiltä vältyttäisiin. Testaus ilman spesifikaatiota on mahdotonta, sillä ilman sitä ei voida tietää miten ohjelman tulisi toimia. Käytännön tasolla kuitenkin suurin ongelma on, että laadukkaimmatkin spesifikaatiot ovat aina puutteellisia, jolloin ristiriitatilannetta (mahdollista ohjelmavirhettä) ei voida ratkaista tutkimalla spesifikaatiota. Täten on tärkeää että tilaajan ja toimittajan välisiin sopimuksiin tulisi dokumentoida, miten tilanteiden tulkinnat näiltä osin tehdään. (Haikala ym. 2006, 287-288.)

Ohjelmoinnin jälkeen ohjelmissa arvioidaan yleensä olevan yksi virhe kymmentä ohjelmariviä kohden. Pitkään käytössä olleissa ohjelmissa arvioidaan olevan yleensä noin yksi virhe tuhatta ohjelmariviä kohden. Yleisesti on arvioitu, että 5 % ohjelmavirheistä jää kokonaan havaitsematta. Syinä tähän ovat yleensä syöteavaruuden suuri koko ja se, että virhetoimintoa ei välttämättä tapahdu, vaikka ohjelman virheellinen kohta suoritettaisiinkin. Näissä tapauksissa virheellisen kohdan suoritus voi aiheuttaa järjestelmän vian (fault), joka voi korjautua itsestään järjestelmän jonkin toisen toiminnon seurauksena. Kaikkein pahimmassa tapauksessa vika voi aiheuttaa häiriön (failure), mikä näkyy ohjelman ulkoisessa toiminnassa. (Haikala ym. 2006, 287-288.)

2.3 Testauksen riittävyys

Tarvittavan testauksen määrää on lähes mahdotonta arvioida. Yleinen käsitys on, että varsinkin järjestelmätestauksessa testausta voidaan jatkaa kunnes aika tai rahat loppuvat. Tuotteessa olevien virheiden aiheuttamien kustannusten ja markkinoilta myöhästymisen aiheuttaman tuoton menetyksen välillä tehty kompromissi on yleensä juuri testauksen lopettaminen. Kuitenkin testauksen lopettamiselle pitäisi aina asettaa niin sanotut hyväksymiskriteerit, jotka on määritelty testaussuunnitelmassa. Kriteeri voi liittyä esimerkiksi löydettyjen virheiden määrään: virhekäyrän tasaantuessa testaus voidaan lopettaa (KUVIO 1). Haastavan tästä tekee se, että tässä vaiheessa projektilla on kiinteät resurssit sekä aikataulu. On haastavaa arvioida projektin kestoa, mikäli testauksen hyväksymiskriteerinä pidetään vain virhekäyrän tasaantumista, sillä tällöin tarvittava työmäärä ei ole etukäteen tiedossa. (Haikala ym. 2006, 293.)



KUVIO 1. Testauksen virhekäyrä (Haikala ym. 2006, 293).

Moduulitestauksessa tarvittavan testauksen määrää voidaan yrittää arvioida niin sanotuilla mutkikkuusmitoilla ja testauksen riittävyyttä kattavuusmitoilla tai virheitä kylvämällä:

Mutkikkuusmitoilla (complexity measure) yritetään päätellä järjestelmän osan eli moduulin monimutkaisuutta ohjelmakoodin rakennetta analysoimalla. Tämän avulla voidaan paikantaa ohjelmistosta paljon testausta vaativat moduulit.

Kattavuusmitoilla voidaan koettaa varmistaa, että testattava aineisto aiheuttaa testattavan ohjelman kaikki osat kattavan suorittamisen. Kattavuusmitalla voidaan myös osoittaa, että testausta on suoritettu riittävästi. Kattavuusmittoja on useita erilaisia: lausekattavuus, päätöskattavuus, ehtokattavuus ja moniehtokattavuus.

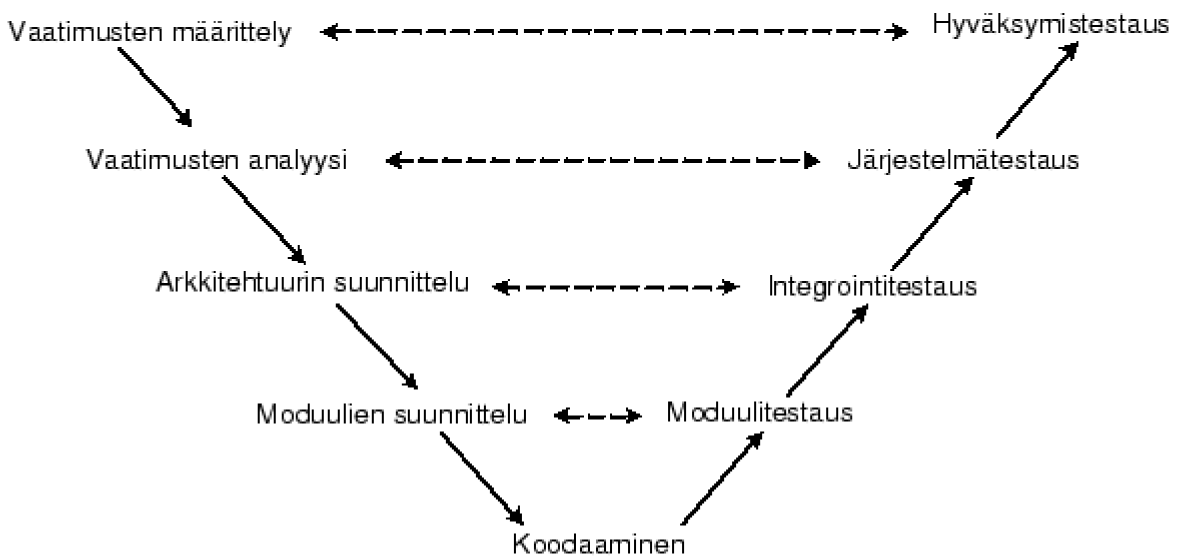
Polkutestauksessa (path testing) ohjelmia tarkastellaan suunnattuina verkkoina, joiden solmuina ovat ohjelman haarautumiskohdat. Polkutestauksessa tarkoituksena on kattaa mahdollisimman monia erilaisia suorituspolkuja. Aivan täydellinen polkukattavuus on lähes mahdotonta saavuttaa, sillä koko ohjelman mahdollisten polkujen määrä on ääretön. 100 % polkukattavuus on tavallisesti mahdollista saavuttaa vain yksinkertaisen funktion sisällä. Polkutestauksen ongelmana on, että kaikki olemassa olevat polut eivät ole suorituksessa mahdollisia.

Virheiden kylväminen (error seeding) on tahallista virheiden lisäämistä ohjelmakoodiin. Testauksen aikana löydettyjen virheiden määristä koetetaan päätellä ohjelmaan jääneiden virheiden lukumäärä. Käytännössä tätä ei juurikaan käytetä, koska yleensä ohjelmistotyössä toimitaan resurssirajoitteisesti ja lisätyön “aiheuttaminen” ei ole toivottavaa. On myös vaikea muistaa onko kaikki kylvetyt viat muistettu poistaa testauksen jälkeen.

Mutaatiotestauksella (mutation testing) tarkoitetaan tapaa, jossa ohjelmasta tehdään useita eri versioita, joista jokaiseen on kylvetty eri virhe. (Haikala ym. 2006, 294-296.)

3 TESTAUSPROSESSI

Testausprosessin loogiseen jaotteluun käytetään niin sanottua V-mallia (KUVIO 2). V-mallin avulla testaus voidaan jakaa osiin, jotka eivät ole riippuvaisia toisistaan muilla tavoin kuin suunnittelun osalta. V-mallin mukaisesti testaus alkaa pienimpien komponenttien testauksesta jatkuen aina kohti kokonaista järjestelmää. V-mallissa testausprosessi on jaettu loogisiin tasoihin, jotka on oltava suoritettu hyväksytysti, ennen kuin voidaan jatkaa seuraavalle tasolle. Lisäksi jokaisella ohjelmistotuotantoprosessitasolla suunnitellaan sitä vastaavan testauksen suunnittelu. Kuvassa testauksen suunnittelu tapahtuu aina testaustasoa vastaavalla suunnitelutasolla. Lisäksi ne on yhdistetty keskenään nuolella. (Confuse 2002, hakupäivä 24.1.2013.)



KUVIO 2. Testauksen V-malli (Confuse 2002, hakupäivä 24.1.2013).

Vaatimusmäärittelyvaiheessa otetaan huomioon asiakkaan vaatimukset sekä selvitetään miten valmiin ohjelman tulisi toimia. Tämän lisäksi määrittelyvaiheessa suunnitellaan hyväksymistestaus. (Waterfall Model 2013, hakupäivä 4.2.2013.)

Vaatimusten analysointivaiheessa järjestelmän ohjelmoijat käyvät läpi vaatimusmäärittelyn. He käyvät läpi erilaisia tapoja kuinka vaatimukset voidaan suorittaa. Kaikista muutoksista tulee ilmoittaa asiakkaalle. Lisäksi analysointivaiheessa suunnitellaan järjestelmätestaus. (Waterfall Model 2013, hakupäivä 4.2.2013.)

Arkkitehtuurin suunnitteluvaiheessa suunnitellaan miten ohjelmistovaatimukset täyttävä järjestelmä toteutetaan. Arkkitehtuurisuunnittelun tuloksena syntyvää dokumenttia kutsutaan tekniseksi määrittelyksi. Lisäksi tässä vaiheessa suunnitellaan integrointitestaus. (Waterfall Model 2013, hakupäivä 4.2.2013.)

Arkkitehtuurin suunnittelun jälkeen seuraa moduulien suunnittelu. Tässä vaiheessa järjestelmä jaetaan mahdollisimman itsenäisiin toisistaan riippumattomiin osiin, eli moduuleihin. Tämän jälkeen ohjelmoijat voivat aloittaa ohjelmoinnin ja ohjelmoida jokaisen moduulin erikseen. Lisäksi yksikkötestaus suunnitellaan tässä vaiheessa. (Waterfall Model 2013, hakupäivä 4.2.2013; Haikala ym. 2006, 39-40.)

3.1 Testauksen tasot

V-mallin mukaisesti erilaisia testaustasoja ovat yksikkötestaus, integrointitestaus, järjestelmätestaus ja hyväksymistestaus. Siirryttäessä V-mallissa alimmalta tasolta ylöspäin testauksen luonne muuttuu pienimmistä osista kokonaisuuteen. Lisäksi testattaville ohjelmille voidaan suorittaa regressiotestausta.

3.1.1 Yksikkötestaus

Yksikkötestauksessa testattavana on yksittäinen moduuli. Se koostuu yleensä noin 100–1000 ohjelmarivistä ja se on pienin yksittäinen testattava osa. Testattavan moduulin toimintaa verrataan tavallisimmin arkkitehtuurisuunnitelun tuloksen tekniseen määrittelydokumenttiin. Moduulitestauksen suorittaa yleensä moduulin ohjelmoija. Yksikkötestaus on tärkeää, sillä sen avulla mahdolliset virheet löydetään aikaisin. Mikäli yksikkötestausta ei tehdä huolella, se yleensä kostautuu testauksen myöhemmässä vaiheessa runsaasti aikaa vievänä vianetsintänä. Yksikkötestauksen suorittamiseksi voidaan toteuttaa testipetejä (test bed), joilla moduulien toimivuutta voidaan kokeilla. Testipetiin voi kuulua myös testiajureita (test driver), jotka simuloivat ohjelman ympäristöä, sekä tynkämoduuleja (test stub). Testiajurien avulla voidaan suorittaa moduulien toteuttamien palveluiden kutsuminen ja tulosten tarkastelu. Tynkämoduulien avulla sen sijaan voidaan korvata testattavan moduulin tarvitsemat muut moduulit, mikäli niitä ei ole vielä olemassa. Tynkäprosesseilla voidaan korvata osa prosesseista testiympäristössä.

Myöhemmin kyseiset tynkäprosessit korvataan prosessien lopullisilla versioilla. (Waterfall Model 2013, hakupäivä 4.2.2013; Haikala ym. 2006, 289-291.)

3.1.2 Integrointitestaus

Integrointitestauksessa yhdistellään moduuleita tai moduuliryhmiä ja testataan moduulien välisten rajapintojen toimivuutta. Integrointitestauksen tuloksia verrataan yleensä tekniseen määrittelyyn, kuten moduulitestauksessakin. Moduuleja voidaan integroida eri järjestyksissä. Yleisimpiä tapoja ovat kokoava ja jäsentävä integrointi. Kokoavassa integroinnissa (bottom up) edetään testaamalla alimman tason moduulit ensin ja jatketaan ylempänä oleville tasoille, kunnes kaikkein ylin taso saavutetaan. Tällöin testausprosessi on valmis. Jäsentävässä integroinnissa etenemissuunta on päinvastainen: testaus alkaa ylemmiltä tasoilta ja edetään alaspäin, kunnes kaikki moduulit on testattu. (Waterfall Model 2013, hakupäivä 4.2.2013; Haikala ym. 2006, 289-291.)

3.1.3 Järjestelmätestaus

Järjestelmätestauksessa testauksen kohteena on kokonainen järjestelmä ja siitä saatuja tuloksia verrataan ohjelmiston toiminnalliseen määrittelyyn ja asiakasdokumentaatioon. Järjestelmätestauksessa testaajina käytetään yleensä ohjelmiston kehitystyöstä mahdollisimman riippumattomia testaajia. Koska suurin osa toiminnallisista virheistä on löydetty jo edellisillä testaustasoilla, niin järjestelmätestauksessa testataan myös järjestelmän ei-toiminnalisia ominaisuuksia kuten kuormitus-, luotettavuus-, asennus- ja käytettävyystestit. Mitä korkeammalla V-mallin testaustasolla ollaan, sitä kalliimmaksi löydetyn virheen korjaaminen kasvaa. Järjestelmätestauksessa havaitun virheen korjaus voi aiheuttaa muutoksia useampaan moduuliin tai se voi aiheuttaa kokonaan uusia virheitä. (Waterfall Model 2013, hakupäivä 4.2.2013; Haikala ym. 2006, 289-291.)

3.1.4 Hyväksymistestaus

Hyväksymistestauksen perusteella tutkitaan täyttääkö testauksessa oleva järjestelmä sille annetut vaatimukset. Hyväksymistestauksessa siis testataan valmista tuotetta ja yleensä testaajana toimii järjestelmän loppukäyttäjä. Hyväksymistestaus on ajallisesti lyhyehkö vaihe, sillä

sen tarkoituksena on vakuuttaa loppukäyttäjälle, että sen vaatimukset on huomioitu. Tässä testausvaiheessa virheiden löytämisen ei tulisi olla tärkein prioriteetti, sillä hyväksymistestausvaiheessa virheiden korjaaminen on liki mahdotonta, koska se tulisi erittäin kalliiksi. Yleensä näissä tapauksissa virheen korjaaminen jätetään seuraaviin ohjelmaversioihin. Hyväksymistestaaminen voidaan jakaa kahteen eri vaiheeseen: alfa- ja beta-testaukseen. Alfa-testit suoritetaan yleensä kehittäjän tiloissa, mahdollisimman realistisessa ympäristössä. Beta-testaus sen sijaan suoritetaan loppukäyttäjän tiloissa ja ympäristössä. Tällä testauksella varmistetaan, että tuote vastaa vaatimusmäärittelyä sekä täyttää asiakkaan tarpeet. Beta-testaus voidaan suorittaa myös julkisesti laittamalla tuotteen beta-versio ilmaiseen levitykseen, josta käyttäjät voivat ladata ohjelmiston itselleen. (Waterfall Model 2013, hakupäivä 4.2.2013; Tampereen teknillinen yliopisto 2011, hakupäivä 4.2.2013.)

3.1.5 Regressiotestaus

Regressiotestauksella tarkoitetaan aiemmin testauksen kohteena olleen ohjelman uudelleentestaamista. Regressiotestauksen tarkoituksena on varmistaa, että ohjelmaan ensimmäisen testauksen jälkeen tehdyt korjaukset toimivat eivätkä ne aiheuta uusia virheitä. Tehdyt muutokset voivat olla ohjelmarivien poistamista, muuttamista tai lisäämistä. Muutoksista riippuen voidaan ohjelman regressiotestauksessa käyttää uudelleen edellisellä testauskerralla onnistuneesti läpäistyjä testitapauksia. Jos tehty muutos on tehty korjaamaan ohjelmasta löydetty virhe, käytetään regressiotestauksessa silloin virheen löytänyttä testitapausta. Lisäksi voidaan joutua tekemään uusia testitapauksia toiminnallisuuden tarkistamiseksi. Seurauksena ohjelmaan tehdyistä muutoksista osa testitapauksista voi kuitenkin olla käyttökelvottomia korjatussa ohjelmassa. Regressiotestaus voidaan jakaa kahteen erilaiseen ryhmään. Korjaava regressiotestaus tarkoittaa, että ohjelman määrityksiin ei ole tullut muutoksia ja ohjelmakoodiin on tullut vain pieniä muutoksia. Näissä tapauksissa on ominaista, että suuri osa vanhoja testitapauksia voidaan käyttää uudelleen. Korjaava regressiotestaus suoritetaan epäsäännöllisin väliajoin. Progressiivinen regressiotestaus tarkoittaa, että ohjelman määritykset ovat todennäköisesti muuttuneet ja ohjelmaan on tehty suuria muutoksia. Näissä tapauksissa ei ole mahdollista käyttää vanhoja testitapauksia uudelleen. Progressiivinen regressiotestaus suoritetaan yleensä tasaisin väliajoin. (Holopainen 2005, hakupäivä 12.2.2013.)

3.2 Testausmenetelmät

3.2.1 Ad hoc

Ad hoc -termillä viitataan yleensä sellaiseen testaukseen, joka ei perustu dokumentteihin tai vaatimusmäärittelyihin, vaan se suoritetaan ilman tarkkaa suunnitelmaa ja perustuu testaajan omaan inspiraatioon, luovaan ajatteluun ja kokemukseen. Ad hoc -testauksessa testaaja pyrkii löytämään ohjelmasta virheitä kaikin mahdollisin keinoin. Ad hoc -testausta käytetään paljon täydentävänä menetelmänä muiden vakiintuneempien testauskäytäntöjen tukena. (Chakraborty 2009, hakupäivä 29.1.2013.)

Ad hoc -testauksen vahvuutena on ajateltu löytää virheitä ohjelmasta sen kehityksen aikaisessa vaiheessa, jolloin korjaaminen on vielä halpaa ja nopeaa. Ad hoc -testaus voi myös paljastaa puutteita itse testausprosessissa, jos sen avulla paljastuu virheitä vielä testauksen myöhemmissä vaiheissa. Tämä menetelmä yksin käytettynä on kuitenkin erittäin epäluotettava käytettäväksi ohjelman laadun varmistukseen. (Chakraborty 2009, hakupäivä 29.1.2013.)

3.2.2 Staattinen ja dynaaminen testaus

Staattinen testaus on ohjelmistotestauksen muoto, jossa tutkittavaa ohjelmaa tai koodia ei suoriteta. Sen sijaan koodia, vaatimusmäärittelyä, ohjelman suunnittelua tai vastaavaa tutkitaan joko manuaalisesti tai automaattisesti ohjelman avulla, tavoitteena löytää virheitä. Staattisen testauksen tavoitteena on parantaa ohjelman tai sen sivutuotteiden laatua löytämällä virheitä ohjelmistotuotannon aikaisessa vaiheessa. Esimerkkejä staattisista testausmenetelmistä ovat muun muassa epäviralliset katselmoinnit, tekniset katselmoinnit, “läpikäynnit” (walkthrough), staattisen koodin analyysit ja erilaiset tarkastukset ammattilaisten johdolla. (Software Testing Mentor 2013a, hakupäivä 5.2.2013.)

Dynaamisella testauksella tarkoitetaan ohjelmistotestausta, jota tehdään suorittamalla testattava ohjelma. Dynaamista testausta käytetään analysoimaan esimerkiksi ohjelman suorituksen aikaista muistin käyttöä, prosessorin käyttöastetta, vasteaikoja ja ohjelman yleistä toimintakykyä. Esimerkiksi yksikkötestaus, integrointitestaus, järjestelmätestaus ja hyväksymistestaus voidaan mieltää dynaamiseksi testaukseksi. (Software Testing Mentor 2013b, hakupäivä 5.2.2013.)

3.2.3 Black Box

Black Box testing eli mustalaatikkotestaus (KUVIO 3) on ohjelmistotestauksen muoto, jossa ohjelman sisäiset rakenteet ovat testaajalle tuntemattomia. Mustalaatikkotestaus ei tarkastele ohjelmiston toteutusta, vaan tässä testauksen muodossa keskitytään testaamaan ohjelmaa erilaisilla syötteillä, joiden tuloksia verrataan odotettuihin tuloksiin. Tarkastellaan siis ohjelmiston ulkoisesti havaittavaa toimintaa. Mustalaatikkotestausta voidaan käyttää kaikilla testaustasoilla yksikkötestauksesta hyväksymistestaukseen: Mitä suurempi ja monimutkaisempi ohjelma on, sitä tärkeämpää testien läpäiseminen on. (Software Testing Fundamentals 2013a, hakupäivä 30.1.2013.)

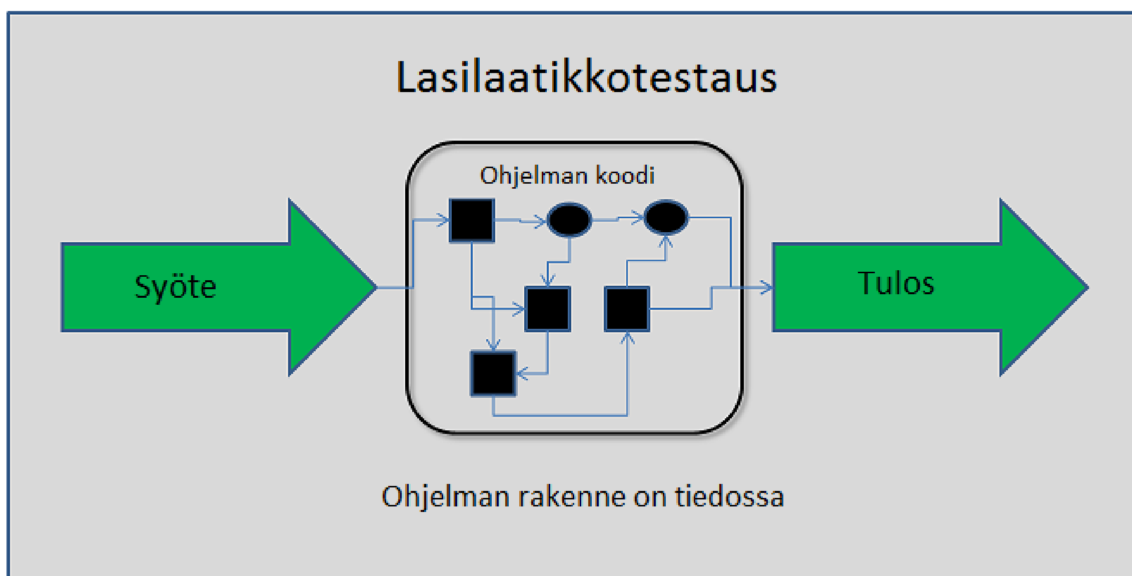
Mustalaatikkotestauksen eduiksi voidaan lukea esimerkiksi se, että ohjelmaa testataan käyttäjän näkökulmasta, jolloin myös epäjohdonmukaisuudet vaatimusmäärittelyssä tulevat todennäköisesti ilmi. Testaajan ei myöskään tarvitse osata ohjelmoida, eli mustalaatikkotestejä voidaan ajaa erillään ohjelmointityöstä. Tämä taas mahdollistaa neutraalimman näkökulman ohjelmistoon ja sen testaamiseen. Toisaalta jos vaatimusmäärittelyä ei ole tehty huolellisesti, on mustalaatikkotestien suunnittelu hankalaa. (Software Testing Fundamentals 2013a, hakupäivä 30.1.2013.)

KUVIO 3. Mustalaatikkotestaus (Muzak Studyzone, hakupäivä 13.3.2013).

3.2.4 White Box

White Box testing eli lasilaatikkotestaus (KUVIO 4) on ohjelmistotestauksen muoto, jossa ohjelman sisäiset rakenteet ovat täysin tiedossa testaajalle. Lasilaatikkotestauksessa ei testata ohjelmaa vaatimusmäärittelyn perusteella, vaan lasilaatikkotestauksessa testit suunnitellaan testattavan koodin rakenteen perusteella. Yleisimmin lasilaatikkotestausta käytetään yksikkötestauksessa ohjelman sisäisten polkujen testaamiseen, mutta sitä voidaan käyttää myös integrointitestauksessa eri yksiköiden keskinäisten polkujen testaamiseen, sekä järjestelmätestauksessa eri järjestelmien välisten polkujen testaamiseen. (Software Testing Fundamentals 2013b, hakupäivä 4.2.2013.)

Lasilaatikkotestauksen vahvuudeksi voidaan mainita se, että testaus on mahdollista aloittaa heti ohjelmoinnin alkuvaiheilla, koska esimerkiksi graafista käyttöliittymää ei välttämättä tarvita. Lasilaatikkotestauksella saadaan myös suuri kattavuus, sillä sen tarkoitus on käydä läpi mahdollisimman paljon itse koodissa olevia polkuja, joiden kautta ohjelman suoritus tapahtuu. Lasilaatikkotestaus voi joissain tapauksissa olla erittäin työlästä: jos koodia kirjoitetaan uusiksi, joudutaan myös testit kirjoittamaan täysin uusiksi. Lasilaatikkotestaus vaatii testaajalta myös paljon ammattitaitoa ja ohjelmaan perehtymistä, siinä missä mustalaatikkotestaus voidaan helpommin ajaa erillään ohjelmointityöstä. (Software Testing Fundamentals 2013b, hakupäivä 4.2.2013.)



KUVIO 4. Lasilaatikkotestaus (Muzak Studyzone, hakupäivä 13.3.2013).

3.2.5 Grey Box

Grey Box testing eli harmaalaatikkotestaus on yhdistelmä mustalaatikkotestausta ja lasilaatikkotestausta. Tässä ohjelmistotestauksen muodossa osa ohjelman rakenteista on testaajalle tiedossa. Yleinen tapaus voisi olla esimerkiksi seuraava: testaaja tutkii ohjelman sisäisiä rakenteita ja suunnitteluperiaatteita, jonka pohjalta hän luo testitapauksia (lasilaatikkotestaus). Nämä testitapauksen ajetaan kuitenkin käyttäen olemassa olevaa graafista käyttöliittymää (mustalaatikkotestaus). (Software Testing Fundamentals 2013c, hakupäivä 4.2.2013.)

4 TESTAUKSEN AUTOMATISOINTI

Automaattinen ohjelmistotestaus voi tarkoittaa eri asioita riippuen henkilöstä keneltä asiasta kysytään. Jotkut mieltävät termin testilähtöisenä ohjelmistokehityksenä tai yksikkötestauksena, toiset ajattelevat taas automaattisessa testauksessa käytettävää työkalua, jonka avulla nauhoitetaan ja toistetaan testejä. Termi voi myös tarkoittaa omien testiskriptien kehitystä eri skriptauskielillä, kuten Perl, Python tai Ruby. Joillekin se taas on vain suorituskykytestausta tai toiminnalliseen testaukseen keskittyvää testausta. Ohjelmistotestauksen automatisoinnille voi siis löytää monenlaisia määritelmiä. (Dustin, Garrett & Gauf 2009, 3.)

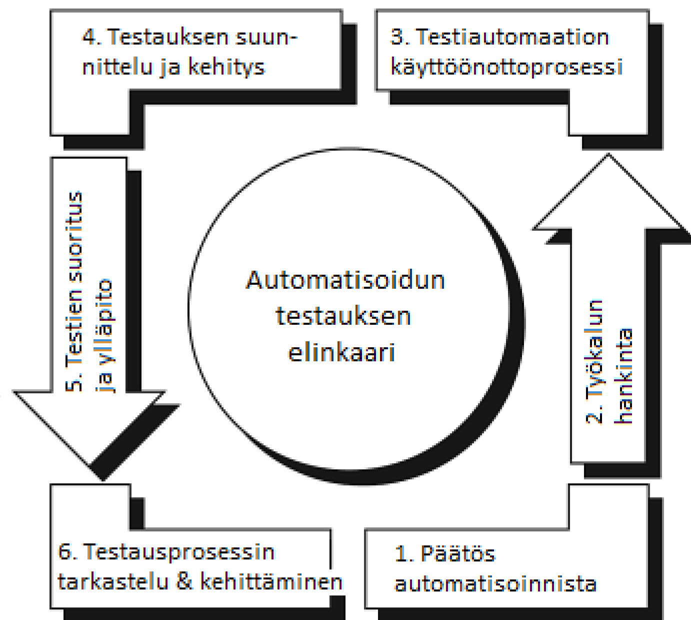
Zambelichin mukaan ihminen on edelleen tärkein osa testausjärjestelmässä. Hänen mukaansa testauksen automatisointi tarkoittaa manuaalisen testauksen suorittamista koneellisesti, jonkin testausohjelman avulla. Ennen kuin automatisointi on mahdollista, täytyy toimiva manuaalinen testausjärjestelmä olla olemassa. Manuaalisen järjestelmän täytyy sisältää ainakin yksityiskohtaiset testitapaukset sekä niiden odotetut tulokset, jotka perustuvat vaatimusmäärittelyihin ja suunnitteludokumentteihin. Toinen osa, joka jo manuaalisesta järjestelmästä tulisi löytyä on toimiva testausympäristö, jossa on testitietokanta, joka voidaan aina tallentaa uudelleen tietyssä vakiomuodossa siten, että testitapaukset voidaan suorittaa uudelleen, kun sovellukseen tulee muutoksia. (1998, hakupäivä 21.2.2013.)

Onnistuakseen testauksen automatisointi vaatii hyvää testausosaamista. Testitoimenpiteiden automatisointiin pätee samoja asioita kuin ohjelmistokehitykseen: Onnistumiseen tarvitaan standardeja ja ohjelmointikuria. (Pohjolainen 2003, hakupäivä 20.1.2013.)

4.1 Automatisoidun testauksen elinkaari

Testiautomaation käyttöönottoa suunniteltaessa on mietittävä tarkkaan, mitä on kannattavaa automatisoida. Muuten on vaarana automatisoida testejä, jotka kuluttavat paljon aikaa ja resursseja, mutta eivät löydä vikoja. Toisin sanoen vaikka testin voisi automatisoida, se ei aina ole kannattavaa. Automaatiosta saadaan paras hyöty, kun laitetaan tietokone ajamaan testejä, joihin ihminen ei kykene, kuten esimerkiksi kuormitustestausta. Ilman huolellista testauksen

suunnittelua voi automatisoidun testauksen tuloksena olla paljon testaustoimintaa ilman juurikaan oikeita tai haluttuja tuloksia. Testausta suunniteltaessa täytyy ymmärtää automaation tarjoamat erityismahdollisuudet tai on vaarana, että nämä mahdollisuudet jäävät käyttämättä. Jotta testiautomaatio voisi onnistua, tarvitaan päteviä testauksen suunnittelijoita sekä automatisoijia. Erikoisosaaminen takaa sen, että testeistä saadaan valittua ne, jotka kannattaa automatisoida, ja että tämä automatisointi suunnitellaan ja toteutetaan tarkoituksenmukaisesti. (Kaner, Bach, Pettichord 2002, 104-105.)



KUVIO 5. Automatisoidun testauksen elinkaari (Dustin, Rashka & Paul 2008, 9).

Ensimmäisessä vaiheessa tehdään päätös testiautomaation käyttöönotosta testausprojektissa, minkä jälkeen toisessa vaiheessa pitää pohtia, mikä työkalu hankitaan. Tässä vaiheessa testataan erilaisia vaihtoehtoja ja myös mietitään, onko järkevää rakentaa testaustyökalu itse. Ehdolle päässeet vaihtoehdot testataan ja pisteytetään soveltuvuuden mukaan. Lopuksi valitaan tarkoitukseen parhaiten soveltuva testaustyökalu ja käynnistetään pilottiprojekti. (Dustin ym. 2008, 10-12.)

Kolmannessa vaiheessa analysoidaan nykyistä testauskäytäntöä ja etsitään siitä kehitettäviä asioita, jotta automatisointi olisi mahdollista toteuttaa. Valittuun työkaluun tutustutaan syvemmin: katsotaan testattavan sovelluksen vaatimuksia, tutkitaan työkalun kelvollisuus sovelluksen

testaamiseen ja myös kokeillaan työkalua käytännössä. Tässä vaiheessa myös aikataulua korjaillaan tarpeen mukaan, jotta käyttöönottoprojektille on riittävästi aikaa. (Dustin ym. 2008, 12-13.)

Neljännen vaiheen olennaisin osa on testaussuunnitelma. Tämä suunnitelma sisältää muun muassa:
-roolit ja vastuualueet
-testauksen aikataulun
-testauksen suunnittelun ja siihen sisältyvät toiminnot
-testiympäristön määrittely ja valmistelu
-riskikartoituksen ja valmiussuunnitelman riskienhallintaan
-hyväksymiskriteerit

Näiden lisäksi suunnitelmasta löytyy tarvittaessa testausprosessit, nimeämiskäytäntö, noudatettavat standardit sekä testauksen jäljitysmatriisi. (Dustin ym. 2008, 13.)

Viidennessä vaiheessa testausryhmä suorittaa yksikkö-, integrointi-, järjestestelmä- ja hyväksymistestit laaditun aikataulun mukaisesti ja dokumentoi saadut tulokset. Ryhmän tulee pyrkiä ymmärtämään testauksessa ilmitulleet virheet ja niiden alkuperä, jotta virheet ovat toistettavissa korjaustoimenpiteitä varten. Testaus voidaan katsoa suoritetuksi, kun testaussuunnitelmassa määritellyt hyväksymiskriteerit on saavutettu. (Dustin ym. 2008, 14.)

Kuudennessa vaiheessa arvioidaan itse testausprosessia kerätyn datan perusteella: Ryhmän keräämä tieto testausprosessin kulusta kerätään yhteen ja katselmoidaan, jotta nähdään kuinka varsinainen testausprosessi vastasi suunniteltua testausprosessia. Tämän katselmoinnin pohjalta laaditaan kehitysideoita, joilla voidaan parantaa tulevien projektien testauskäytäntöjä ja täten varmistaa testausprosessin laatu jatkossa. (Dustin ym. 2008, 14.)

4.2 Automatisoinnin vahvuudet

Automaattisen testauksen suurin vahvuus piilee sen kyvyssä tehdä asioita, joita manuaalisella testauksella ei ole mahdollista tai järkevää tehdä. Automaattinen testaus kannattaa ottaa avuksi esimerkiksi testeihin, jotka sisältävät paljon toistoa tai testeihin jotka veisivät muuten ihmiseltä

paljon aikaa suorittaa. Tällaisesta testauksesta hyvänä esimerkkinä olisi yksikkötestien suorittaminen. (Myers, Sandler, Badgett & Thomas 1979, 184.)

Kaner, Bach ja Pettichord kirjoittavat, että automaattisella testauksella on muitakin vahvuuksia kuin toistojen suorittaminen. Seuraavat kappaleet ovat heidän teoksessaan “Lessons learned in software testing : a context-driven approach” antamia esimerkkejä joistain tilanteista, joissa automaattinen testaus on vahvimmillaan. (2011, 96.)

Kuormitustestauksella otetaan selvää miten järjestelmä tai ohjelma kestää useita samanaikaisia käyttäjiä, oli sitten puhe kymmenistä, sadoista tai tuhansista käyttäjistä. Tällaiset testit on helpointa hoitaa simuloimalla käyttäjiä automaattisella testauksella.

Suorituskykytestauksella mitataan esimerkiksi järjestelmän suorituskykyä erilaisissa tilanteissa erilaisiin aikoihin. Oikeilla testaustyökaluilla voidaan automatisoida testit ja tulosten keruu, jolloin tehtäväksi jää vain tulosten katselmointi ja johtopäätösten tekeminen. Tuloksista voi poimia esimerkiksi testaukseen kuluneen ajan, jonka perusteella suorituskykyä on helppo tarkkailla ja mahdollisiin poikkeamiin puuttua.

Konfiguraatiotestauksen tarkoituksena on selvittää miten ohjelma toimii erilaisilla käyttöjärjestelmillä, asetuksilla ja oheislaitteilla. Tällaisissa tapauksissa automaattinen testaus auttaa laajentamalla testauksen kattavuutta verrattuna siihen mitä ihminen saisi käsin testattua samassa ajassa. Jotta tämä toteutuisi täytyy testeistä kirjoittaa sellaisia, että ne voidaan siirtää alustalta toiselle vaivattomasti.

Kestävyystestauksella otetaan selvää kuinka testattava tuote toimii vielä viikkojen, kuukausien tai jopa vuosien yhtäjaksoisen käytön jälkeen. Näissä tapauksissa syntyy usein virheitä, jotka eivät ole välittömästi näkyviä, mutta aiheuttavat ongelmia pidemmän päälle. Yleisiä esimerkkejä tällaisista tapauksista ovat muun muassa muistivuodot (memory leaks), pinojen korruptoituminen (stack corruption) ja villit osoittimet (wild pointers). Automatisoimalla voidaan helposti toteuttaa testejä, jotka testaavat laitetta yhtäjaksoisesti päivien tai viikkojen ajan ilman että testattavaa laitetta käynnistetään uudelleen.

“Race conditions” -termillä tarkoitetaan tilannetta, jossa kaksi säiettä tai prosessia koittavat varata saman resurssin aiheuttaen kilpailutilanteen, joka johtaa virheeseen. Tämä virhe vaatii tiettyjä olosuhteita ja tiettyä ajoitusta, jota on vaikea löytää käsin testaamalla. Automatisoinnilla saadaan tällaisissa tilanteissa ajettua sama testi pienin muutoksin läpi vaikka tuhansia kertoja, jolloin virheen löytämisen todennäköisyys kasvaa huomattavasti.

“Combination errors” -termi kuvaa tilannetta, jossa ohjelman eri ominaisuudet ovat vuorovaikutuksessa keskenään synnyttäen samalla virheen. Automatisoinnilla on mahdollista ajaa suuria määriä monimutkaisia testejä, jotka käyttävät näitä eri ominaisuuksia vuorovaikutuksessa keskenään.

Automaattinen testaus mahdollistaa testien toistettavuuden. Testaaja saattaa törmätä työssään tilanteeseen, jossa hän on löytänyt virheen ohjelmasta, mutta ei kykene toistamaan tilannetta käsin testaamalla. Tällaisessa tilanteessa automaattinen testityökalu on avuksi, sillä testaaja voi toistaa jo ajetun testitapauksen uudelleen samassa muodossa. (Dustin ym. 2009, 45.)

Useat automaattiset testityökalut tarjoavat myös mahdollisuuden testien ajoittamiseen esimerkiksi tiettyyn kellonaikaan. Tämä ominaisuus mahdollistaa testauksen ajamisen tai jatkamisen myös sen jälkeen kun ihmiset ovat jo päättäneet työpäivänsä. Testaukseen voidaan myös käyttää useampia tietokoneita ja käyttää niitä hajautettuihin ja yhtäaikaisiin testeihin silloin, kun niiden resurssit ovat vapaana. (Dustin ym. 2009, 45-46.)

4.3 Automatisoinnin tavoitteet ja edut

Automatisoinnille tulee asettaa selkeät tavoitteet. Näitä tavoitteita voivat olla esimerkiksi testikierrokseen käytettävän ajan vähentäminen, testien kattavuuden lisääminen tai manuaalisesti suoritettavan testauksen vähentäminen, joka puolestaan tuo kustannussäästöjä pitkällä aikavälillä. Lähtökohta testauksen automatisointiin on väärä, jos tarvoitteena on kaiken testauksen automatisointi, sillä se ei ole mahdollista. Automatisoinnin ansiosta vapautuneet resurssit, kuten testaajien säästämä aika, kannattaa mahdollisuuksien mukaan siirtää vaativampiin testauskohteisiin. (Hendrickson 1998, hakupäivä 21.1.2013.)

Hyödyn maksimoimiseksi täytyy selvittää, mitä kannattaa automatisoida. Yksinkertaisimmin tämä onnistuu tutkimalla sitä, mihin testeihin on käytetty eniten aikaa. Usein suoritettavissa testitapauksissa on yleensä jo toimiva ja luotettava rakenne, ja niiden automatisointi on ajan säästön kannalta järkevää. Sitä vastoin vähän manuaalisesti läpikäydyt testitapaukset eivät välttämättä ole järkeviä kohteita automatisoinnille, sillä on mahdollista että niihin löytyy vielä tehokkaampia menetelmiä. Itse automatisoitavien testien täytyy olla tärkeitä, jotta niiden automatisointiin kannattaa käyttää aikaa. (Pettichord 2001, hakupäivä 21.1.2013.)

Automatisoimalla toisarvoisia ja yksitoikkoisia tehtäviä, esimerkiksi testejä, jotka vaativat paljon samanlaisia syötteitä ja toistoja, voidaan saavuttaa suurempi tarkkuus ja parantaa testaajan työintoa. Näin kyseinen testaaja vapautetaan suunnittelemaan parempia testitapauksia. (Fewster & Graham 1999, 9.)

Automatisoinnilla saavutetaan myös testien yhdenmukaisuus. Tästä on etua, jos halutaan hyödyntää standardeja. Yhdenmukaisuus auttaa myös testauksen suorittamista eri ympäristöissä valmiiden testitapausten avulla, eli testit on helpommin siirrettävissä toisille alustoille. Yhdenmukaisuudella saadaan myös ennustettavuutta: Automatisoitujen testitapausten suorittamiseen kuluvaa aikaa voidaan arvioida aikaisempien suoritusten perusteella. Tämän ansiosta koko testausprosessiin kuluvaa aikaa on helpompi ennustaa. (Fewster ym. 1999, 9-10, 346.)

Automatisoiduilla testeillä on tietty elinkaari, jonka aikana niiden tulisi maksaa itsensä takaisin. Jotta tämä toteutuisi, täytyy tietää millainen testi on hyödyllinen pitkään ja mitkä asiat voivat johtaa sen hylkäämiseen. Hyviä automatisoitavia kohteita ovat esimerkiksi regressiotestaus ja kuormitustestaus, sillä niitä ajetaan usein ja lähes muuttumattomina. (Marick 2013, hakupäivä 21.1.2013.)

Onnistunut automatisointi voi vaikuttaa aikatauluun, budjettiin ja testausmahdollisuuksien laajenemiseen: Jos testit tuottavat tuloksia nopeammin ja edullisemmin, vapautuu resursseja muualle. Hyvä automatisointi siis pienentää kehityksen ja testauksen kustannuksia, lyhentää läpimenoaikoja, parantaa työmäärien hallintaa testikierroksen aikana, antaa etenemiseen näkyvyyttä ja antaa mahdollisuuden muuttaa suunnitelmia markkinoiden tarpeita vastaaviksi. Testauksen luotettavuus lisääntyy, kun testauksen julkaisuista saadaan nopeammin palaute.

Testejä voidaan yhdistellä tarvittaessa, sekä voidaan ajaa yhtäaikaisesti useampia testejä. (Pyhäjärvi & Pöyhönen 2004, 16.)

4.4 Automatisoinnin heikkoudet

Kun testausta automatisoidaan törmätään usein yllättäviin ongelmiin. Tällaiset ongelmat, jotka tulevat täytenä yllätyksenä niin järjestelmän kehittäjille kuin käyttäjille, ovat usein kaikkein hankalimpia käsitellä. Automatisointia ei voida pitää oletuksena parhaana ratkaisuna testaukseen, vaan hyvin suunniteltuna ja toteutettuna se tehostaa testaamista monella alueella. Automatisointi ei silti tarjoa ratkaisua kaikkiin testauksessa ilmeneviin ongelmiin. Epärealistiset odotukset ovat lienee yksi suurimmista syistä, jotka johtavat testauksen automatisoinnin epäonnistumiseen. (Fewster ym. 1999, 10.)

Testauksen automatisoiminen on kallis projekti, jonka riskinä on toteutumaton hyöty. On siis tärkeää, että projektin tavoitteet ovat alusta lähtien mahdollisimmat realistiset. Testauksen automatisoiminen on aikaa vievää varsinkin aluksi, jolloin valitaan menetelmät ja työkalut automatisoinnin toteuttamiseksi, sekä koulutetaan työkalun käyttö. Lisäkustannuksia voi myös aiheutua silloin, kun merkittäviä virheitä löytyy vielä ohjelmistokehityksen loppupäästä, jolloin aikaa niiden korjaamisen on vähemmän. (Pyhäjärvi ym. 2004, 17.)

Usein kuvitellaan että automatisoitu testaus löytää paljon uusia virheitä. Tämä on kuitenkin väärin. Suurin osa virheistä löydetään ensimmäisellä testauskerralla. Tämän jälkeen löytyvät virheet ovat usein virheitä, jotka johtuvat ohjelman korjauksesta, eli automatisoinnin hyöty on tällöin vähäinen. Vaikka testaus menisi läpi ilman yhtään virhettä, ei saisi syntyä väärää turvallisuuden tunnetta. Ei ole vielä takeita siitä, että virheitä ei olisi olemassa: on mahdollista, että automatisoitu testitapaus on itsessään jollain tavalla virheellinen, jolloin siitä saatava tulos on myös virheellinen. (Fewster ym. 1999, 10.)

Automatisoinnin yhteydessä testien lukumäärä ja sisältö voivat kasvaa nopeasti, joka lisää työmäärää ohjelmiston muutosten yhteydessä sekä aikaa, joka kuluu testien suorittamiseen. Työmäärän kasvaessa saattaa esimerkiksi dokumentointi jäädä toisarvoiseksi, joka voi johtaa myöhemmin ongelmiin. Senkin jälkeen kun testauksen automatisointi on saatu onnistumaan, voi sen ylläpitäminen osoittautua haasteelliseksi, kun uusia versioita ohjelmistoista otetaan käyttöön.

Usein myös testit vaativat päivityksen ennen kuin ne voidaan ottaa uudelleen käyttöön. Nykyisistä testeistä voi tulla turhia tai ne saattavat korvautua uusilla testeillä. Nämä muutokset taas voivat johtaa ylläpitokustannusten kasvuun. (Fewster ym. 1999, 191-198.)

4.5 Erilaisia tapoja automatisoida testausta

4.5.1 Makrojen nauhoitus ja toisto

Alkeellisin tapa automatisoida testausta on nauhoittaa testaajan hiirellä ja näppäimistöllä tekemät toimet, jonka jälkeen ne voidaan toistaa tarvittaessa. Jos testattava ohjelma toimii Windows- tai Mac-ympäristössä, niin makrojen nauhoitus ja toisto on melko yksinkertaista. Mac-ympäristössä käytettävissä on QuickKeys-ohjelma ja Windows-ympäristössä sitä vastoin Macro Magic. On olemassa useita nauhoitus ja toisto -ohjelmia, joten testaajan on mahdollista valita itselleen tarpeitaan parhaiten vastaava ohjelma. Ohjelmat ovat eräänlaisia ajureita. Niiden avulla testattavaa ohjelmaa voidaan hallita ja käyttää. (Patton 2001, 228.)

Makrojen avulla on siis mahdollista automatisoida testausta. Vaikka makrojen avulla on helpompaa ja nopeampaa uudelleen testata ohjelmaa, ne eivät ole täydellisiä. Suurin ongelma on tulosten varmentaminen: makrojen avulla ei voida testata, että ohjelma toimii, kuten sen kuuluisi toimia. Toinen ongelma on toistojen nopeus. Vaikka makrojen toiston nopeutta voidaan säätää, niin se ei aina riitä pitämään makroja synkronoituna. Huolimatta näistä ongelmista makrojen nauhoitus ja toisto on yksinkertainen ja helppo tapa testaajille aloittaa testauksen automatisointi. (Patton 2001, 229.)

4.5.2 Ohjelmoitavat makrot

Ohjelmoitavat makrot ovat askel eteenpäin pelkästä nauhoitus/toisto-toiminnosta. Ohjelmoitavia makroja luodaan antamalla makroja käsittelevälle ohjelmalle yksinkertaisia ohjeita valitulla ohjelmointikielellä sen sijaan, että nauhoitettaisiin kaikki käyttäjän toimet tämän suorittaessa testejä ensimmäistä kertaa. Testausohjelmasta riippuen käyttäjän ei välttämättä tarvitse edes kirjoittaa komentoja, vaan haluttuja toimintoja voidaan valita esimerkiksi pudotusvalikosta, jolloin ohjelma luo lauseet ohjelmointikielellä testaajan puolesta. (Patton 2001, 230.)

Ohjelmoitavilla makroilla voidaan myös välttää ajoitukseen liittyviä ongelmia, joita liittyy nauhoitus/toisto toiminnon käyttöön: Ohjelmoitavilla makroilla voidaan antaa testille käsky odottaa, kunnes edellinen toiminto on suoritettu loppuun, sen sijaan että odotetaan X sekuntia ennen seuraavaan toimintoon siirtymistä. (Patton 2001, 231.)

Ohjelmoitavilla makroilla pääsee jo pitkälle testauksen automatisoinnissa, kun käytössä on yksinkertainen ohjelmointikieli makrojen tekemiseen, valmiita komentoja annettavaksi testausohjelmalle sekä mahdollisuus hienosäätää makroja tarvittaessa. Jo pelkästään näillä asioilla on mahdollista säästää paljon aikaa testauksessa. (Patton 2001, 231.)

Ohjelmoitaviin makroihin liittyy kuitenkin tiettyjä rajoitteita. Ensinnäkin ohjelmoitavat makrot ovat erittäin suoraviivaisia, ne voivat ainoastaan tehdä silmukoita (loop) ja toistaa itseään (repeat). Perinteisissä ohjelmointikielissä olevia muuttujia ja päättelylausekkeita ei voida toteuttaa ohjelmoitaviin makroihin. Ohjelmoitavat makrot eivät osaa automaattisesti tarkistaa ja tulkita tuloksia, vaan siihen tarvitaan käyttäjää tai kehittyneempää testausohjelmistoa. (Patton 2001, 231.)

4.5.3 Täysin ohjelmoitavat automaattiset testaustyökalut

Täysin ohjelmoitava automaattinen testaustyökalu sisältää täysin kehittyneen ohjelmointikielen, makrokomennot, joilla voidaan testata ohjelmaa sekä lisäkapasiteettia varmentamiseen. Ohjelmoitavat testaustyökalut mahdollistavat testaajille tehokkaiden testien ohjelmoimisen. Täysin ohjelmoitavat automaattiset testaustyökalut saattavat olla liian kalliita yksityiselle käyttäjlle, koska ne on tarkoitettu lähinnä yrityksille. (Patton 2001, 232.)

Kaikkien komentojen ohjelmointi mahdollistaa joustavuuden testien ohjelmoimiseen sekä tekee testeistä paljon luotettavampia. Tulosten varmentamisen mahdollisuus on kuitenkin kaikkein tärkein ominaisuus täysin ohjelmoitavissa testaustyökaluissa. Niiden avulla voidaan testata, että ohjelma toimii, kuten sen kuuluu toimia. Se voidaan varmistaa useammalla eri tavalla:

Ensimmäisellä kerralla testejä suoritettaessa testaaja voi ottaa kuvankaappauksen tuloksista, jotka tiedetään olevan oikeita. Ohjelmoitu testaustyökalu voi vertailla myöhempien testien tuloksia tallennettujen tulosten kanssa; jos jotain on erilailla tai jotain odottamatonta on tapahtunut, voi

työkalu merkitä virheen. Kuvankaappauksen sijaan testaaja voi määritellä tietyt kontrolliarvot, johon testattavan ohjelman tulisi päätyä. Automaatiotyökalun avulla se onnistuu yksinkertaisesti testattavassa ohjelmassa. Jos ohjelma tallentaa tulokset tiedostoon, niin automaatiotyökalu voi verrata sitä tiedostoon, jossa tuloksen tiedetään olevan oikea. Samaa tekniikkaa voidaan käyttää, vaikka tiedosto lähetetään verkossa jonnekin muualle. (Patton 2001, 233.)

4.5.4 Satunnaistestaus

Yksi automaatiotyökalun muoto on testiapina (test monkey), jonka tarkoituksena on simuloida käyttäjien toimenpiteitä. Termi “testiapina” viittaa ajatukseen, että olisi tilastollisesti mahdollista, että miljoona apinaa näppäilemässä miljoonalla näppäimistöllä miljoonassa vuodessa voisi saada aikaan Shakespearen näytelmän tai muun vastaavan teoksen. (Patton 2001, 234.)

Kun ohjelma julkaistaan yleiseen käyttöön, voi sen parissa olla tuhansia tai parhaimmillaan miljoonia käyttäjiä näppäilemässä. Huolimatta kaikesta tehdystä testauksesta on todennäköistä, että jokin virhe on päässyt läpi lopulliseen tuotteeseen ja joku näistä miljoonista käyttäjistä tulee törmäämään siihen. Testiapinoiden tavoitteena on simuloida näiden miljoonien käyttäjien satunnaisia toimia ja löytää nämä virheet ennen ohjelman julkaisua. (Patton 2001, 235.)

Dumb Monkeys

Helpoin ja suoraviivaisin testiapina on niin kutsuttu “Dumb Monkey”. Dumb Monkey ei tiedä mitään testattavasta ohjelmasta, vaan sen tehtävä on klikkailla ja näppäillä kaikkea satunnaisesti ilman mitään logiikkaa. Dumb Monkeyn toimintaan ei liity minkäänlaista varmentamista, vaan se jatkaa toimintaansa, kunnes se saa sille annetun silmukan päätökseen tai kaataa testattavan ohjelman. (Patton 2001, 235.)

Vaikka saattaisi vaikuttaa epätodennäköiseltä, että yksinkertaisella klikkailulla ja näppäilyllä löytyisi virheitä testattavasta ohjelmasta, sitä kuitenkin tapahtuu. Kun aikaa on tarpeeksi, voi testiapina käydä läpi paljonkin sellaisia tapauksia, joita testaajat eivät ole ottaneet huomioon omissa testeissään. Testiapinan voi antaa pyöriä myös pitkiä aikoja, jolloin ilmenee virheitä, joita ei normaalissa testauksessa tulisi vastaan, kuten esimerkiksi muistivuotoja. Valitettavasti myös

nykyisin saattaa törmätä ohjelmaan, joka muuttuu epävakaammaksi mitä kauemmin sitä käyttää. Tämäkin ongelma olisi voitu löytää ennen julkaisua testiapinan avulla. (Patton 2001, 235-236.)

Semi-Smart Monkeys

Seuraava testiapinan muoto on niin kutsuttu “Semi-Smart Monkey”, joka on kehittyneempi versio edeltäjästään. Semi-Smart Monkeyn uusia ominaisuuksia ovat esimerkiksi testiapinan toiminnan kirjaaminen tiedostoon. Tämän avulla testaajan on helpompi paikantaa apinan aikaansaama virhe ja lähettää raportti eteenpäin ohjelmoijalle. Tämän testiapinan voi myös rajoittaa toimimaan ainoastaan annetun ohjelman sisällä sulkematta sitä, siinä missä Dumb Monkey saattoi satunnaisten klikkaustensa kautta sulkea ohjelman ja lopettaa testaamisen siihen. Semi-Smart Monkeyn voi myös komentaa aloittamaan testauksen alusta, jos se sattuu löytämään virheen ja kaatamaan ohjelman. Tämä mahdollistaa kyseisen testiapinan käyttämisen tehokkaasti myös yöllä tai muuna aikana, kun ihminen ei ole valvomassa toimintaa. (Patton 2001, 236.)

Smart Monkey

Älykkäin testiapina on niin kutsuttu “Smart Monkey”. Smart Monkey on tietoinen ympäristöstään. Se ei vain paukuta näppäimistöä satunnaisesti vaan tarkoituksellisesti. Smart Monkey tietää, mitä se voi tehdä, mihin se voi mennä, missä se on ollut ja osaa tulkita näkemäänsä. Testiapina on myös tietoinen ohjelman tilamuunnoksista. Smart Monkey ei yritä ainoastaan kaataa ohjelmaa, vaan se voi tutkia tietoa, tarkistaa tulokset sekä etsiä virheitä niistä. Jos testitapaukset ovat ohjelmoituja, niin Smart Monkey voi satunnaisesti suorittaa niitä, etsiä niistä virheitä sekä kirjata tulokset logiin. (Patton 2001, 237-238.)

Smart Monkey osaa tunnistaa testattavan ohjelman tilan ja valita toiminnan sen mukaan samalla simuloiden oikean käyttäjän toimenpiteitä. Toimenpiteen jälkeen Smart Monkey osaa tulkita tuloksia ja tunnistaa ohjelman tilan muutoksen, jonka jälkeen se jatkaa testiä muuttuneen tilan pohjalta jälleen eteenpäin. Smart Monkey siis mahdollistaa oikeiden käyttäjien simuloinnin ja lyhentää oikeilta käyttäjiltä testaukseen kuluvan ajan tuhansista tunneista parhaimmillaan alle kymmeneen tuntiin. (Patton 2001, 238.)

5 AVOIN LÄHDEKOODI

Avoimen lähdekoodin (Open Source) ohjelmalla tarkoitetaan tietokoneohjelmaa, josta on saatavilla itse toimivan ohjelman lisäksi ohjelman lähdekoodi, jota saa kuka tahansa muuttaa, kopioida ja jakaa vapaasti. Avoimen lähdekoodin ohjelmat ovat usein yksittäisten ihmisten tai julkisten organisaatioiden, kuten yliopistojen, tai yritysten kehittämiä. Yritysten motivaationa toimii luoda ohjelmistoalustoja omille laitteilleen tai ohjelmilleen. Eräs kuuluisimmista avoimeen lähdekoodiin perustuvista ohjelmistoista tulee Suomesta: Linux-käyttöjärjestelmän ydin, eli Linuxin "kernel". Avoimen lähdekoodin ohjelmistot ovat saaneet paljon maailmanlaajuista julkisuutta ja ne ovatkin nykypäivänä yksi informaatioteknologian johtavista suuntauksista. (Grönroos 2003, hakupäivä 23.1.2013.)

Open Source Initiativen määritelmän mukaan avoimen lähdekoodin ohjelman tulee täyttää seuraavat vaatimukset:

1.Ohjelman täytyy olla vapaasti levitettävissä ja välitettävissä.
2.Lähdekoodin täytyy tulla ohjelman mukana tai olla vapaasti saatavissa.
3.Myös johdettujen teosten luominen ja levitys pitää sallia.
4.Lisenssi voi rajoittaa muokatun lähdekoodin levittämistä vain siinä tapauksessa, että lisenssi sallii korjaustiedostojen ja niiden lähdekoodin levittämisen. Voidaan myös vaatia, ettei johdettua teosta levitetä samalla nimellä tai versionumerolla kuin lähtöteosta.
5.Yksilöitä tai ihmisryhmiä ei saa asettaa eriarvoiseen asemaan.
6.Käyttötarkoituksia ei saa rajoittaa.
7.Kaikilla ohjelman käsiinsä saaneilla on samat oikeudet.
8.Lisenssi ei saa olla riippuvainen laajemmasta ohjelmistokokonaisuudesta, jonka osana ohjelmaa levitetään, vaan ohjelmaan liittyvät oikeudet säilyvät, vaikka se irrotettaisiin kokonaisuudesta.
9.Lisenssi ei voi asettaa ehtoja muille ohjelmille. Ohjelmaa saa levittää myös yhdessä sellaisten ohjelmien kanssa, joiden lähdekoodi ei ole avointa.
10.Lisenssin sisällön pitää olla riippumaton teknisestä toteutuksesta. Oikeuksiin ei saa liittää varaumia jakelutavan tai käyttöliittymän varjolla. (Coss 2013, hakupäivä 29.5.2013.)

6 TESTAUSTYÖKALUJA

6.1 Nauhoitustyökaluja

Marathon

Marathon on testaustyökalu Java/Java Swing -ohjelmointikielille. Sen avulla voi nauhoittaa, toistaa sekä muokata testitapauksia sovelluksille, jotka sisältävät graafisen käyttöliittymän. Marathon nauhoittaa testit helposti luettavaan ja muokattavaan formaattiin käyttäen Jython- tai JRuby-kääntäjää riippuen, mitä käyttäjä valitsee luodessaan projektin. Marathonin avulla testit voidaan suorittaa graafisen käyttöliittymän kautta valvotusti tai niin kutsutussa batch-moodissa ilman käyttäjän valvontaa. Jalian Systems tarjoaa myös maksullista MarathonITE-versiota, johon kuuluu lisäominaisuuksia ja tuki kaupalliseen käyttöön. Marathon toimii Windows-, Mac- ja Linux-alustoilla. (GitHub 2013, hakupäivä 25.4.2013.)

iMacros

iMacros on vuodesta 2001 alkaen kehitetty työkalu, joka tarjoaa nauhoitus/toisto-toiminnon Chrome-, Firefox- ja Internet Explorer -selainten lisäosana. iMacros soveltuu www-sivujen regressiotestaukseen ja sillä voi esimerkiksi mitata sivuston vasteaikoja eri tilanteissa. iMacrosin erikoisuutena on sen tarjoama mahdollisuus jakaa sillä nauhoitettuja makroja muiden käyttäjien kanssa. iMacros selainten lisäosana on avoimen lähdekoodin ohjelma, mutta sen kehittäjä iOpus tarjoaa myös kaupallista versiota laajemmilla ominaisuuksilla ammattilaiskäyttöön. iMacros toimii Windows-, Linux- ja Mac-alustoilla. (iMacros 2012, hakupäivä 29.4.2013.)

Selenium IDE

Selenium IDE on ohjelmointiympäristö Selenium skripteille. Selenium IDE toimii ainoastaan Mozilla Firefox -selaimen lisäosana (extension), jonka avulla voi nauhoittaa, muokata ja debugata testejä. Selenium IDE sisältää Selenium Coren, joka mahdollistaa testien nauhoituksen ja toiston juuri siinä ympäristössä, missä testejä on tarkoitus ajaa. Muita ominaisuuksia ovat muun muassa

mahdollisuus tallentaa testit HTML-, Ruby- tai mihin tahansa muuhun muotoon, sekä itse Selenium IDE:n kustomointi lisäosia (plugin) käyttäen. (Selenium HQ 2013, hakupäivä 15.4.2013.)

6.2 Ohjelmistokehyksiä

JUnit

JUnit on yksikkötestaukseen tarkoitettu ohjelmistokehys (framework), josta on muodostunut niin sanottu de-facto standardi Java-ohjelmointikielen testaukseen. JUnitin suurimpia vahvuuksia on sen yksinkertaisuus ja sen avulla saavutettu toimintavarmuus. JUnit perustuu avoimeen lähdekoodiin, joten sille löytyy myös runsaasti kolmansien osapuolien tekemiä laajennuksia, joilla voi tarvittaessa lisätä testien kattavuutta. JUnit toimii Windows-, Linux- ja Mac-alustoilla. (Methods and Tools 2013, hakupäivä 22.4.2013.)

MActor

MActor on laajennettavissa oleva työkalu erityisesti toiminnalliseen integraatiotestaukseen. MActor tukee jo ilman lisäosia jo valmiiksi integraatiotestaukseen sopivaa teknologiaa ja kieliä, kuten JMS ja XML (HTTP, SOAP, TIBCO Rv sekä IBM MQ kieliä käyttäen). MActorin avulla voidaan testata monimutkaisia integraatioskenaarioita, joihin kuuluu kommunikointia useiden järjestelmien välillä, erilaisia integraatiotekniikoita sekä suuria määriä testitapauksia. (Sourceforge 2008, hakupäivä 25.4.2013.)

FitNesse

FitNesse on wiki-periaatteella toimiva avoimen lähdekoodin testaustyökalu hyväksymistestaukseen. FitNesse on käytännössä netissä toimiva wiki-palvelu, jonne käyttäjät voivat ladata omia testitapauksiaan, joita toiset käyttäjät sitten saavat käyttöönsä FitNessen kautta. FitNessen voi laittaa toimimaan automaattisesti taustalle tai käyttäjän haluamana aikana. FitNesseä suositellaan käytettäväksi esimerkiksi JUnitin tai NUnitin kanssa parhaan tuloksen aikaansaamiseksi. Tuettuja testauskieliä ovat Java, Python ja C#. Tukea voi laajentaa lisäosien avulla. FitNesse toimii Windows- ja Linux-alustoilla. (FitNesse 2012, hakupäivä 8.5.2013.)

6.3 Suorituskykytestaustyökaluja

JMeter

JMeter on täysin Javalla toteutettu avoimen lähdekoodin testaustyökalu, jolla on mahdollista ajaa kuormitustestejä palvelimille, tietoverkolle tai yksittäisille objekteille. JMeterin avulla käyttäjä voi testata staattisia tai dynaamisia resursseja, kuten tiedostoja, tietokantoja, Java-objekteja tai erilaisia palvelimia (FTP, Web, SOAP, JMS, SMTP ja LDAP). JMeter antaa tulokset myös graafisessa muodossa, sekä mahdollistaa tulosten tallentamisen ja myöhemmän tutkimisen myös ilman Internet-yhteyttä. JMeter on myös laajennettavissa lisäosien avulla. JMeter toimii Windows- ja Linux-alustoilla. (Apache Software Foundation 2013, hakupäivä 29.4.2013.)

nGrinder

Avoimen lähdekoodin ohjelmistoalusta nGrinderin käyttötarkoituksena on pääasiallisesti verkkosivujen kuormitustestaus. Se koostuu kahdesta osasta: Controllerista ja agentista. Controller on selaimen kautta käytettävä hallintapaneeli, jonka kautta käyttäjä voi luoda ja ajaa testejä. Agent on puolestaan palvelimelle asennettava Java-sovellus, jonka tehtävä on palvelimesta riippuen joko tarkkailla kuormaa tai luoda kuormaa simuloimalla käyttäjiä. Testausskriptien luomiseen ja useiden testien ajamiseen samanaikaisesti nGrinder käyttää Jythonia. Toimiakseen nGrinder vaatii Oracle JDK 1.6:n tai uudemman, sekä Windows-, Mac- tai Linux-alustan. (nhnopensource 2013, hakupäivä 23.5.2013.)

6.4 Muita testaustyökaluja

Cucumber

Cucumber on kirjoitettu Ruby-ohjelmointikielellä ja sitä voidaan käyttää testaamaan esimerkiksi Ruby-, Java-, C#- ja Python-kielisiä ohjelmia. Se on testaustyökalu, jonka ideana on ottaa vastaan normaalilla kielellä kirjoitettuja tapauksia (skenaarioita) ja tehdä niistä testitapauksia. Tämän jälkeen käyttäjä kirjoittaa koodia, ajaa testiä uudestaan ja uudestaan, kunnes käyttäjä on onnistunut luomaan koodin, joka läpäisee kaikki testin vaiheet. Siis Cucumberia käytettäessä itse koodi kirjoitetaan läpäisemään aiemmin kirjoitettu testi. Cucumberia voidaan ajatella siis

enemmän käyttäytymislähtöisen kehityksen (BDD, behaviour-driven development) apuvälineenä, kuin pelkkänä testaustyökaluna. (GitHub 2013b, hakupäivä 26.4.2013.)

Cucumber ottaa testitapaukset vastaan niin kutsutulla Gherkin -kielellä, jonka tarkoitus on muistuttaa normaalia kirjoitettua kieltä (KUVIO 5). Gherkin ymmärtää syötteitä 37 eri kielellä, joista yksi on englanti. (GitHub 2013c, hakupäivä 26.4.2013.)

```
Feature: Serve coffee
  In order to earn money
  Customers should be able to
  buy coffee at all times

  Scenario: Buy last coffee
    Given there are 1 coffees left in the machine
    And I have deposited 1$
    When I press the coffee button
    Then I should be served a coffee
```

KUVIO 5. Gherkin skenario englanniksi (GitHub 2013c, hakupäivä 8.5.2013.)

7 JOHTOPÄÄTÖKSET JA POHDINTA

Opinnäytetyön tavoitteena oli tehdä tehdä tiivis teoriapainotteinen työ testauksen automatisoinnista. Tarkoituksena oli myös esitellä muutamia erilaisia ohjelmia, joiden avulla testausta voidaan automatisoida. Työn lähtökohtana oli ottaa selvää automatisoinnin kohteet ja tilanteet milloin kannattaa automatisoida. Opinnäytetyössä kävimme läpi testauksen yleisellä tasolla, jonka jälkeen siirryimme käsittelemään enemmän testauksen automatisointia. Pohjatietoa testauksesta meillä oli alunperin niukasti, vaikka muutamilla opintojaksoilla aihetta oltiin hieman sivuttu.

Testauksesta ja sen automatisoinnista löytyi runsaasti tietoa kirjallisuudesta ja internetistä. Meidän tehtäväksemme jäi suodattaa aiheemme kannalta oleellinen tieto ja koostaa ne loogiseksi kokonaisuudeksi. Työn loppupuolella yllättävänä seikkana nousi esille testaustyökalujen laaja valikoima. Lähempi tarkastelu kuitenkin paljasti, että monet näistä avoimen lähdekoodin projekteista olivat auttamatta vanhentuneita. Monessa työkalussa oli mainittu uusimpana tuettuna Windows-versiona esimerkiksi Windows NT tai Windows 2000, sekä itse työkalua oltiin päivitetty viimeksi edellisen vuosikymmenen puolella.

Tapaa lajitella testaustyökalut pohdimme pitkään ja lopulta päädyimme kysymään asiasta myös ohjelmointia työkseen tekevältä tuttavaltamme. Tämän neuvonpidon tuloksena päädyimme nykyiseen kolmijakoon. Cucumber edusti hieman erilaista lähestymistapaa, mutta halusimme sen myös mukaan ohjelmiin sen suosion takia, joten päädyimme laittamaan sen otsikon “Muut” alle.

Opinnäytetyön tavoitteet täyttyivät mielestämme melko hyvin. Onnistuimme käsittelemään testausta laajasti ja loogisessa järjestyksessä ottaen huomioon useamman näkökulman. Tavoitteena oli myös antaa lukijalle yleiskäsitys testauksesta ja olemme tyytyväisiä lopputulokseen. Lukijan on mahdollista sisäistää testauksen perusperiaatteet ja käsitteet työmme avulla.

Eniten parannettavaa opinnäytetyötä kirjoittaessa olisi ollut aikataulun noudattamisessa. Aikataulu myöhästyi alkuperäisestä suunnitelmasta noin kahdella kuukaudella. Osasyy tähän oli

toisen jäsenen alkanut työsuhde, joka vaikeutti hieman yhteisen ajan löytämistä. Aikaa vievin yksittäinen osa-alue oli testaustyökalujen lajittelu tarkoituksenmukaisiin kokonaisuuksiin. Kaiken kaikkiaan opinnäytetyön kirjoitus oli kuitenkin opettavainen ja mielenkiintoinen prosessi.

LÄHTEET

Apache Software Foundation 2013. JMeter. Hakupäivä 29.4.2013, http://jmeter.apache.org/

Chakraborty, P. 2009. What is AD Hoc Testing? Hakupäivä 29.1.2013, http://ezinearticles.com/?What-is-AD-Hoc-Testing?&id=2047543

Confuse 2002. V-mallin mukainen testausmenetelmä. Hakupäivä 24.1.2013, http://www.soberit.hut.fi/T-76.115/01-02/palautukset/groups/Confuse/lu/docs/vmalli/vmalli.pdf

Coss 2013. Avoin lähdekoodi. Hakupäivä 29.5.2013, http://coss.fi/avoimuus/avoin-lahdekoodi/

Dustin, E. Garrett, T & Gauf, B. 2009. Implementing automated software testing. New York: John Wiley & Sons.

Dustin, E. Rashka, J. & Paul, J. 2008. Automated Software Testing. Introduction, Management and Performance. Addison-Wesley.

Fewster, M. & Graham, D. 1999. Software Test Automation. Effective use of test execution tools. New York: ACM Press.

FitNesse 2012. Hakupäivä 8.5.2013, http://fitnesse.org/

Gelperin, D. & Hetzel, B. 1988. The growth of software testing. Hakupäivä 9.1.2013, http://staff.unak.is/andy/MScTestingMaintenance/Homeworks/STMHeima6GrowthTesting.pdf

GitHub 2013a. Jalian Systems/Marathon. Hakupäivä 25.4.2013, https://github.com/jalian-systems/Marathon

GitHub 2013b. Cucumber. Hakupäivä 26.4.2013, https://github.com/cucumber/cucumber/wiki

GitHub 2013c. Cucumber/Gherkin. Hakupäivä 26.4.2013,
https://github.com/cucumber/cucumber/wiki/Gherkin

Grönroos, M. 2003. Avoimen lähdekoodin ohjelmistojen lokaisoinnin tila. Hakupäivä 23.1.2013,
http://www.lokalisointi.org/files/floss-lokalisoinnin-tila-2003.pdf

Haikala, I. & Märijärvi, J. 2006. Ohjelmistotuotanto. 11. uudistettu painos. Helsinki: Talentum

Hendrickson, E. 1998. The Differences Between Test Automation Success
and Failure. Quality Tree Consulting. Hakupäivä 21.1.2013, http://testobsessed.com/wp-content/uploads/2011/04/dbtasaf.pdf

Holopainen, J. 2005. Regressiotestaus ja testien valintatekniikat. Hakupäivä 12.2.2013,
http://opiskelu.businesscollege.fi/burot/2012syksy/2_jakso/2_Tietoj%C3%A4rjestelm%C3%A4ty%C3%B6_%281%292_ov_II/_2_6_5_Regressiotestaus_ja_testien_valintatekniikat.pdf

iMacros 2012. Introducing iMacros. Hakupäivä 29.4.2013,
http://wiki.imacros.net/Introducing_iMacros

Kaner, C. Bach, J. & Pettichord, B. 2002. Lessons learned in software testing: a context-driven approach. New York: Wiley Computer Publishing

Kaner, C. Bach, J. & Pettichord, B. 2011. Lessons learned in software testing : a context-driven approach. New York: John Wiley & Sons.

Luo L. 2013. Software Testing Techniques. Hakupäivä 9.1.2013,
http://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf

Marick, B. 2013. When Should a Test Be Automated? Hakupäivä
21.1.2013, http://www.exampler.com/testing-com/writings/automate.pdf

Methods & Tools 2013. JUnit. Hakupäivä 22.4.2013,
http://www.methodsandtools.com/tools/tools.php?junit

Muzak Studyzone. 2012. White and black box testing techniques. Hakupäivä 13.3.2013, http://muzakstudyzone.blogspot.fi/2012/12/white-and-black-box-testing-techniques.html

Myers, G. Sandler, C. Badgett, T. & Thomas, T. 1979. The art of software testing. New York: John Wiley & Sons.

nhnopensource 2013. nGrinder. Hakupäivä 23.5.2013, http://www.nhnopensource.org/ngrinder/

Patton R. 2001. Software Testing. Indianapolis: Sams

Pettichord, B. 2001. Success with Test Automation. Hakupäivä 21.1.2013, http://www.prismnet.com/~wazmo/succpap.htm

Pohjolainen, P. 2003. Ohjelmiston testauksen automatisointi. Hakupäivä 20.1.2013, http://www.cs.uku.fi/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf

Pyhäjärvi, M. & Pöyhönen, E. 2004. Testausvälineet ja testauksen automatisointi. Hakupäivä 21.1.2013, http://users.jyu.fi/~kolli/testaus2006/materiaali/8_TestausvalineetJaTestauksenAutomatisointi_v0_1.ppt

SeleniumHQ 2013. Selenium IDE Plugins. Hakupäivä 15.4.2013, http://docs.seleniumhq.org/projects/ide/

Software Testing Fundamentals 2013a. Black Box Testing. Hakupäivä 30.1.2013, http://softwaretestingfundamentals.com/black-box-testing/

Software Testing Fundamentals 2013b. White Box Testing. Hakupäivä 4.2.2013, http://softwaretestingfundamentals.com/white-box-testing/

Software Testing Fundamentals 2013c. Gray Box Testing. Hakupäivä 4.2.2013, http://softwaretestingfundamentals.com/gray-box-testing/

Software Testing Mentor 2013a. Static Testing. Hakupäivä 5.2.2013, http://www.softwaretestingmentor.com/types-of-testing/static-testing.php

Software Testing Mentor 2013b. Dynamic Testing. Hakupäivä 5.2.2013, http://www.softwaretestingmentor.com/types-of-testing/dynamic-testing.php

Sourceforge 2008. MActor. Hakupäivä 25.4.2013, http://mactor.sourceforge.net/

Tampereen teknillinen yliopisto 2011. Ohjelmistojen testaus. Hakupäivä 4.2.2013, http://www.cs.tut.fi/~testaus/s2011/luennot/OHJ-3060_2011_110-170.pdf

Waterfall Model 2013. V Model. Hakupäivä 4.2.2013, http://www.waterfall-model.com/v-model-waterfall-model/

Zambelich, K. 1998. Totally Data-Driven Automated Testing. A White Paper. Hakupäivä 21.2.2013, http://www.google.fi/url?sa=t&rct=j&q=&esrc=s&frm=1&source=web&cd=1&ved=0CE4QFjAA&url=http%3A%2F%2Fwww.qanc.co.kr%2F4research_0402_download.htm%3Fdata_no%3D70%26name%3DTotally%2520Data-Dr-ven%2520Automated%2520Testing.pdf&ei=kfwlUb2SNOmH4ATssYGgDw&usg=AFQjCNEstBx4wvW-BQcMKAKVes5Oq1_WfQ&sig2=WKobkKIokSJgsWssWkDTZg