Wille Litmanen

AUTOMAATTISEN TOIMINNALLISEN TESTAUKSEN KEHITTÄMINEN

AUTOMAATTISEN TOIMINNALLISEN TESTAUKSEN KEHITTÄMINEN

Wille Litmanen
Opinnäytetyö
Syksy 2010
Tietojenkäsittelynkoulutusohjelma
Oulun seudun ammattikorkeakoulu

Oulun seudun ammattikorkeakoulu

Koulutusohjelma: Tietojenkäsittelynkoulutusohjelma

Tekijä(t): Wille Litmanen

Opinnäytetyön nimi: Automaattisen toiminnallisen testauksen kehittäminen

Työn valmistumisvuosi: 2010

Sivumäärä + liitteet: 41

TIIVISTELMÄ

Tämän opinnäyte työn aihe on automaattisen toiminnallisen testauksen kehittäminen. Opinnäytteen avulla pyrittiin kehittämään MediMaker 7 -ohjelmiston automaattista toiminnallista testausta suunnittelemalla ja toteuttamalla Selenium-pohjaisia testiskriptejä. Testiskripteille suunniteltiin myös testausympäristö käyttäen Hudson-ohjelmistoa, joka mahdollistaa testien ja buildien jatkuvan integroinnin.

Testiskriptien tekemiseen käytettiin Selenium IDE-, Selenium RC-, JUnit- ja Eclipse-työkaluja Tutustun myös Maven- ja Hudson-ohjelmistoihin. Maven on Java-koodin kääntämistä helpottava komentopohjainen ohjelma. Hudson on jatkuvan integroinnin palvelinohjelma, joka pitää huolen, ettei ohjelmakoodeihin pääse virheitä ja, että koodit kääntyvät. Toisin sanoen Hudson testaa koodeja yksikkötesteillä sekä koostaa koodeja jatkuvasti. Opinnäytetyön raportissa kuvataan myös ohjelmistotestauksen sekä automaattisen ohjelmistotestauksen kenttää.

Opinnäytetyöni toiminnallisen osuuden tuloksena syntyi pohja MediMaker 7 -ohjelmiston automaattiselle toiminnalliselle testaukselle Selenium-testiskriptejä käyttäen. Tätä pohjaa Mawell Oy voi lähteä kehittämään haluamaansa suuntaan. Pohja koostuu Selenium-testiskripteistä, dokumentaatiosta sekä karkeahkosta testausympäristöstä, jossa testiskriptejä voidaan ajaa automatisoidusti Hudsonin avulla.

Avainsanat: Automaattinen testaus, Toiminnallinen testaus, Testaustyökalut, Selenium IDE, Selenium RC, Maven, Hudson

Oulu University of Applied Sciences

Degree Programme: Business Information Systems

Author (s): Wille Litmanen

Title of thesis: Development of automated functional testing

Year: 2010

Number or pages + number of appendices: 41

ABSTRACT

The subject of my thesis is development of automated functional testing. The goal is to develop automated functional testing for MediMaker 7. This is done by designing and creating a suitable test environment and creating new Selenium test scripts of MediMaker 7's web user interface.

Eclipse, Selenium IDE, Selenium RC, JUnit are used in creating the Selenium test scripts. Maven is command line based software, which used in compiling the test scripts. Hudson is continuous integration server software, which continuously builds software projects. Hudson also makes sure that any bugs or faults are not found in the Java projects. This is done by running unit tests.

Result of the thesis is a foundation for the automated functional testing of MediMaker 7, which can be used for developing it further. New Selenium test scripts are designed to widen scope of automated functional testing of MediMaker 7. Test scripts can be run automatically without human interaction with help of Hudson.

Keywords: Automated software testing, Functional testing, Testing tools, Selenium IDE, Selenium RC, Maven, Hudson

SANASTOA

Apache Ant

Käännöstyökalu java-ohjelmille. Ant hallitsee kääntämisen, JAR-pakettien luomisen, tiedostojenhallinnan sekä auttaa dokumentoinnissa.

Concurrency testing

Testaus, jossa useat eri käyttäjät testaavat samaa ohjelman osaa tai ominaisuutta suorittaen toiminnot täsmälleen samaan aikaan. Tällä testauksella pyritään löytämään järjestelmän lukittumisen aiheuttajat.

DICOM-kuvat

(The Digital Imaging and Communications in Medicine) DICOM on standardi lääketieteellisen kuvantamiseen käsittelyyn, tulostukseen ja tietojen välittämiseen.

Distributed testing

Tarkoittaa useiden eri automatisoitujen testien suorittamista samanaikaisesti eri työasemilla. ”Hajautettu” termiin liittyy myös samanaikaisesti suoritettavien testien lisäksi se, että työasemat ovat vuorovaikutuksessa keskenään.

De facto -standardi

De facto -standardit ovat laajalti käytettyjä määriteltyjä toteutustapoja, jotka ovat saavuttaneet standardinomaisen aseman ilman virallisen standardin statusta. Ratkaisut perustuvat ainoastaan

	menetelmän kehittäjän valintoihin, eivätkä laajaan eri tavoitteiden huomioimiseen kuten viralliset standardit yleensä.
Jatkuva integrointi	Jatkuvalla integraatiolla tarkoitetaan prosessia, jossa koko ohjelmisto koostetaan ja integroidaan jatkuvasti. Perinteisissä ohjelmistokehitysmalleissa integraatio sijoittuu projektin loppupuolella tehtäväksi kertarypistykseksi.
Pinon korruptoituminen	Pino -tyyppisen muistin korruptoituminen, joka johtaa ohjelman kaatumiseen.
Testauskehys	Runko testien kirjoittamista varten, joka sisältää erilaisia valmiita toimintoja testien ajamiseen ja raportointiin.
Testilähtöinen ohjelmistonkehitys	Ohjelmiston kehitys lähtee liikkeelle testien suunnittelusta ja kirjoittamisesta. Kun testi ovat valmiit, kirjoitetaan vasta sitten itse ohjelmakoodi.
Testiskripti	Sarja komentoja, jotka suoritetaan testattavassa järjestelmässä. Tällä pyritään selvittämään toimiiko järjestelmä kuten sen oletetaan toimivan.
Regressiotestaus	Järjestelmän uudelleentestaamista vanhoilla testeillä. Tällä varmistetaan, että ohjelman vanhat osat toimivat uusista muutoksista huolimatta.

Villi osoitin	Osoitin, joka ei osoita mihinkään ohjelman osaan tai osoitteeseen. Tilanne yleensä tapahtuu kun paikka, johon osoitin osoitti poistetaan tai siirretään.

SISÄLLYS

1 JOHDANTO

Työn toimeksiantaja on Mawell Oy, joka on vuonna 2001 perustettu Informaatio- ja viestintäteknologia-alan yritys. Mawell Oy:n kehitysyksiköt sijaitsevat Oulussa ja Tukholmassa ja pääkonttori sekä myyntiosasto Helsingissä. Asiakkaita Mawell Oy:llä on yli 350 Suomessa, Ruotsissa, Tanskassa, Norjassa, iso-Britaniassa ja Hollannissa. Yrityksellä on 3 tuoteperhettä, jotka ovat

- S7 Omahoito ja sähköinen asiointi
- M7 Kuvantaminen ja multimedia
- X7 Yhteentoimivuus ja yhteistyö.

Lähtökohtana opinnäytteessä on toimeksiantajan testausprosessin kehittäminen M7 Kuvantaminen ja multimedia tuoteperheeseen kuuluvaan MediMaker 7 ohjelmistoon. MediMaker 7 on selainkäyttöliittymäinen Java- ja JavaScript-teknologioihin perustuva ohjelmisto, jonka toiminnallisuuksia ovat DICOM-kuvien, kuviin liittyvän informaation muokkaus ja kuvien liittäminen raportteihin.

Aihetta voidaan pitää tärkeänä, sillä testauksella on suora vaikutus toimeksiantajan tuotteen laatuun. Mitä vähemmän tuotteesta löytyy virheitä, sitä tyytyväisempiä asiakkaat ovat. Yllättäen ilmenneiden virheiden korjaaminen on kallista jälkikäteen, sillä se johtaa aikataulujen venymiseen. Testauksella pyritään myös puuttumaan tähän seikkaan tehostamalla virheiden etsintää, jotta mahdollisimman paljon virheitä löydetään testauksen aikana. Automaattisen toiminnallisen testauksen kehittämisellä pyritään lisäämään tuotteen testaukseen laajuutta ja syvyyttä sekä vapauttamaan työaikaa itse tuotteen kehitykseen.

Opinnäytteen tietoperusta koostuu ohjelmistotestauksesta sekä automaattisesta ohjelmistotestauksesta. Ohjelmistotestausta on lyhyesti mukana työssä, koska ohjelmistotestauksen ymmärtäminen on olennainen osa automaattista

ohjelmistotestausta. Automaattisen ohjelmistotestauksen osalta raportissa on pyritty vastaamaan kysymyksiin:

- Mitä automaattinen ohjelmistotestaus on?
- Mitä eroja manuaalisen ja automaattisen ohjelmisto testauksen välillä on?
- Mihin automaattinen ohjelmistotestaus kykenee?
- Mitkä sen heikkoudet ovat?

Toiminnallinen osa opinnäytetyössä on rajattu koskemaan MediMaker 7:n web-käyttöliittymästä kirjoitettuja Selenium-testiskriptejä.

2 OHJELMISTOTESTAUS

Ohjelmistotestausta on ollut maailmassa vähintäänkin yhtä kauan kun on ollut olemassa ohjelmistojen kehitystä. Maurice Wilkes tekikin vuonna 1949 tärkeän havainnon; ”*It was one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of the stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.* ” Ohjelmistotestaus onkin kehittynyt valtavasti Maurice Wilkesin vuosista ja sen tärkeys on kasvanut entisestään ohjelmistojen kasvaessa.

Testaus on termi, jolla on useita erilaisia määritelmiä. William Hetzel (1988, 4) on kirjassaan määritellyt testauksen prosessiksi, jossa pyritään löytämään vahvistus sille, että ohjelma tai järjestelmä tekee sitä mitä sen on suunniteltu tekevän. Myers Glenford, J. (1979, 6) on määritellyt testauksen toiminnaksi, jossa testattavaa järjestelmään tai sen osaa suoritetaan tarkoituksena löytää siitä virheitä.

Puhekielessä termillä testaus tarkoitetaan lähes mitä tahansa kokeilemista, kuten ”Testataan toimiiko tämä lamppu”. Ohjelmistotestauksessa testaus määritellään perinteisesti suunnitelmalliseksi toiminnaksi, jossa pyritään etsimään virheitä suorittamalla ohjelmaa tai sen osaa. Ilman suunnitelmallisuutta testaus on umpimähkäistä satunnaisten syötteiden kokeilua, joka ei yllä samaan samanlaisiin tuloksiin kuin huolellisesti suunnitellut testaus. Onnistuneen testauksen tuloksena löydetään testattavasta ohjelmasta virheitä (error, mistake, bug), jotka ovat poikkeamia siitä miten ohjelman on tarkoitettu toimivan. Tämä siis tarkoittaa sitä, että on olemassa jonkinlainen määritelmä siitä miten ohjelman tulisi toimia kuten spesifikaatio. Spesifikaatiota tarvitaan, jotta testauksen lopputuloksen oikeellisuus voidaan todeta ja testausta voidaan tehdä. (Haikala, Märijärvi 2006, 284; Haikala ym. 2006, 287.)

Testaus voidaan mieltää toiminnaksi, jossa pyritään tuhoamaan. Voidaan ajatella, että ohjelmistotestausta lähdetään tekemään olettamalla, että ohjelma sisältää virheitä. Tällöin testaus määritellään prosessiksi, jossa ohjelmaa suoritetaan, jotta siitä löydettäisiin virheitä. Testaus on epäonnistunut, jos testauksesta ei löydetä ollenkaan virheitä, sillä hyvin suurella todennäköisyydellä ohjelmassa on virheitä. (Badgett, Myers & Sandler, 2004, 6-7.)

Virheettömien ohjelmien tekeminen on hyvin vaikeaa, ellei jopa mahdotonta. Arvioiden mukaan ohjelmassa on yksi virhe jo muutamaa kymmentä koodiriviä kohden. Pitkään käytössä olleissa ohjelmissa arvioidaan olevan yksi virhe tuhatta koodiriviä kohden. On myös arvioita, joiden mukaan 5 prosenttia ohjelma virheistä ovat sellaisia, ettei niitä koskaan havaita. Tämän selittää, jo yksinkertaisissakin ohjelmissa oleva suuri syöteavaruus ja se ettei virheellisen kohdan suorittaminen ohjelmassa aina johda virheeseen. Virheellisen kohdan suorittaminen voi aiheuttaa järjestelmässä vian, joka voi korjaantua itsestään, jonkin toisen toiminnon seurauksena. (Haikala ym. 2006, 287-288.)

Suuria kysymyksiä ohjelmistotestauksessa on milloin testaus tulisi lopettaa? Silloin kun testausta tehdessä ei enää löydetä ohjelmasta virheitä? Koko ohjelmien kattava ja täydellinen testaus on lähestulkoon mahdottomuus, sillä se vaatisi kaikkien syötteiden ja polkujen testausta. Myös kaikki koodiin tehdyt korjaukset, joilla virheitä pyritään korjaamaan, täytyy testata sillä ne voivat luoda uusia virheitä ohjelmaan. Päätös testauksen lopettamisesta onkin usein kompromissi tuotteessa olevien virheiden aiheuttaminen kustannusten ja markkinoilta myöhästymisen aiheuttaman tuoton menetysten välillä. Testauksen lopettamiselle tulisinkin asettaa päätöskriteeri, jonka täytyttyä testaus voidaan lopettaa. Päätöskriteerejä voisivat olla esimerkiksi kun tarpeeksi virheitä on saatu korjattua eikä kriittisiä virheitä enää löydy tai kaikki testitapaukset (syöttöarvojen joukko) ovat suoritettu niin ettei niiden ajon aikana löytynyt virheitä. Testauksen riittävyyttä voi myös arvioida erilaisilla mittareilla.

- Mutkikkuusmitoilla pyritään päättelemään ohjelmamoduulien monimutkaisuutta. Mutkikkuusmitan avulla voidaan yrittää paikantaa ne moduulit, jotka vaativat eniten testausta.
- Kattavuusmitoilla voidaan yrittää varmistaa testiaineiston kattavuus eli koko ohjelma ja sen eri osat käydään läpi. Kattavuusmitta kertoo siis onko testausta tehty riittävästi.

(Haikala ym. 2006, 293–294; Aaltonen, K. 2009. Ohjelmistotestauksen periaatteet, hakupäivä 24.10.2009.)

3 AUTOMAATTINEN TESTAUS

Automaattinen ohjelmistotestaus voi tarkoittaa useita eri asioita henkilöstä riippuen. Joillekin termi tarkoittaa testilähtöistä ohjelmistokehitystä tai yksikkötestausta, toisille automaattisessa testauksessa käytettävää työkalua, joka nauhoittaa ja toistaa testejä. Termi voi myös tarkoittaa oman testiskriptien kehittämistä käyttäen apuna erilaisia skriptauskieliä kuten Perl, Python tai Ruby. Joillekin termi tarkoittaa vain suorityskykytestausta tai testausta joka keskittyy täysin toiminnalliseen testaukseen. Automaattiselle ohjelmistotestaukselle voi siis löytää monenlaisia määritelmiä. (Dustin, Garrett & Gauf 2009, 3.)

Elfriede Dustin on kirjassaan (1999, 4) määritellyt automaattisen testauksen automaattisen työkalun tuomista mukaan testausprosessiin, jolle kehitetään testiskriptejä. Testiskriptejä ajamalla varmistetaan, että ohjelma läpäisee sille asetetut vaatimukset.

Tässä raportissa automaattisella ohjelmistotestauksella tarkoitetaan ohjelmistotestausta, jota automatisoidaan siihen tarkoitettujen ohjelmien ja skriptien avulla. Tämä tarkoittaa sitä, että tietokoneet testaavat ohjelmistoa testiskriptejä jatkuvasti ajaen ja varmistaen, ettei kehitettävään ohjelmistoon pääse virheitä.

3.1 Automaattisen testauksen vahvuudet

Automaattisen testauksen suurin vahvuus on se, että sillä voidaan tehdä asioita, jota manuaalisella testauksella ei voi tai ei ole järkevää tehdä. Automaattista testausta voi käyttää apuna esimerkiksi sellaisten testien suorittamisessa, joiden suorittaminen manuaalisesti ihmisillä veisi paljon aikaa tai jotka sisältäisivät paljon toistamista. Hyvänä esimerkkinä tällaisesta kohteesta olisi yksikkötestit. (Myers, Glenford J. 1979. 184).

Automaattisella testauksella on muitakin vahvuuksia kuin toistojen suorittaminen. On tilanteita, joita ei kannata edes harkita lähteä tekemään manuaalisesti. Seuraavassa osuudessa esitellään Backin ym. käsitys (2002, 96.) siitä minkälaisissa tilanteissa automaattinen testaus on vahvimmillaan.

Kuormitustestauksella pyritään selvittämään mitä tapahtuu jos järjestelmällä on useita kymmeniä, satoja tai tuhansia käyttäjiä. On helpompaa lähteä selvittämään näitä seikkoja automaattisella testauksella simuloiden.

Suoritustestauksessa halutaan tietää, minkälainen testattavan järjestelmän suorituskyky on erilaisissa tilanteissa. Mittaustuloksia voidaan kerätä automaattisen testauksen avulla. Tuloksia tarkastelemalla voidaan puuttua suorituskyvyn heikkenemiseen ajoissa.

Konfiguraatiotestauksessa tarkoituksena on selvittää miten kehitetty järjestelmä toimii erilaisilla käyttöjärjestelmillä, asetuksilla ja erilaisilla oheislaitteilla. Automaattinen testaus auttaa tässä laajentamalla testauksen kattavuutta sillä kone testaa erilaisia konfiguraatioita eri käyttöjärjestelmillä nopeammin kuin ihminen. Testejä kirjoittaessa täytyy vain muistaa kirjoittaa testeistä sellaisia, että ne ovat helposti siirrettävissä eri alustoille.

Kestävyystestaus kertoo miten testattava ohjelmisto toimii, jos se on käytössä useita viikkoja tai kuukausia yhtäjaksoisesti. Muistivuodot, pinojen korruptoituminen (stack corruption), villit osoittimet (wild pointer) ja muut tämän kaltaiset ongelmat eivät välttämättä ole ilmeisiä kun ne tapahtuvat, mutta varmasti aiheuttavat ongelmia. Yksi tapa päästä selville ongelmien olemassa olosta on suorittaa sarja testejä päivien tai viikkojen ajan käynnistämättä uudelleen ohjelmistoa. Tämän toteuttaminen vaatii automatisointia.

”Race conditions”-tyyppinen testaus tarkoittaa testauksen muotoa, jossa pyritään löytämään virheitä, jotka esiintyvät vain tietyissä olosuhteissa. Olosuhde muodostuu kun kahden säikeen tai prosessin samanaikainen ajoitus varaa saman resurssin aiheuttaen kilpailutilanteen, joka johtaa virheeseen.

Automaattinen testaus on isoksi avuksi tällaisten virheiden kanssa koska, samoja testejä voidaan toistaa useilla pienillä muutoksilla.

”Combination errors”-tyyppisessä testauksessa pyritään löytämään virheitä, jotka edellyttävät eri ominaisuuksien vuorovaikutusta. Automaattista testausta voi käyttää tällaisissa tilanteissa ajamaan suuria määriä monimutkaisia testejä, joissa jokaisessa käytetään eri ohjelman ominaisuuksia erilaisilla tavoilla.

Testaaja törmäävät työssään monesti tilanteisiin, jossa he ovat löytäneet ohjelmasta virheitä, mutta eivät myöhemmin enää kykene toistamaan tilannetta, jossa virhe esiintyi. Automaattinen testityökalu auttaa tällaisissa tilanteissa, sillä kehittäjä voi toistaa ajetun testiskriptin. Kehittäjän ei tarvitse huolehtia siitä, että jokainen eri vaihe tulee dokumentoida tarkasti, jotta virhe voidaan toistaa. (Dustin ym. 2009, 45.)

Automaattinen testaus tarjoaa testien toistettavuuden. Useimmat automaattiset testaustyökalut sallivat, että testausskriptit ajastetaan suoritettavaksi tiettynä aikana päivästä. Tätä ominaisuutta käyttäen voidaan testausta tehdä myös silloin kun työpäivät ovat päättyneet. Testaukseen voidaan ottaa mukaan myös useita eri tietokoneita ja niitä voidaan käyttää apuna yhtäaikaisissa ja hajautetuissa testeissä. (Dustin ym. 2009, 45-46.)

Elfriede Dustin esittää kirjassaan, että regressiotestausta voidaan parantaa jos testaus tehdään automaattisella testaustyökalulla. Työkalu testaa kaikki uudet ominaisuudet, jota ohjelmaan ollaan lisäämässä ja varmistaa ettei niissä ole virheitä. Dustin esittää, että automaattisen testityökalun olisi syytä tehdä regressiotestaus, sillä se on aikaa vievää, ikävystyttävää ja tämän takia virhealtis vaihe. (2009, 43-44.)

Cem Kanerilla on kriittisempi näkökulma kirjassaan regressiotestauksen automatisoimisesta. Hänen mukaansa suurimpia ongelmia regressiotestauksen automatisoinnissa on testien rappeutuminen ja kuoleminen. Käyttöliittymän ja tulosteiden muuttaminen uusien ominaisuuksien tullessa ohjelmaan ovat suurimpia syitä siihen miksi regressiotestaukseen tehdyt regressiotestaukset

rappeutuvat. Isot määrät testejä, jotka ennen menivät testeistä läpi muutosten jälkeen epäonnistuvat vaikkakin selvästi mitään sellaisia muutoksia ei ole tehty ohjelmaan, jotka vaikuttaisivat testeihin. Muutokset voivat olla niin vähäisiä että manuaalista regressiotestausta tekevä ihminen ei huomaisi niitä. Tämä vie paljon aikaa testaajilta, sillä testit generoivat paljon vääriä tuloksia ja näin olleen on tehotonta ja kallista. (2002, 108.)

Kanerin mukaan testejä korjattaessa, korjaajat voivat tehdä virheitä lisäten testeihin virheitä. Tällöin regressiotestauksessa testiskriptit kehittävät itse regressiovirheitä. Kaner on kirjassaan tullut tulokseen, jossa regressiotestauksen automatisointi on liian kallista ja aikaa vievää saataviin hyötyihin nähden. Rappeutuneiden testiskriptien tulosten analysointi ja korjaus vie liikaa aikaa ja on näin poissa työntekijöiden jäljellä olevasta työajasta. (2002, 108.)

Kirjoittajien näkökulmia tulkiten regressiotestauksen automatisoinnista on varmasti hyötyä, mutta regressiotestausta on syytä automatisoida maltillisesti. Automatisointia voi käyttää hyödyksi esimerkiksi testaajan kannalta ikävystyttävien ja paljon toistoja sisältäviin kohtiin.

3.2 Manuaalisen testauksen vahvuusalueet

Automaattisesta testauksesta voidaan olla montaa mieltä, sillä sen avulla ei voida ratkaista kaikkia testaamiseen liittyviä ongelmia. Automaattiselle ja manuaaliselle testaukselle on omat tilanteensa, joihin ne sopivat parhaiten. Manuaalinen testaus sopii esimerkiksi tilanteisiin, joissa on tiukka aikataulu ja automaattista testausta ei ole vielä olemassa, sillä automaattisen testauksen toteuttaminen vie enemmän aikaa kuin ihmisen suorittama manuaalinen testaus.

Manuaalista testausta tekevä ihminen voi vahingossa löytää uusia virheitä, sillä ihminen tahattomasti suorittaa testejä pienillä erilaisilla muutoksilla. Tällaisiin

suorituksiin ei tietokone kykene, sillä siltä puuttuu älykkyys. Tietokone suorittaa testejä samalla tavalla joka kerta. (Bach, Kaner & Pettichord 2002, 99.)
Tietokoneen virheentunnistuskyky on myös rajoittuneempi verrattuna ihmiseen. Tietokone ei huomaa jos kaiuttimista lähtee outoja ääniä, monitorin kuva värisee testejä suorittaessa tai ohjelman suorituskyvyn huonontumisesta. Ihminen huomaisi tällaiset ongelmat ilman erillisiä ohjeita. Automaattisella testauksella on monenlaisia vahvuuksia, mutta siitä ei ole kilpailemaan ihmisen kanssa. Tietokone voi laukaista virhetilanteita, mutta ne jäävät siltä huomaamatta, sillä se on tehty etsimään tietynlaisia virheitä. (Bach ym. 2002, 99.)

Toisena esimerkkinä voidaan mainita tilanne, on myös tiukka aikataulu ja testauksen kohteena olisi ohjelmiston käyttöliittymä. On myös tiedossa, että käyttöliittymä on muuttumassa lähiaikoina. Tällaisessa tilanteessa manuaalinen testaus on myös oikea työkalu, sillä vaikka automaattiset testit olisivat käyttöliittymään olemassa, ne lakkaisivat todennäköisesti toimimasta siinä vaiheessa kun käyttöliittymää muutetaan. Automaattiset testit vaativat ylläpitoa, sillä niiden täytyy muuttua testattavan ohjelmiston kanssa, muuten ne lakkaavat toimimasta ja ovat hyödyttömiä. Manuaalinen testaus on siis lyhyemmällä aikavälillä kannattavampaa. Vasta pidemmällä aikavälillä automaattisesta testauksesta on hyötyä. (Selenium project 2010a, hakupäivä 25.2.2010.)

3.3 100 %:n testikattuvuus automaattisen ohjelmistotestauksen avulla

Kokonaisia ohjelmistoja ei voida testata täydellisesti edes automaattisen testauksen avulla. Yksi merkittävin syy miksi testauksella on mahdollisuus muodostua äärettömäksi tehtäväksi on se, että ohjelman eri toiminnot tulisi testata kaikella mahdollisella syötteillä, niin hyväksyttävillä kuin väärillä. Automaattinen testaus voi laajentaa testauksen laajuutta ja syvyyttä, mutta automatisoinnin avullakaan ei ole tarpeeksi aikaa tai resursseja ohjelmiston täydelliseen testaukseen. On mahdotonta testata sataprosenttisesti järjestelmän kaikkia mahdollisia syötteitä tai polkuja, koska syötteiden tai polkujen eri vaihdosten ja yhdistelmien määrä on niin valtava. Testien suunnittelijat käyttävät erilaisia tekniikoita kuten ekvivalenssiositusta, jossa järjestelmää

testataan vain sitä edustavilla syötteillä. Tällöin järjestelmää ei testata kaikilla mahdollisilla syötteillä vaan mahdollisesta syöteavaruudesta testataan muutama virheellinen syöte ja muutama hyväksyttävä syöte. (Dustin ym. 2009, 79.)

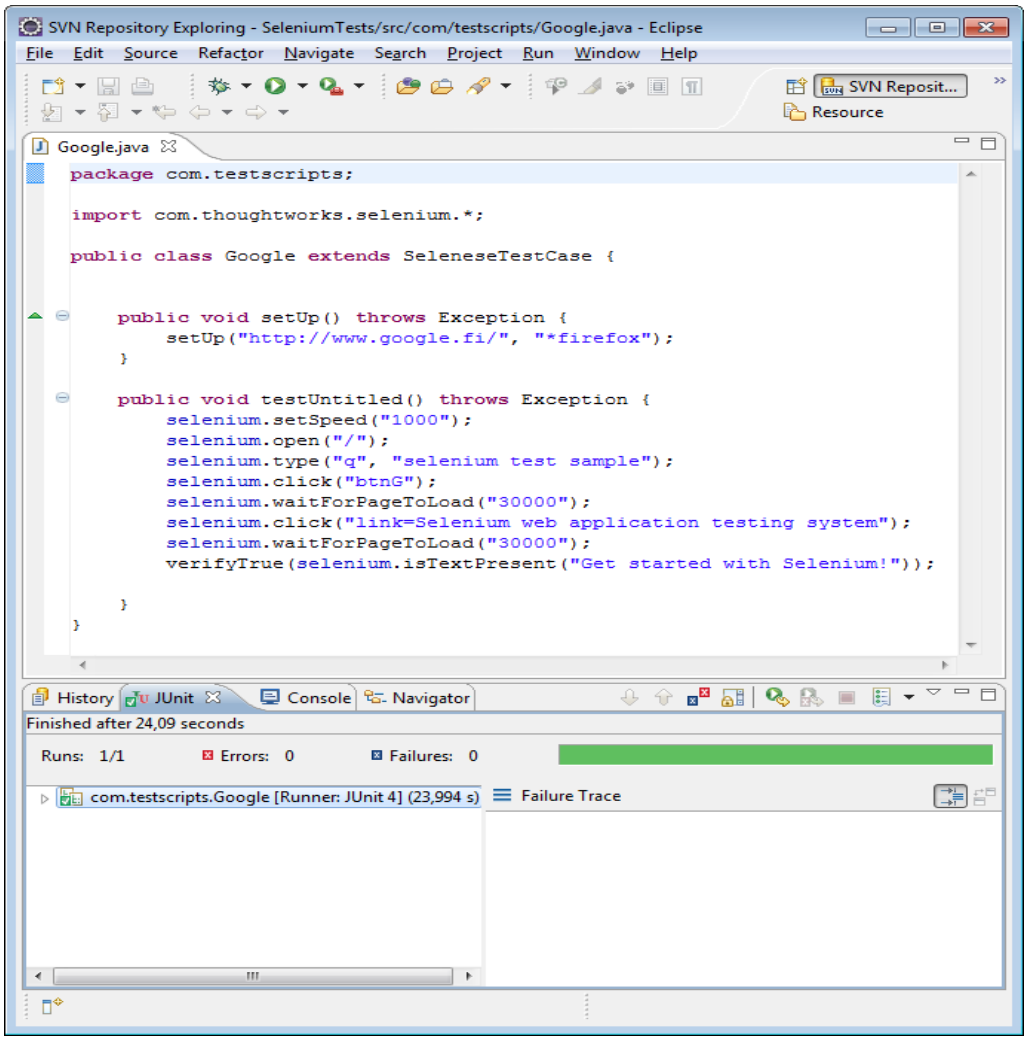
4 OHJELMISTOT JA TESTAUSTYÖKALUT

Seuraavissa alaluvuissa esitellään opinnäytetyössä käytetyt ohjelmistot. Ohjelmistot voidaan jaotella kahteen osaan. Aluksi esitellään neljä työasemalle asennettavaa ohjelmistoa, joita käytetään testiskriptien kirjoittamiseen ja nauhoittamiseen sekä Java-kielellä olevien testiskriptien kääntämiseen. Lopuksi esitellään palvelinohjelmisto, joka huolehtii Selenium-testiskriptien jatkuvasta ajamisesta.

4.1 JUnit

JUnit on Java-ohjelmointikielelle suunniteltu testauskehys, jonka avulla voidaan kirjoittaa ja toistaa yksikkötestejä. JUnit:n ajatus on se, että ohjelman kehittäjä testaa itse ohjelmansa kirjoittamalla ohjelmiston kehityksen ohessa JUnit-testejä varmistaen, että ohjelma toimii. JUnit siis tukee siis vahvasti testilähtöistä ohjelmiston kehitystä. (SearchSoftwareQuality 2010, hakupäivä 3.3.2010.)

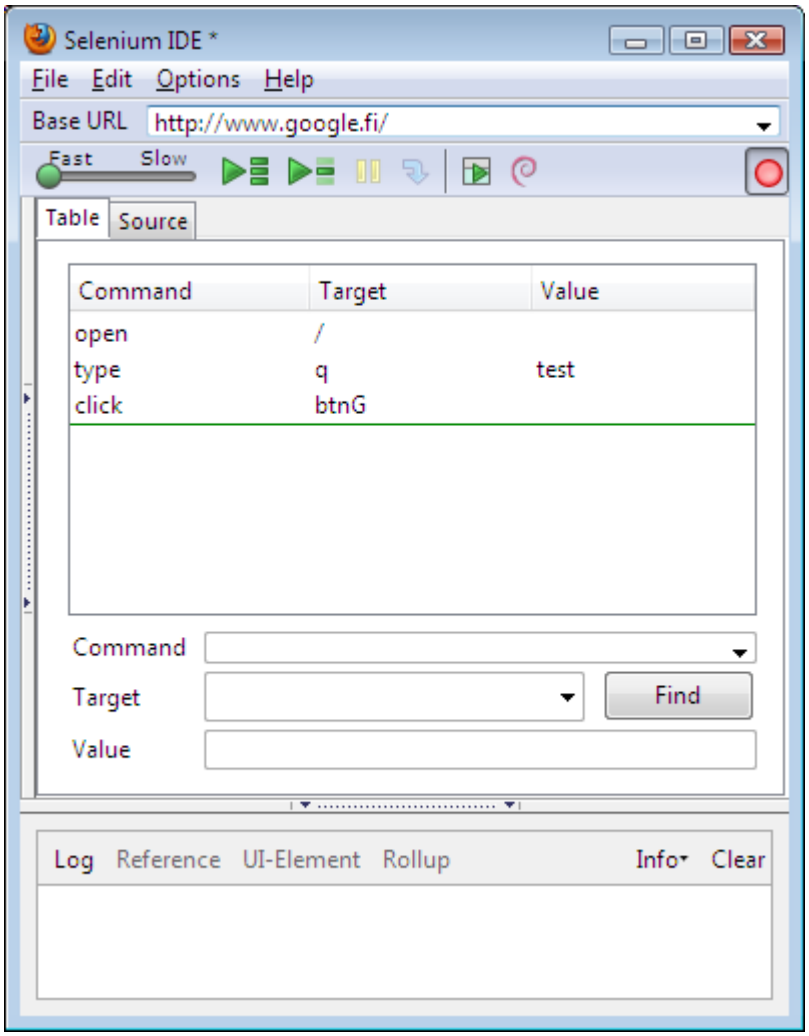
JUnit:ssa on graafinen käyttöliittymä, jonka avulla testien kirjoitus ja toisto sujuu nopeasti. Testejä pystytään ajamaan testejä taukoamatta sekä useita eri testejä yhtä aikaa ja tulokset on näkyvissä käyttäjälle heti. JUnit kertoo, kauanko eri testien ajamiseen meni ja mitkä ovat tulokset. Vihreällä viivalla kuvataan kaikkien testien onnistumista ja punaisella jos yksi tai useampi testi epäonnistuu. JUnit:n yksinkertaisuus on tehnyt siitä Java-ohjelmointikielen yksikkötestityökaluna de facto standardin. JUnit löytyy useimmille kehitysympäristöille, kuten Eclipselle ja NetBeansille. (SearchSoftwareQuality 2010, hakupäivä 3.3.2010.)



Kuva 1. JUnit Eclipse ohjelmointiympäristössä

4.2 Selenium IDE

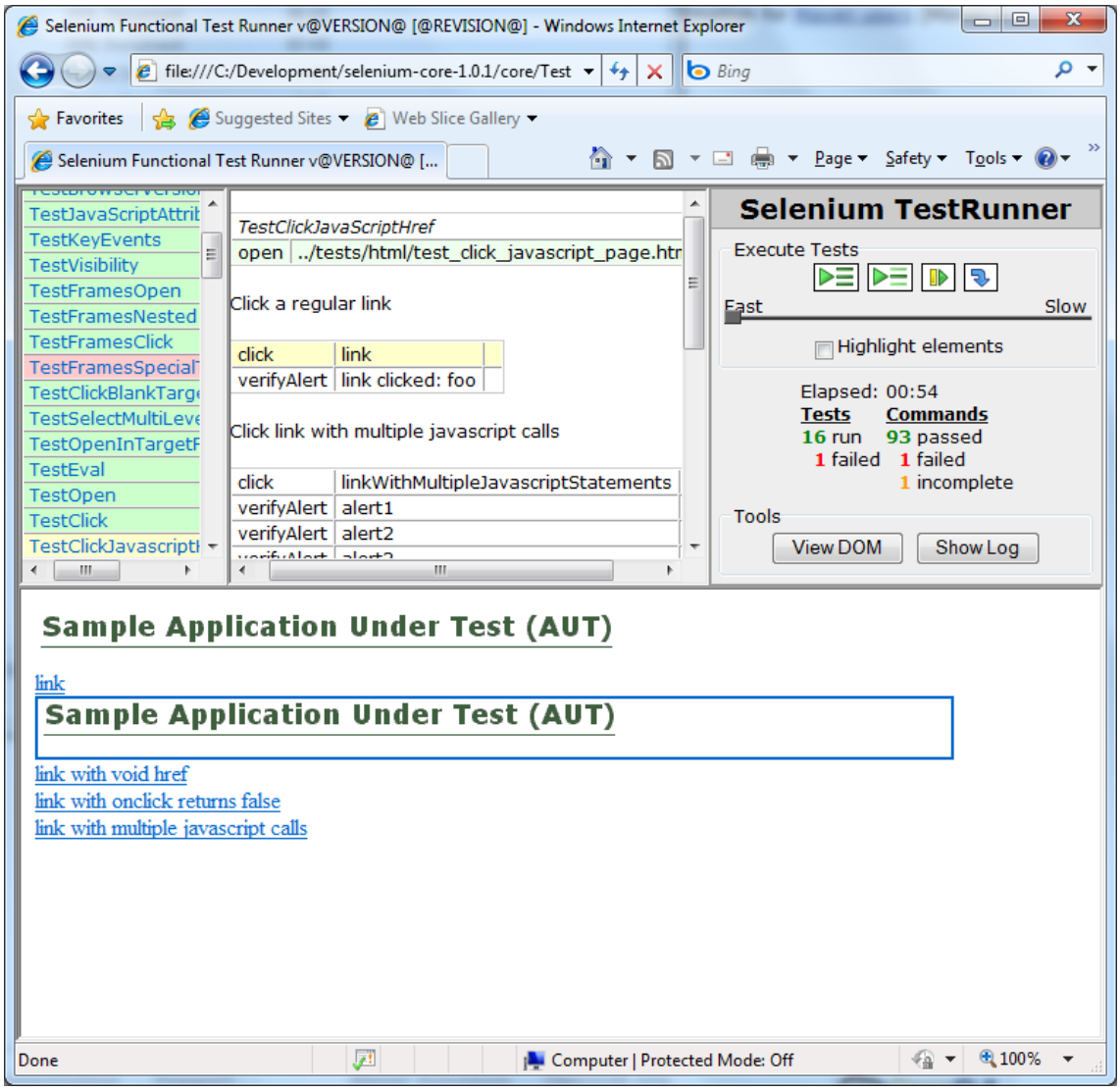
Selenium IDE on Mozilla Firefox Internet-selaimen laajennus, joka on avoimeen lähdekoodiin pohjautuva ilmainen testaustyökalu. Se on suunniteltu websivujen ja -sovellusten testaukseen ja sen päätoiminto on testien nauhoitus ja niiden toistaminen. Selenium IDE:n käyttö on yksinkertaista selkeän graafisen käyttöliittymän takia. Nauhoittaminen tapahtuu käynnistämällä Selenium IDE laajennoksen ja normaalisti websivustoja selaamalla. Selenium IDE seuraa käyttäjän toimia ja nauhoittaa jokaisen hiiren ja näppäimistön painalluksen. Nauhoituksen aikana Selenium IDE generoi skriptin, jota voidaan käyttää nauhoitetun testin toistamiseen Selenium IDE:ssä. Skriptiä toistaessa Selenium IDE:n tekemät liikkeet näkyvät käyttäjälle selaimessa.



Kuva 2. Selenium IDE:n käyttöliittymä

Selenium IDE tukee useita eri ohjelmointikieliä, joihin nauhoitetut testit voidaan tallentaa. Tuettuja kieliä ovat Java, Groovy, C#, Perl, PHP, Python ja Ruby. Javan ja Groovy:n tapauksessa testit tallennetaan suoraan JUnit:n syntaksin mukaisesti. Tämän takia nauhoitettuja testejä voidaan käyttää helposti myös muissa käyttötarkoituksissa sekä yhdessä muiden testityökalujen kanssa.

Kirjoitushetkellä testejä ei voi nauhoittaa kuin Firefox selaimella, sillä Selenium IDE ei tue muita selaimia. Selenium IDE:llä generoituja testiskriptejä voidaan toistaa muissa selaimissa ja käyttöjärjestelmissä Selenium RC sekä Selenium Core pakettien avulla. (TestingGeek 2009, hakupäivä 15.11.2009.)



Kuva 3. Selenium core paketista löytyvä Test Runner

4.3 Selenium RC

Selenium RC eli Remote Control on konsolipohjainen palvelinohjelma, jolla voidaan toistaa Selenium testejä useilla eri selaimilla. Selenium RC on kirjoitettu Javalla, joka tekee siitä alustariippumattoman. Se siis toimii useissa eri käyttöjärjestelmissä kunhan vai Java on vain tuettu.

Selenium RC:ssä toistettuihin testeihin voidaan siis lisätä toiminnallisuutta ja ohjelmointilogiikkaa, kirjoittamalla Selenium-testiskriptin sekaan ohjelmointikieltä. Apuna voidaan käyttää vaikka taulukoita tai for-looppeja ohjelmointikielistä. Selenium IDE:llä nauhoitettuja testejä voidaan siis laajentaa

ja parantaa, sillä edellä mainittuja asioita Selenium IDE ei suoraan tue. Selenium RC:n avulla voidaan automatisoida testausta. Selenium testejä voidaan määrittää ajattavaksi useita kertoja päivässä ja useilla eri selaimilla.

Selenium RC tukee log-tiedostojen kirjoitusta, jolloin log-tiedostoja apuna käyttäen voidaan ajettujen testien tuloksista rakentaa kattavia raportteja. Eri testityökaluille löytyy jo valmiita raporttien generointi ohjelmia, joita voidaan käyttää apuna myös Selenium RC:n tapauksessa. Testejä JUnit:n kautta ajaessa voi raporttien generoinnissa käyttää JUnit Report työkalua. Muita työkaluja ovat esimerkiksi TestNG Reports sekä ReportNG. (Selenium project 2010b, hakupäivä 3.3.2010.)

4.4 Maven

Maven on samankaltainen ohjelma kuin Apache Ant, jonka tarkoituksena on helpottaa Java ohjelmien kääntämistä. Maven on myös konsolipohjainen ohjelma, jolloin ohjelmaa käytetään komentokehotteeseen kirjoittamalla Mavenin ymmärtämiä komentoja. Maven erottaa projektista erikseen konfigurointi tiedostot, projektin dokumentaation, projektissa vaadittavat riippuvuudet kuten ulkoiset kirjastot sekä mahdolliset pluginit, joita projekti käyttää. (Redmond, E. 2006, Hakupäivä 4.3.2010.)

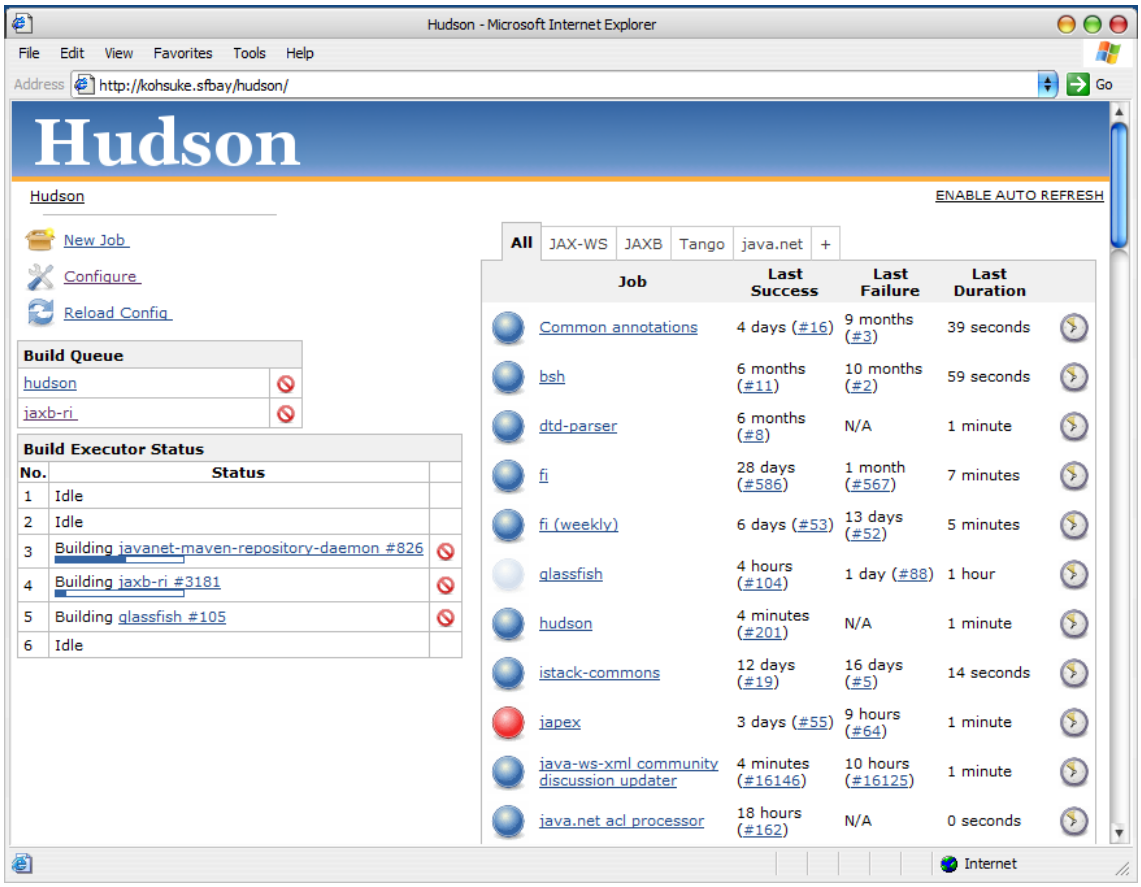
Maven siis luo projektista kansiorakennerungon, jonka avulla kansiorakenne yksinkertaistuu ja ohjelmointiympäristön valinnalla ei ole merkitystä, sillä jokaisella ohjelmointiympäristöllä on omanlainen tapa luoda projektien hakemistorakenteet. Maven projekteja pystyy käyttämään vaivattomasti eri ohjelmointiympäristöissä. (Spagettikoodi 2009, Hakupäivä 20.5.2010.)

Tärkeimpiä Mavenin ominaisuuksia on riippuvuuksien hallinta. Riippuvuuksien hallinnan avulla on helppo testata millä tavalla eri kirjastojen tuominen mukaan projektiin vaikuttaa testauksen läpäisyyn. Mavenin avulla riippuvuuksien lisääminen ja poistaminen on helpompaa. (Spagettikoodi 2009, Hakupäivä 20.5.2010.)

4.5 Hudson

Hudson on avoimeen lähdekoodiin perustuva jatkuvan integroinnin palvelinohjelma, joka on kirjoitettu Javalla. Hudsonilla Java-projekti kootaan ja testataan tasaisin aikavälein. Hudson vahtii, ettei Hudsonissa oleviin projekteihin pääse virheitä ja projektit pystytään kääntämään. Ongelman ilmetessä, Hudson ilmoittaa niistä esim. sähköpostin välityksellä. Hudson on siis yksi automaattisen ohjelmistotestauksen työkaluista. (Dyer, D. 2008, hakupäivä 4.3.2010)

Hudsoniin löytyy useita eri laajennoksia, joilla Hudsonia voi muokata mieleisekseen ja lisätä siihen ominaisuuksia. Hudsoniin löytyy myös satoja laajennoksia, joilla Hudsonin saa toimimaan yhdessä toisten ohjelmien kanssa. Esimerkkeinä mainittakoon SeleniumHQ laajennus ja IRC laajennus.
SeleniumHQ laajennus mahdollistaa ajettujen Selenium testien tulosten tuomisen Hudsoniin, jolloin Hudson luo tuloksista luettavat raportit. IRC laajennus on IRCbot, joka ilmoittaa halutulla IRC kanavalla Hudsonissa tapahtuneista asioista, kuten onnistuneista tai epäonnistuneista käännöksistä ja testeistä. (Kawaguchi, K & Harder, A. 2010, hakupäivä 4.3.2010)



Kuva 4. Hudsonin web-käyttöliittymä

5 TESTISKRIPTIEN KIRJOITUS JA KÄYTTÖÖNOTTO

Tässä kappaleessa kerrotaan opinnäytetyön toiminnallisen osan vaiheista. Aluksi on kuvattu testiskriptien suunnittelusta, jonka jälkeen kerrotaan testien kirjoittamisesta. Lopuksi on kuvattu testien käyttöönotto.

5.1 Ohjelmiston testattavat ominaisuudet

Selenium-testiskriptien tekoon liittyy paljon suunnittelua, sillä huomioitavia asioita on paljon. Ennen Selenium-testiskriptien kirjoitusta on tärkeää tunnistaa mitä ohjelman ominaisuuksista kannattaa lähteä testaamaan automaattisesti. Asiaa voi pohtia manuaalista testausta suorittavan ihmisen kannata ja miettiä mitkä ominaisuudet ohjelmassa ovat ikävystyttäviä ja paljon toistoja sisältäviä. Helppoja kohteita ovat esimerkiksi sisäänkirjautumisen testaamisen automatisointi, sillä sen testaaminen on lähinnä erilaisten syötteiden avulla kirjautumista.

Testiskripteistä olisi hyvä tehdä sellaisia, että ne ovat helposti ylläpidettävissä eivätkä rikkoudu pienistä käyttöliittymän muutoksista. Mitä enemmän työaikaa käytetään rikkinäisten testien korjaamiseen tai testien päivittämiseen, sitä vähemmän automaattinen testaus tuo säästöjä.

Monet testit sisältävät sellaisia toimintoja, jotka ovat välttämätöntä tehdä testien läpikäynnin kannalta, mutta niitä ei kannata lähteä toteuttamaan jokaiseen mallitestitapaukseen erikseen. Tällainen toiminto voisi olla esimerkiksi ohjelmaan sisäänkirjautuminen. Tämänkaltaisista toiminnoista olisi syytä kirjoittaa erilliset metodit, joita kutsutaan erikseen jokaisessa testissä. Tämä auttaa testien ylläpidettävyydessä, sillä jos jokin muuttuu sisäänkirjautumisprosessissa, ei muutosta tarvitse lähteä tekemään jokaiseen testiin erikseen. Riittää jos muutoksen käy tekemässä sisäänkirjautumista

hoitavaan metodiin. Seuraavaksi esitelty tekniikka auttaa testien myös testien ylläpidettävyyden kanssa.

5.2 UI Mapping

UI mapping on mekanismi, joka tallentaa käyttöliittymä elementtien tunnisteet eli ID:t. Samalla elementeille annetaan toinen kuvaavampi nimi, jota käytetään testiskripteissä elementtien kutsumiseen.

UI mapping auttaa testiskriptien hallinnassa, koska tiedot eri käyttöliittymäelementeistä sijaitsevat mappauksen jälkeen yhdessä paikassa. Jos elementtien tunnisteita tai nimiä pitää muuttaa, uusia elementtejä poistaa tai lisätä, ei kartoituksen jälkeen muutoksia tarvitse tehdä kuin yhteen paikkaan. Tämä säästää aikaa ja vaivaa ja tekee myös testiskriptistä helpommin luettavaa, sillä testiskripteissä käytetään kartoituksessa annettuja uusia elementtien nimiä, jotka ovat kuvaavampia kuin elementtien ID:t

Alla olevassa esimerkissä on Selenium-testiskripti Java-kielellä, jossa ei ole käytetty UI Mapping -tekniikkaa. Skripti on vaikea ymmärtää, sillä siitä ei ilmene tarkasti mitä käyttöliittymän elementtejä skriptissä käytetään.

```
1  public void testNew() throws Exception {
2              selenium.open("http://www.test.com");
3              selenium.type("loginForm:tbUsername","xxxxxxxx");
4              selenium.click("loginForm:btnLogin");
5              selenium.click("adminHomeForm:_activitynew");
6              selenium.waitForPageToLoad("30000");
7              selenium.click("addEditEventForm:_idcancel");
8              selenium.waitForPageToLoad("30000");
9              selenium.click("adminHomeForm:_activityold");
10             selenium.waitForPageToLoad("30000");
11  }
```

Alla on esimerkki tiedostosta, johon mapatut elementit tulevat. Jokaiselle elementille on annettu uusi kuvaavampi alias, jota käytetään testiskripteissä.

```
admin.username = loginForm:tbUsername
admin.loginbutton = loginForm:btnLogin
admin.events.createnewevent = adminHomeForm:_activitynew
admin.events.cancel = addEditEventForm:_idcancel
admin.events.viewoldevents = adminHomeForm:_activityold
```

UI Mappingin avulla uudet elementtien aliakset tekevät skriptistä huomattavasti luettavamman. Aliakset kertovat minkälaisista elementeistä on kyse. Alla näkyvässä esimerkissä esitetään yllä esitetty koodi UI Mappingia käyttäen. (Selenium project 2009c, hakupäivä 13.4.2010.)

```
1  public void testNew() throws Exception {
2              selenium.open("http://www.test.com");
3              selenium.type(admin.username, "xxxxxxxx");
4              selenium.click(admin.loginbutton);
5              selenium.click(admin.events.createnewevent);
6              selenium.waitForPageToLoad("30000");
7              selenium.click(admin.events.cancel);
8              selenium.waitForPageToLoad("30000");
9              selenium.click(admin.events.viewoldevents);
10             selenium.waitForPageToLoad("30000");
11 }
```

5.3 Assertointi ja verifiointi

Selenium käyttää termejä assertointi ja verifiointi kuvaamaan kahta sen kautta hallittavissa olevaa testausmetodia. Assertoinnin ja verifioinnin ero on lähinnä virheiden hallinta ja mukavuuskysymys. Assertointi lopettaa testin ajamisen jos ehto ei täyty siinä missä verifiointi jatkaa eteenpäin vaikka ehto ei täyttyisikään. Molemmille on omat käyttötilanteensa. Assertointeja on hyvä käyttää testien alkupäässä esimerkiksi tilanteissa, joissa on tarkoitus tarkistaa onko sivulla oleva painike olemassa. Ei ole mitään hyötyä tarkistaa onko painike olemassa, jos testi epäonnistunut kun testi on menossa väärällä sivulla, jossa kyseistä painiketta ei ole olemassa. Tässä tapauksessa testin alkuun olisi hyvä lisätä assertointi, jossa tarkistetaan ollaanko oikealla sivulla, jonka jälkeen verifioidaan onko painike olemassa. (Selenium project 2009d, hakupäivä 23.3.2010)

```
1  public void assertAndVerificationExample() throws Exception {
2              selenium.open("http://seleniumhq.org");
3              selenium.waitForPageToLoad("30000");
4              selenium.open("/download/");
5              selenium.click("link=Download");
6              selenium.waitForPageToLoad("30000");
7              assertEquals("Downloads", selenium.getTitle());
8              verifyTrue(selenium.isTextPresent("Downloads"));
9              assertEquals("Selenium Core",
10                selenium.getTable
11                ("//div[@id='mainContent']/table[1].1.0"));
12             verifyEquals("June 10, 2009",
13                selenium.getTable
14                ("//div[@id='mainContent']/table[1].1.1"));
15             verifyEquals("1.0.1", selenium.getTable
16                ("//div[@id='mainContent']/table[1].1.2"));
17  }
```

Yllä olevassa esimerkissä rivillä 7 on aluksi assertoitu ollaanko testissä oikealla sivulla vertaamalla sivun otsikkoa oletettuun arvoon. Vain jos tämä ehto läpäistään, edetään testissä eteenpäin, muutoin testin suoritus päättyy kyseiseen riviin. Tämän jälkeen rivillä 8 verifiodaan onko ”Downloads” oletetussa kohdassa, jonka jälkeen assertoidaan taulukon ensimmäinen rivin ensimmäinen solu. Jos taulukon ensimmäisessä rivillä ensimmäisessä solussa on oletettu arvo, verifiodaan loput taulukon solut. (Selenium project 2009d, hakupäivä 23.3.2010)

5.4 Testien kirjoitus

Itse testien kirjoittaminen Java-ohjelmointikielellä ja Selenium-kielen soveltaminen, ei ole kovin hankalaa, sillä Selenium-skripteissä käytetty kieli on suhteellisen selkeälukuista. Testin teko internet selainkäyttöliittymästä lähtee aluksi liikkeelle Selenium IDE Firefox -laajennuksen käytöstä. Selenium IDE:llä nauhoitetaan testi laittamalla nauhoituksen päälle ja tämän jälkeen käyttämällä käyttöliittymää kuin normaali käyttäjä. Selenium IDE nauhoittaa kaikki hiireltä ja näppäimistöltä tulevat syötteet sekä mitä eri elementtejä käyttöliittymässä käytettiin. Näistä toiminnoista Selenium IDE luo HTML-pohjaisen testiskriptin, jonka voi muuntaa Java-kielelle, JUnit-testin muotoon suoraan Selenium IDE:stä. Jos nauhoitetut testit muutetaan toiselle ohjelmointikielelle HTML-

pohjasta, ei testejä enää voida ajaa Selenium IDE:ssä. Muutetut testit voidaan tämän jälkeen ajaa Selenium RC:n avulla.

Lyhyenä esimerkkinä JUnitin rakenteesta ja Selenium-testistä esitellään seuraavaksi, mitä alla oleva lyhyt testi pitää sisällään. Varsinaiset testit, joita olen opinnäytetyöni toimeksiantajalle tehnyt, ovat olleet laajempia, mutta rakenteeltaan samankaltaisia. Alla oleva testi ajetaan Selenium RC ohjelmalla.

```
0  package com.example.tests;
1
2  import com.thoughtworks.selenium.*;
3  import java.util.regex.Pattern;
4
5  public class Esimerkki extends SeleneseTestCase {
6         public void setUp() throws Exception {
7                   setUp("http://www.google.com/", "*chrome");
8         }
9         public void testEsimerkki () throws Exception {
10                  selenium.open(“/”);
11                  selenium.type("q", "Selenium IDE");
12                  selenium.click("btnG");
13        selenium.click("//ol[@id='rso']/li[1]/h3/a/em");
14        selenium.waitForPageToLoad("30000");
15        verifyTrue(selenium.isTextPresent("Seleniumis a
16        suite of tools"));
17        }
18  }
```

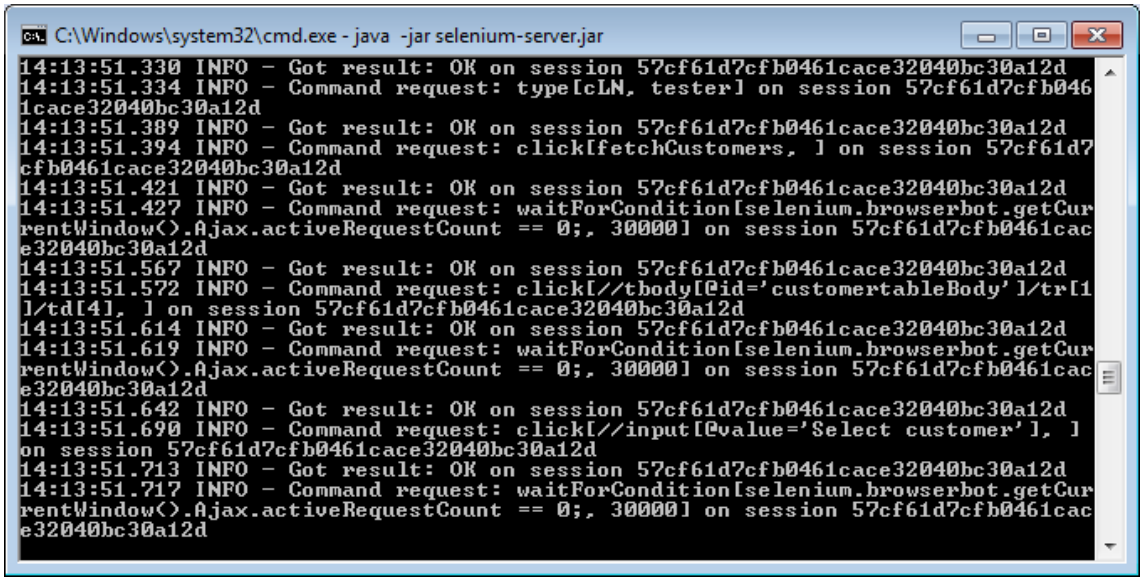
Esimerkki Selenium testiskriptistä, joka on muutettu Java-kielelle JUnit muotoon.

JUnit testi koostuu alussa olevista import-lauseista, jotka löytyvät riveiltä 0-3. Niillä tuodaan testiin testissä tarvittavat luokat. Sen jälkeen rivillä 5, on luotu itse testiluokka, jonka nimi tässä tapauksessa on Esimerkki. Esimerkki luokka perii SeleneseTestCase-luokan ominaisuudet. Tämän jälkeen rivillä 6 ja 7 on esitelty metodissa setUp(), joka suoritetaan ennen itse testin suorittamista. Tässä tapauksessa avataan www-osoite http://www.google.com, selaimella Google Chrome. Selain on määritelty setUp-metodiin, jotta Selenium RC tietää millä selaimella testi ajetaan.

Tämän ominaisuuden avulla voidaan samat testit laittaa ajettavaksi useilla eri selaimilla sekä voidaan varmistaa, että testattu järjestelmä toimii mahdollisimman monella eri selaimella.

setUp metodin jälkeen riveillä 9-16 on itse testi, jossa on ensimmäisenä toimintona rivillä 10 avattu setUp metodiin määritelty Googlen www-osoitteen juuresta eli pääsivulta. Tämän jälkeen rivillä 11 ja 12 on kirjoitettu Googlen pääsivulla olevaan hakutekstikenttään hakusana Selenium IDE ja painettu Google-haku painiketta. Rivillä 13 on klikattu ensimmäistä hakutulosta, jonka jälkeen rivillä 14 odotetaan, että ladattava websivu latautuu kokonaisuudessaan auki ennen testin jatkumista. Viimeiseksi rivillä 15 verifioidaan löytyykö ladatulta sivulta teksti ’Selenium is a suite of tools’ Rivillä 14 oleva sivun latautumisen odottelu on tärkeä vaihe, sillä testi epäonnistuu jos testissä edetään ennen kuin sivu on ladattu kokonaisuudessa. WaitForPageToLoad-metodilla vältetään tilanne, jossa testi yrittää löytää sivulta tekstiä tai elementtiä, jota ei löydy koska sivustoa ei keritty ladata kokonaan auki.

Testin nauhoituksen ja JUnit-muotoon muutoksen jälkeen testi on valmis ajettavaksi Selenium RC:ssä. Jäljellä jää enää testattavaksi se, toimiiko testi.
Selenium RC näyttää testiä ajaessaan jokaisen tapahtuvan toiminnon selaimessa sekä näyttää komentokehotteessa, mitä riviä testiskriptissä ollaan suorittamassa, missä ja minkälaisia virheitä tapahtuu.



Kuva 5. Selenium RC testin ajonaikana

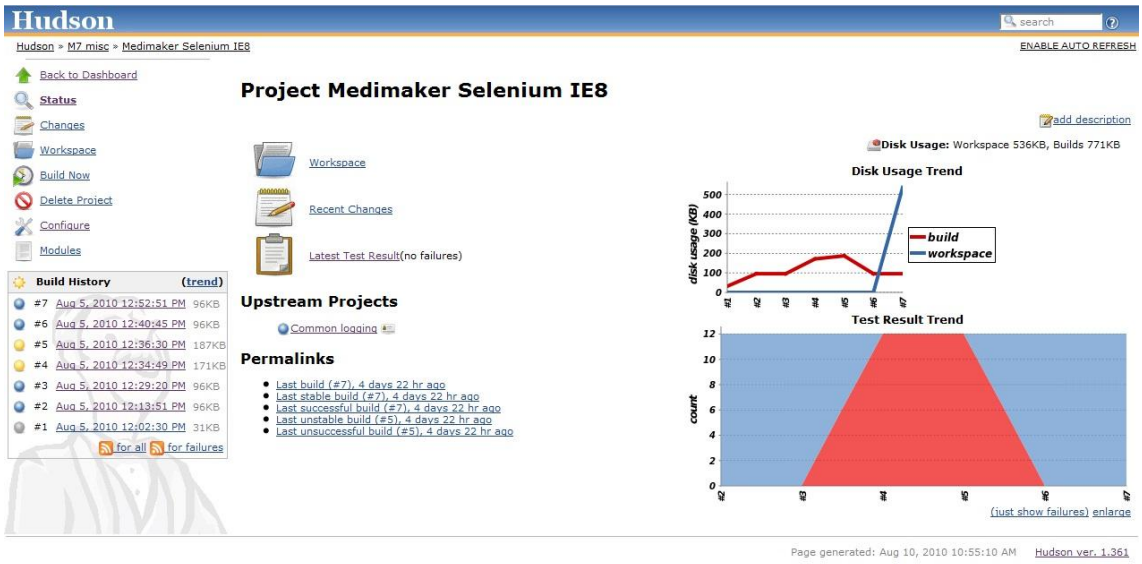
Omassa Selenium testiprojektissa itse Selenium testiskriptien lisäksi projektissa on myös yläluokka, jonka testiskriptit perivät. Yläluokassa on ylikirjoitettu JUnitin tearDown ja setUp -metodit sekä kirjoitettu apumetodeita sellaisista asioista kuten asiakkaan valinta MediMakerissa, sisäänkirjautuminen ja uloskirjautuminen. Näiden lisäksi yliluokan sisältä löytyy toinen luokka, joka muuttaa Selenium RC:n käyttäytymistä niin, että se tukee Maven komentojen parametrisoimista.
Maven komentojen parametrisoinnista kerrotaan seuraavassa kappaleessa.

5.5 Testien käyttöönotto

Kun Selenium-testiskriptit ovat valmiit, ne asetetaan Hudsonin suoritettavaksi tietyn väliajoin. Tällä tavoin testiskriptejä ei tarvitse olla manuaalisesti käynnistämässä vaan Hudson huolehtii testauksesta. Hudson laatii testien päätyttyä raportin testauksen tuloksista, josta ilmenee seuraavat asiat:

- Mitkä testit testattava koodi läpäisi
- Mitkä testit testattava koodi ei läpäissyt
- Mihin kohtaan läpäisemättä jääneiden testien suoritus pysähtyi
- Kauanko testien suorittamiseen käytettiin aikaa
- Milloin testit suoritettiin

Raporttien avulla testauksen kulusta saa samanlaisen käsityksen kuin olisi testauksen suoritusta seuraamassa. Testauksen kulkua voi toki myös seurata Hudsoniin tulevien konsolitulosteiden avulla, mutta tämä ei ole välttämätöntä raporttien takia.



Kuva 6. Yleisnäkymä testien suorituksesta Hudsonissa

Pelkkä Hudson ei tässä tapauksessa itsessään riitä Selenium-testiskriptien ajamiseen automaattisesti vaan Hudsonin ja Selenium RC:n lisäksi, Mavenilla on myös oma tärkeä tehtävänsä. Mavenin komentoja voi parametrioida eli Mavenin peruskomentojen seuraksi voidaan tehdä omia parametrejä. Peruskomentojen lisäksi tarvitaan kaksi lisäparametria joihin annetaan IP-osoite, jossa testiskriptit ajetaan sekä millä selaimella testit ajetaan. Nämä tiedot Maven välittää Selenium testiskripti projektin yläluokalle. Tiedot luetaan yläluokasta kun testiskriptejä aletaan ajamaan Selenium RC:ssä. Uusien parametrien avulla voidaan Selenium testiskriptit asettaa Hudsonissa ajettavaksi eri koneilla ja selaimilla. Tämä vaatii aluksi vain sen että kyseisille koneille on asennettu Selenium RC ja selaimet, joilla testiskriptit aiotaan ajaa. Selenium RC täytyy olla valmiina päällä siinä työasemassa, jossa testit aiotaan suorittaa.

6 TULOKSET JA JOHTOPÄÄTÖKSET

Opinnäytetyön toiminnallisen osuuden työstämisen aikana aikaa kului paljon teknisien ongelmien ratkontaan, kuten ponnahdusikkunoiden hallintaan Seleniumissa ja dokumentoimaton metodi checkForVerificationErrors. Ponnahdusikkunat ovat MediMakerissa toteutettu erilaisella tekniikalla perinteiseen HTML:n verrattuna, joka aiheuttaa sen etteivät ponnahdusikkunat avaudu Selenium testiskriptejä ajaessa. Varsinaista ratkaisua ongelmaan ei löytynyt, vaan ongelma täytyi kiertää aukaisemalla ponnahdusikkunan URL suoraan testiskriptin koodissa sen sijaan, että ponnahdusikkuna aukaistaisiin navigaatiopainikkeita painamalla. Tämän osalta tehdyt testiskriptit ovat puutteellisia.

checkForVerificationErrors on metodi, jota käytetään verifikaatiotilanteissa virhetilanteiden käsittelemiseen. Metodi täytyy kirjoittaa jokaisen testin perään tai viimeistään tearDown-metodissa. Ilman metodia verifikaatiolauseet palauttavat *true* arvon, vaikka oikeasti tilanteessa tapahtuisikin virhe ja palautettavan arvon tulisi olla *false*. Tämä johtaa siihen, että testit menevät läpi vaikka oikeasti ne sisältävätkin virheitä.

Työssäni esitellystä UI Mapping -tekniikasta ei ollut hyötyä testiskriptejä kirjoittaessa, sillä kaikilla käyttöliittymän elementeillä ei ollut olemassa valmista ID:tä, eikä niitä olisi kannattanut lähteä lisäämään niitä vain opinnäytetyöni takia. Tekniikka olisi parantanut testiskriptien ylläpidettävyyttä, mikäli sitä olisi voinut käyttää.

Selenium-testiskriptien ajaminen eri selaimissa ei välttämättä ole täysin saumatonta. Esimerkiksi Internet Explorer 8:n kohdalla ennen testien ajamista täytyy asetuksien kautta poistaa käytöstä ponnahdusikkunoiden esto sekä suojattu tila. Jos nämä asetukset ovat päällä ja testejä yritetään ajaa IE 8 -selaimessa, voi testien suoritus pysähtyä jo alussa Script Error -virheviestiin, joka johtuu ponnahdusikkunaestosta.

Ennen testiskriptien ajoa on syytä varmistaa, ettei testiskriptejä suorittavassa koneessa ole maa-asetukset asetettu osoittamaan Suomeen, sillä maa-asetukset vaikuttavat MediMaker 7 -käyttöliittymän lokalisointiin. Testiskriptit ovat kirjoitettu testaamaan englanninkielistä käyttöliittymää, eivätkä ne toimi ilman muutoksia suomenkielisessä käyttöliittämässä.

Jatkokehitystä ajatellen ensimmäiset askeleet olisivat oman tietokannan hankinta testiskriptien ajamista varten ja työasemien hankinta, jotka hoitavat Selenium RC:n ajoa ja testiskriptien suoritusta. Selenium-testiskripteille tarkoitettu oma tietokanta helpottaisi testitulosten ennakoitavuutta sekä varmistaisi, etteivät testiskriptit rikkoutuisivat helposti tietokantaan kohdistuvien muutosten takia. Kun oma tietokanta testeille olisi olemassa, myös monimutkaisempien testiskriptien teko olisi mahdollista. Tämän opinnäytteen testiskripteistä on pyritty tekemään sellaisia, että niissä assertoidaan tai verifioidaan mahdollisimman vähän tietoa, joka tulee tietokannasta tai on riippuvainen siitä. Kun tietokantaa käytetään muuhunkin kuin testaamiseen ja sieltä poistetaan, muutetaan tai lisätään tietoa, jota testiskriptit käyttävät, aiheuttaa se testiskriptejä ajaessa testien epäonnistumisen.

Myöhemmässä vaiheessa, jos testiskriptejä on huomattavasti enemmän ja jos testiensuoritusajat ovat liian suuret, voi ratkaisuna koittaa Selenium Grid:n asennusta Selenium RC:n päälle. Selenium Grid vähentää testiskriptien ajamisesta aiheutuvaa kuormaa hajauttamalla testien suorituksen eri koneille ja tällä tavalla pienentää suoritusaikaa. Selenium Grid kykenee myös ajamaan useita testiskriptejä samanaikaisesti.

Lopputuloksena opinnäytetyön toiminnallisesta vaiheesta syntyi pohja MediMaker 7 selainkäyttöliittymän automaattiselle testaukselle. Se koostuu Hudson, Maven ja Selenium RC –ohjelmista, Selenium-testiskripteistä sekä dokumentista. Dokumentissa on kuvattu Selenium-testiskriptit tarkemmin mitä jokainen testiskripti sisältää, mitä ne testaavat, mikä on oletettu testin lopputulos sekä missä luokassa ja minkä nimisessä metodissa testit sijaitsevat. Testiskriptit ovat toiminnaltaan ja rakenteeltaan yksinkertaisia, eivätkä ne ole

määrältään kattavia. Niitä voidaan käyttää pohjana uusia testiskriptejä kirjoittaessa, kun testikattavuutta halutaan laajentaa.

Testiskriptit voidaan ajaa automaattisesti haluttuna ajankohtana Hudsonin avulla. Testiskriptit tukevat Maven-komentojen parametrisointia, jolloin Hudsoniin voidaan syöttää parametreinä missä koneessa testiskriptit halutaan suorittaa ja mitä selainta testien suorittamiseen käytetään. Käyttäjää testien suorituksen kannalta tarvitaan ainoastaan testiskriptien tulostan analysointiin, mikäli Selenium RC on asetettu palveluksi työasemiksi, jotka hoitavat testiskriptien suorittamisen. Tällöin Selenium RC on valmiina tietokoneen taustaprosessina heti käynnistyksestä lähtien.

Ainoina vaatimuksina työasemien kannalta, jotka suorittavat Selenium-testiskriptit ovat, että työasemilla tulee olla asennettuna JRE versio 1.5 tai uudempi, sekä se että työasemiin on asennettu selaimet, joilla testiskriptit aiotaan suorittaa.

Kokonaisuudessaan toimintaketju Hudson-Maven-Selenium RC –ohjelmien välillä on seuraavanlainen:

- Hudson ajaa Maven komennon tiettyinä hetkinä. Testien suorituskomennon lisäksi mukana on myös parametreinä IP-osoite koneelle sekä selain, jossa testit ajetaan.
- Maven välittää parametreina annetut tiedot Selenium testiprojektin koodiin sekä ajaa koodin.
- Koodin suorituksen seurauksena Selenium RC ajaa Selenium testiskriptit.
- Testauksen tulokset välittyvät takaisin Hudsoniin.

7 POHDINTA

Tämän opinnäytetyön tavoitteena oli kehittää toimeksiantajalle pohja Medimaker 7 -ohjelmiston automaattiselle toiminnalliselle testaukselle. Tämä tarkoittaa Selenium-testiskriptien kirjoitusta ja testausympäristön suunnittelua.

Työ onnistui mielestäni vähintäänkin kohtalaisesti vaikkakin opinnäytetyöhön meni paljon enemmän aikaa, kuin aluksi oli suunniteltu. Työn tulokset ovat karkean automaattisen käyttöliittymätestauksen pohjan muodossa. Se koostuu kolmesta Selenium-testiskriptiluokasta, joissa jokaisessa on vaihteleva määrä testejä. Testiskriptiluokkien lisäksi mukana on yksi pääluokka, joka sisältää apumetodeja kuten sisäänkirjautuminen sekä Selenium RC:lle välitettäviä tietoja. Näiden lisäksi Hudsoniin on konfiguroitu kaksi tehtävää, Firefox- sekä Internet Explorer -selaimille, jotka voidaan asettaa suoritettavaksi haluttuna ajankohtina. Tehtäviin välitetään Maven-komentojen parametreina tieto siitä, missä tietokoneessa testejä halutaan ajettavan sekä siitä, mitä selainta käytetään.

Suurin osa käyttämistäni ohjelmista olivat ennestään tuntemattomia. Ennestään tuttuja ohjelmia kaikista kuudesta ohjelmasta olivat Selenium IDE sekä Eclipse-kehitysympäristö. Ohjelmistotestaus ennen tätä opinnäytetyötä oli lähinnä tuttu manuaalisen testaamisen kautta. Opinnäytetyö oli aiheeltaan ja uuden oppimisen kannalta antoisa ja kiinnostava, ja en näkisi yhtään hullumpana ideana lähteä kehittämään itseäni enemmän automaattisen testauksen suuntaan.

Edellisessä kappaleessa kuvattuihin teknisiin ongelmiin olisi voinut löytyä ratkaisu aiemmin kehitystyön aikana, mikäli olisin ahkerammin kysynyt apua ongelmiini. Pyrin työssäni ratkaisemaan itse ongelmat, vaikka siihen kuluikin huomattavasti enemmän aikaa kuin avun kanssa. Itse ongelmat ratkaisemalla ehkä opin asiat paremmin.

Kehitystyö olisi voinut ollut tehokkaampaa, jos olisin aloittanut toiminnallisen osan työstä testiskriptien kirjoittamisen sijasta testausympäristön rakentamisesta. Jos oma tietokanta testiskriptejä varten olisi ollut olemassa, olisin voinut kirjoittaa monipuolisempia testiskriptejä. Näiden asioiden oivaltaminen olisi toisaalta vaatinut aiempaa kokemusta automaattisesta ohjelmistotestauksesta sekä Selenium RC:stä ja testiskripteistä. Syy miksi lähdin testiskriptien tekemisestä liikkeelle oli se, että se oli ainut entuudestaan tuttu asia ja halusin saada jonkinlaista näkyvää tulosta aikaiseksi. Ennen toiminnallista vaihetta olin käyttänyt huomattavasti aikaa aiheen perehtymiseen eikä näkyvää tulosta opinnäytetyöhön vielä ollut syntynyt.

Opinnäytetyön suunnitelman näkökulmasta ajatellen työ on muuttunut paljon siitä, mitä aluksi suunnittelin ja kuvittelin opinnäytetyön olevan. Aluksi toiminnalliseen työhön piti myös kuulua selainkäyttöliittämän testien kehittämisen lisäksi myös Java-kielellä kirjoitetun lääketieteelliseen kuvantamiseen käytetty työkalu. Toiminnallisen työn tavoita alkoi vasta loppupuolella tarkentua minulle kunnolla, sillä en ollut koskaan täysin varma siitä, mitä toiminnallisesta osasta olisi saada testiskriptien lisäksi aikaiseksi. Työmäärältään opinnäytetyö oli arvioitua paljon laajempi, johtuen siitä, ettei automaattinen testaus ja suurin osa työkaluista olleet entuudestaan millään tavalla tuttuja.

Olen opinnäytetyön lopputulokseen tyytyväinen vaikkakin tuntuu, että toiminnallinen vaihe työstä on edelleen kesken. Nyt kun pohja automaattiselle toiminnalliselle testaukselle on valmis, voisi testiskriptejä lähteä kunnolla kehittämään eteenpäin. Olisi myös mielenkiintoista tutustua tarkemmin Selenium Gridiin ja siihen, minkälaisia vaikutuksia sillä olisi testien suoritusaikaan. Säästettäköön Selenium Grid aiheeksi johonkin toiseen opinnäytetyöhön.

LÄHTEET

Aaltonen, K. 2009. Ohjelmistotestauksen periaatteet. hakupäivä 24.10.2009, http://users.evtek.fi/~karita/2007-2ITProgDAP07S/Testaus/

Dyer, D. 2008. Why are you still not using Hudson. Hakupäivä 4.3.2010, http://blog.uncommons.org/2008/05/09/why-are-you-still-not-using-hudson/

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

Dustin, E, Rashka J & Paul, J. 1999. Automated software testing : introduction, management and performance. John Wiley & Sons.

Haikala, I. & Pyhäjärvi, M. 2009. Ohjelmistotuotanto. Helsinki: Talentum

Hetzel, B & Hetzel, W. 1988. The complete guide to software testing Second Edition. New York: John Wiley & Sons.

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

Kawaguchi, K & Harder, A. 2010. Hudson documentation : How to install plugins. Hakupäivä 4.3.2010, http://wiki.hudson-ci.org/display/HUDSON/Plugins

Kyllönen, T. 2008. Ohjelmistotestauksen kehittäminen ja parantaminen. Joensuun yliopisto. Tietojenkäsittelytiede. Pro gradu –tutkielma

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

Redmond, E. 2006. The Maven 2 POM demystified. Hakupäivä 4.3.2010,

http://www.javaworld.com/javaworld/jw-05-2006/jw-0529-maven.html?page=1

Selenium project 2010a. Introducing Selenium. Hakupäivä 25.2.2010, http://seleniumhq.org/docs/01_introducing_selenium.html

Selenium project 2010b. Selenium RC : reporting results. Hakupäivä 3.3.2010, http://seleniumhq.org/docs/05_selenium_rc.html#reporting-results

Selenium project 2010c. Test design considerations : UI mapping. Hakupäivä 13.4.2010, http://seleniumhq.org/docs/06_test_design_considerations.html#ui-mapping

Selenium project 2010d. Selenium commands : assertion or verification. Hakupäivä 23.3.2010, http://seleniumhq.org/docs/04_selenese_commands.html#assertion-or-verification

SearchSoftwareQuality 2010. What is JUnit. Hakupäivä 3.3.2010, http://searchsoftwarequality.techtarget.com/sDefinition/0,,sid92_gci1249188,00.html

Spagettikoodi 2009. Maven rakkautta ensi silmäyksellä. Hakupäivä 20.5.2010 http://spagettikoodi.wordpress.com/2009/10/19/maven-rakkautta-ensi-silmayksella/

TestingGeek 2009. Selenium IDE - introduction. Hakupäivä 15.11.2009, http://www.testinggeek.com/index.php/testing-tools/test-execution/97-selenium-ide-introduction