View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Theseus

Jere Valpas

Finder-toiminnanohjausjärjestelmän automatisoitu käyttöliittymätestaus

Opinnäytetyö
Kevät 2012
Tekniikan yksikkö
Tietojenkäsittelyn koulutusohjelma

Seinäjoen ammattikorkeakoulu
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan yksikkö

Koulutusohjelma: Tietojenkäsittelyn koulutusohjelma

Suuntautumisvaihtoehto: Sovellustuotannon suuntautumisvaihtoehto

Tekijä: Jere Valpas

Työn nimi: Finder-toiminnanohjausjärjestelmän automatisoitu käyttöliittymätestaus

Ohjaaja: Markku Lahti

Vuosi: 2012 Sivumäärä: 44 Liitteiden lukumäärä: 0

Opinnäytetyön tavoite oli määrittää ja toteuttaa Finder-toiminnanohjausjärjestelmälle sopiva automatisoitu käyttöliittymän testiratkaisu. Testaamisen automatisointi tehtiin Selenium-testijärjestelmällä. Testijärjestelmä käytti hyväkseen kohdeyritykseltä valmiiksi löytyvän Hudson-ohjelmiston automaatiota.

Työssä käydään läpi ohjelmistotestaamiseen, käyttöliittymätestaamiseen sekä näiden automaatioon liittyvää teoriaa. Testausjärjestelmän toteutuksessa käytettiin hyväksi JUnit-, Selenium WebDriver-, Hudson- ja NetBeans IDE työkaluja.

Opinnäytetyön toteutusosuuden tuloksena saatiin määritettyä ja rakennettua toimiva kokonaisuus käyttöliittymän automatisoidulle toiminnalliselle testaamiselle. Testit voidaan ajaa ennalta määrättyinä ajankohtina, muun muassa kellonajan mukaan tai kun sivuston koodissa havaitaan muutos. Testit testaavat sivuston toiminnallista puolta, kuten napin painamisen aiheuttamaa toimintaa. Testitulokset näytetään helposti tulkittavana graafisena esityksenä. Työ sisältää esimerkkejä joiden pohjalta uusien testien luominen on helpompaa.

Avainsanat: testaus, testausautomaatio, selenium, junit

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Faculty: School of Technology

Degree Programme: Business Information Technology

Specialisation: Software Development

Author: Jere Valpas

Title of thesis: Finder ERP- system’s automated user interface testing

Supervisor: Markku Lahti

Year: 2012 Number of pages: 44 Number of appendices: 0

The goal of this thesis was to define and implement an automated user interface testing solution for the Finder ERP system. Test automation was realized through the use of the Selenium testing system. Selenium tests were executed using the Hudson continuous integration system.

In the theoretical part of software testing, user interface testing and test automation are presented. The tools used for the test implementation were JUnit, Selenium WebDriver, Hudson and NetBeans IDE.

The result of the thesis is a solution on which automated functional tests can be created and executed on. The tests can be configured to be executed at a certain time of day, or if a change has been made to the website’s code. These tests are aimed at testing the website’s functionality. The test results are shown as an easily readable graphical presentation. There are examples included for easier test creation.

Keywords: software testing, test automation, Selenium, JUnit

SISÄLTÖ

OPINNÄYTETYÖN TIIVISTELMÄ 2
THESIS ABSTRACT 3
SISÄLTÖ 4
KÄYTETYT TERMIT JA LYHENTEET 6
1 JOHDANTO 7
1.1 Työn tausta 7
1.2 Työn tavoite 7
1.3 Yritysesittely 8
1.4 Rajaus 8
1.5 Työn rakenne 9
2 OHJELMATESTAUS 10
2.1 Yleistä testauksesta 10
2.2 Käyttöliittymätestaus 14
2.3 Testauksen automatisointi 17
3 FINDER-TOIMINNANOHJAUSJÄRJESTELMÄ 20
3.1 Kuvaus nykyisestä järjestelmästä 20
3.2 Ongelmat ja haasteet 20
3.3 Tavoitteet ja vaatimukset 21
4 SUUNNITTELU 22
4.1 Käyttöliittymän testausjärjestelmät 22
4.2 Järjestelmän valinta 23
4.2.1 Selenium 23
4.2.2 JUnit 24
4.2.3 Hudson 25
4.2.4 Maven 26
4.3 Testaussuunnitelma 26
5 TOTEUTUS 28
5.1 Järjestelmän käyttöönotto 28

5.1.1 Testien ajaminen..28
5.1.2 Testitulosten käsittely...32
5.2 Testien toteutus 33
6 ARVIOINTI...42
7 LÄHTEET..43

KÄYTETYT TERMIT JA LYHENTEET

DNS Domain Name Service (DNS) on internetin nimipalvelujärjestelmä, joka muuntaa verkkotunnuksia IP-osoitteiksi.

Jatkuva integraatio Jatkuvan integraation perusajatuksena on liittää ohjelmistoon tehdyt muutokset kehitettävään kokonaisuuteen mahdollisimman aikaisessa vaiheessa.

Toiminnanohjausjärjestelmä Yrityksen tietojärjestelmä, joka integroi eri toimintoja, esimerkiksi tuotantoa, jakelua, varastonhallintaa, laskutusta ja kirjanpitoa.

1 JOHDANTO

1.1 Työn tausta

Finder on Barona Group Oy:n käyttämä toiminnanohjausjärjestelmä. Järjestelmään kirjataan työpaikkojen, työnhakijoiden ja työntekijöiden tiedot. Baronan konsultit käyvät läpi työnhakijoiden tiedot ja niiden perusteella valitsevat sopivimmat ehdokkaat avoimiin työpaikkoihin.

Finder-järjestelmää ylläpitää Barona Group Oy:n oma kehitystiimi. Uusia ominaisuuksia toivotaan lisää. Kehitystiimi pitää huolen toivotuimpien ja tarvituimpien ominaisuuksien lisäämisestä järjestelmään. Jokaisen uuden version jälkeen kehitystiimi suorittaa käyttäjätestausta järjestelmälle varmistaakseen kaikkien ominaisuuksien toimivuuden muutosten myötä.

1.2 Työn tavoite

Työn tavoite on mahdollistaa Finder-järjestelmälle tehtävän käyttäjätestaamisen automatisointi sekä toteuttaa se ainakin osittain. Tämä työ pyrkii aluksi tutkimaan Finder-järjestelmän automatisointiin vaadittuja työkaluja sekä testausteoriaa. Mahdollisista työkaluista valitaan järjestelmälle sopivin vaihtoehto. Työn myöhemmissä vaiheissa valitut testaustyökalut ja teoria otetaan käyttöön ja Finder-järjestelmälle kehitetään automatisoitu testiratkaisu. Testiratkaisun pohjalle koodataan joitakin automatisoituja esimerkkitestejä. Valmiin järjestelmän ja esimerkkitestien pohjalta Finder-kehitystiimin on mahdollista lisätä omia testejään tarpeen mukaan.

1.3 Yritysesittely

Barona Group Oy on koko maan alueella toimiva henkilöstöpalveluyritys. Yrityksen päätoimipaikka sijaitsee Helsingissä, jonka lisäksi sillä on 13 muuta toimipaikkaa. Yrityksen vuoden 2010 liikevaihto oli noin 2,8 miljoonaa euroa. Yrityksen henkilöstömäärä oli tuolloin 23 työntekijää (Finder Yritystieto, [viitattu 11.04.2012].) Barona työllistää vuosittain yli 8000 henkilöä hoiva-, IT-, logistiikka-, rakennus-, teollisuus- ja toimistoaloilla (Sponsor Capital Oy, [viitattu 11.04.2012]).

Barona ottaa vastaan työpaikkailmoituksia työantajilta. Ne lisätään Finder-järjestelmään, josta sivustot kuten Monster hakevat tiedot omille sivuilleen. Baronan konsultit käsittelevät saamansa työhakemukset työnantajan antamien kriteerien mukaan. Parhaiten työhön sopivat henkilöt välitetään eteenpäin ilmoituksen jättäjälle.

1.4 Rajaus

Työ rajataan testausteorian tutkimiseen, testausjärjestelmän suunnitteluun sekä järjestelmän käyttöönottoon. Testauksessa käytetyt käyttöjärjestelmät rajataan Windowsiin, Linuxiin ja Maciin. Internetselaimista mukaan valitaan Internet Explorer, Chrome ja Firefox. Työssä otetaan huomioon järjestelmän mahdolliset pitkän aikavälin kehitysmahdollisuudet. Testien kirjoittaminen rajautuu muutamaan esimerkkitestiin.

1.5 Työn rakenne

Työ aloitetaan käymällä läpi testaukseen liittyvää teoriaa. Teoriaosuus on jaettu kolmeen osaan: yleiseen testausteoriaan, käyttöliittymätestaukseen sekä testauksen automatisointiin. Nämä osiot vastaavat omalla kohdallaan kysymyksiin mitä, miksi, milloin ja miten.

Testausteoriasta siirrytään tutkimaan tämän työn kohdetta, Finder-toiminnanohjausjärjestelmää. Kappaleista selviää tarkemmin mikä Finder on, mitä haasteita ja ongelmia Finderin automatisoinnissa tulee ottaa huomioon sekä järjestelmän kohdalle asetetut testitavoitteet ja vaatimukset.

Seuraavaksi suunnitellaan työssä käytetty järjestelmä teoriatiedon ja vaatimusten perusteella. Näiden perusteella valitaan ohjelmistot, joita työssä tullaan käyttämään. Lopuksi järjestelmälle luodaan testaussuunnitelma läpikäydyn teorian ja valittujen ohjelmistojen perusteella.

Suunnittelun jälkeen siirrytään toteutukseen. Kappaleessa näytetään kuinka läpikäydyn teorian sekä valittujen ohjelmistojen mukainen järjestelmäkokonaisuus toteutetaan.

Lopuksi työ tulee sisältämään henkilökohtaisen arvion työn onnistumisesta sekä työn teettäjän mielipiteen työn onnistumisesta.

2 OHJELMATESTAUS

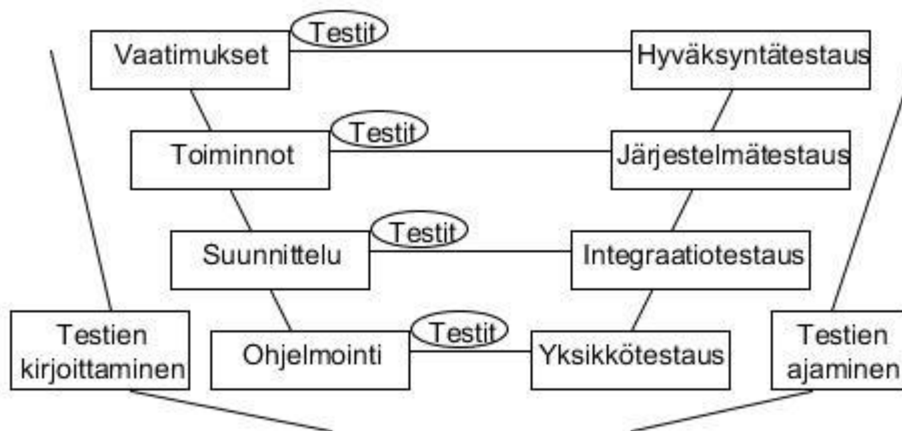
2.1 Yleistä testauksesta

Sanalle ”testaus” löytyy useita määritelmiä. Esimerkkejä ohjelmistotestausta käsittelevistä teoksista:

- Testaus on prosessi jossa ohjelmistoa käytetään tarkoituksena löytää siitä virheitä (Myers 1979, 5).
- Ohjelmistotestauksen tarkoitus on löytää ohjelmasta virheitä (Kaner, Falk & Nguyen 1993, 27).
- Testaus tarkoittaa mitä vain toimintoa jolla pyritään arvioimaan saavuttaako ohjelma tai ohjelmisto vaaditut tulokset (Hetzel 1993, 4).

Testausta pidetään usein prosessina joka voidaan suorittaa vasta kun ohjelmisto on valmis. Tällöin oletetaan, että testauksella tarkoitetaan vain testien ajamista. Tietenkään testejä ei voida ajaa mikäli testattavaa ohjelmistoa ei ole olemassa, mutta testaus pitää sisällään muutakin kuin testien ajamisen. (Fewster & Graham 1999, 6.)

Sovelluskehityksessä käytettävä V-malli osoittaa milloin testaustoimintaa tulisi suorittaa. V-malli osoittaa kuinka jokaiselle kehitysvaiheelle on olemassa vastaava testaustoiminta (ks. kuva 1). Samaa periaatetta voidaan käyttää kehityksen jokaisessa vaiheessa ohjelmiston elinkaaresta riippumatta. Tärkein tekijä V-mallin onnistuneessa käytössä on testien suunnittelun ajankohta. Testaustoiminta löytää virheitä alueelta jolle testi on suunniteltu. Esimerkiksi hyväksyntätestaus löytää virheitä vaatimuksista ja järjestelmätestaus löytää virheitä toiminnoista. Testien suunnittelun jädessä kehitysvaiheen loppuun virheet löydetään juuri ennen testien ajamista. Näin myöhäisessä vaiheessa niiden korjaaminen on vaikeampaa ja tulee kalliimmaksi. (Fewter & Graham 1999, 6.)



Kuva 1: V-malli esittää aikaista testausmallia (Fewster & Graham 1999)

Vaikka testejä ei voida kirjoittaa ennen varsinaista ohjelmakoodia, voidaan ne koodata aikaisessa vaiheessa. Testit ajetaan vastakkaisessa suunnassa niiden kirjoittamisjärjestykseen verrattuna. Esimerkiksi yksikkötestit koodataan viimeisenä, mutta ne ajetaan ensimmäisenä. (Fewter & Graham 1999, 7.)

Kanerin, Bachin ja Pettichordin (2002, 32) mukaan testaus voidaan aina jakaa viiteen osa-alueeseen:

Testaajat. Tämä vastaa kysymykseen ’kuka testaa’. Käytetään esimerkkinä käyttäjätestausta. Käyttäjätestauksen suorittavat tuotteen kohderyhmään kuuluvat henkilöt.

Kattavuus. Tämä vastaa kysymykseen ’mitä testataan’. Käytetään esimerkkinä toiminnallista testaamista. Testaaja testaa kaikki ennalta määritellyt toiminnot.

Mahdolliset ongelmat. Tämä vastaa kysymykseen ’miksi testataan’ eli mitä virheitä testaamisella etsitään. Käytetään esimerkkinä testaamista äärilukujen aiheuttamien virheiden varalta.

Testaustavat. Tämä vastaa kysymykseen ’miten testataan’. Käytetään esimerkkinä tutkivaa testaamista.

Testitulosten arviointi. Tämä vastaa kysymykseen ’kuinka tiedetään onnistuiko vai epäonnistuiko testi’. Käytetään esimerkkinä saadun tuloksen vertaamista tunnettuun hyväksyttyyn tulokseen.

Testaustehtävät annetaan testaajalle usein yksiulotteisina. Tästä huolimatta testaus suoritetaan aina käyttäen edellä mainittuja viittä osa-aluetta. Testaajaa saatetaan pyytää suorittamaan toiminnallista testaamista. Tässä tapauksessa tehtävänanto määrittelee mitä osiota testataan. Tämän lisäksi testaajan täytyy päättää kuka suorittaa testaamisen, millaisia virheitä testaamisella etsitään, miten testaus suoritetaan sekä miten tiedetään onnistuiko testi vai ei. (Kaner ym. 2002, 33.)

Ohjelmiston täydellinen testaus on käytännössä mahdotonta. Otetaan esimerkkinä yksinkertainen ohjelma, joka summaa kaksi lukua jotka sisältävät joko yhden tai kaksi numeroa. Ensin ohjelmassa on testattava kaikki mahdolliset hyväksytyt luvut. Jopa näin yksinkertaisessa sovelluksessa erilaisia testattavia yhdistelmiä on 39601 kappaletta. (Kaner ym. 1993, 20.)

Täydellisen testauskattavuuden saavuttamiseksi ei kuitenkaan riitä pelkkien hyväksyttyjen lukujen testaus. Ohjelma tulisi testata myös kaikkien hylättyjen merkkijonojen osalta. Käytännössä tämä tarkoittaa jokaista näppäimistöltä löytyvää merkkiyhdistelmää. (Kaner ym. 1993, 20.)

Mikäli sovellus sallii syötettyjen tietojen muokkaamisen tulee myös tämä osa-alue sovelluksesta testata. Kokeillaan toimiiko ohjelma jos kaikkien lukuyhdistelmien kohdalla suoritetaan muutoksia. Entä jos syötettä muokataan useaan kertaan? Kuinka monen muokkaamiskerran jälkeen voidaan olla varmoja ettei tämä aiheuta virheitä? (Kaner ym. 1993, 21.)

Testaamisessa tulee myös ottaa huomioon ajoitus. Mitä tapahtuu, jos uutta tietoa syötetään niin lyhyessä ajassa ettei edellisen syötteen tulosta ole vielä

päivitetty ruudulle? Mitä tapahtuu, jos aikaa lyhennetään vielä entisestään? Nämä kaikki ajoitukset tulee testata kaikilla mahdollisilla syötteillä täydellisyyden saavuttamiseksi. (Kaner ym. 1993, 21.)

Ohjelman toimivuutta ei voida osoittaa käyttämällä logiikkaa. Tietokone toimii loogisella pohjalla. Ohjelmat ilmaistaan tarkasti määritellyllä kielellä. Mikäli ohjelma on hyvin jäsennetty tulisi pelkän ohjelman sisällön perusteella pystyä todistamaan mitä tapahtuu jokaisessa mahdollisessa tilanteessa. (Kaner ym. 1993, 24.)

Tämä todistaa vain ohjelman toimivuuden määritellyissä olosuhteissa. Ohjelman kirjoittajan on vaikea ottaa huomioon jokaista ulkoista olosuhdetta, erityisesti tietokoneohjelmissa jotka ovat maailmanlaajuisessa levityksessä. Jokin tietty ohjelmistoyhdistelmä saattaa aiheuttaa ennalta-arvaamattomia virheitä, joita ei kuuluisi tapahtua ohjelmistoa loogisesti tutkittaessa. Erilaisten ohjelmistojen lisäksi mukaan tulevat laitteistovaihtoehdot. Jokainen laitteisto on uniikki. (Kaner ym. 1993, 24.)

Testaajan tavoite ei ole osoittaa, että ohjelmisto toimii oikein. Yritykset rahoittavat testaamista, koska ohjelmistojen kohdalla perusolettamuksena on, että ohjelmisto ei toimi oikein. Testaajan tehtävänä on löytää ohjelmiston osa-alueet, jotka eivät toimi. Suurin osa ohjelmoijista löytää ja korjaa yli 99 % virheistään ennen ohjelmiston julkaisua. Testaajan tehtävänä on löytää puuttuva 1 % virheistä. (Kaner ym. 1993, 26–27.)

2.2 Käyttöliittymätestaus

Dumas ja Redish (1999, 22) vakuuttavat jokaisen käytettävyystestin jakavan viisi ominaisarvoa:

- Päälimmäinen tavoite on parantaa tuotteen käytettävyyttä. Jokaisella testillä on myös tarkemmin määritelty tarkoitus ja tavoite, joka ilmaistaan testiä suunnitellessa.
- Testiin osallistuvat henkilöt edustavat oikeita käyttäjiä.
- Osallistujat suorittavat oikeita tehtäviä.
- Testaaja seuraa ja tallentaa kaiken mitä osallistujat tekevät ja sanovat.
- Testaaja analysoi osallistujista saadun tiedon, määrittää järjestelmässä esiintyneet ongelmat ja ehdottaa muutoksia näiden ongelmien korjaamiseksi.

Päälimmäinen tavoite on parantaa testattavan tuotteen käytettävyyttä. Toinen tavoite on parantaa tuotteiden suunnitteluun ja kehitykseen liittyvää prosessia. Näin vältytään törmäämästä samoihin ongelmiin muissa tuotteissa. Tämä ominaisuus erottaa käytettävyystestauksen tieteellisestä testauksesta, laadunvarmistuksesta ja toiminnallisesta testauksesta. (Dumas & Redish 1999, 22.)

Yleinen testaustavoite, eli tuotteen käytettävyyden parantaminen, tulee sisältämään tarkennettuja tavoitteita tai huolia jotka vaihtelevat testistä toiseen. Testaaja saattaa esimerkiksi olla huolissaan sivun navigoinnin helppokäyttöisyydestä. Tätä huolenaihetta voidaan käsitellä luomalla interaktiivinen prototyyppi menusta ennen projektin koodaamista. Nämä tarkennetut tavoitteet ja huolenaiheet auttavat päättämään, ketkä käyttäjistä sopivat testauksen eri vaiheisiin ja osiin. (Dumas & Redish 1999, 22-23.)

Testeistä saatua tietoa tullaan käyttämään tuotteen kehityksen parantamisessa. Tavoitteena on varmistaa tuotten hyödyllisyys kohdeyleisölleen, sen käytön oppimisen helppous, sen vaikutus ihmisten suorituskykyyn sekä sen helppokäyttöisyys. (Rubin & Chisnell 2008, 22.)

Kirjallisuus on täynnä erilaisia testausmetodeja joista jokaisella on hieman eri päämäärä. Tuoteen elinkaaren jokaiseen vaiheeseen sopii jonkinlainen käytettävyystestausmenetelmä. Erilaisia termejä käytetään usein kuvaamaan samanlaisia testausmenetelmiä. Tämä aiheuttaa usein hämmennystä. Tämän vuoksi testeistä päätettäessä ja niitä suunniteltaessa on hyvä käyttää tuotteen elinkaarta lähteenä (ks. kuva 2). (Rubin & Chisnell 2008, 27-28.)

Kuva 2: Käytettävyystestaus tuotteen elinkaaren aikana (Rubin & Chisnell 2008).

Toinen hyvä syy käyttää tuotteen elinkaarta testausprosessin lähteenä on toistuvan kehityksen tehokkuus. Käytettävyystestaus on tehokkaimmillaan toistuvan kehityksen prosessiin yhdistettynä. Toistuva kehityksen, testauksen ja arvioinnin jakso koko tuotteen kehityksen ajan antaa suurimman todennäköisyyden luoda toimiva kokonaisuus. Vaikka tärkeitä virheitä ja ongelmia jäisi huomaamatta kehityksen ensimmäisellä kierroksella, antaa toinen kehityskierros uuden mahdollisuuden löytää nämä virheet. (Rubin & Chisnell 2008, 28.)

Käytettävyystestaamista voidaan käyttää kaikentyyppisten käyttöliittymien testaamiseen. Mikä tahansa tuote jota ihminen käyttää sisältää käyttöliittymän. Luovuudella mille tahansa käyttöliittymälle voidaan kehittää käytettävyystesti. Käytetään esimerkkinä hissiä. Hissin napit muodostavat hissin käyttöliittymän. Miten suorittaisit käytettävyystestausta hissin käyttöliittymälle? Kuinka siitä löydetään virheet ennen hissin kokoamista? Tällaisen käyttöliittymän testaaminen voidaan hoitaa ennen tuotteen kokoamista. Käyttöliittymää voidaan testata esimerkiksi tietokoneella kasatulla mallilla, tai jopa paperille piirretyllä esimerkillä. Malli annetaan testaajien testattavaksi ja käyttäjän toimia seurataan vierestä. Kehitysvaiheesta ja nykyisen tuotteen huolenaiheista riippuen saatat haluta keskittyä käyttöliittymän tiettyihin osa-alueisiin. Tästä huolimatta on myös otettava huomioon osien yhteentoimivuus. (Dumas & Redish 1999, 28.)

2.3 Testauksen automatisointi

Ohjelmiston toimivuus on hyvä varmistaa testaamalla sitä sille aiotussa ympäristössä. Testauksen automatisointi pystyy laskemaan testaukseen kuluvaa aikaa ja nostamaan testaustehokkuutta huomattavasti. Manuaalisesti suoritettuna tunteja vievät testit voivat automatisoituina kestää vain minuutteja. Automatisoituja testejä voidaan toistaa täysin samoilla asetuksilla useita kertoja. Tätä ei voida taata manuaaliselle testaukselle. (Fewster & Graham 1999, 3.)

Testaus ja testauksen automatisointi eivät tarkoita samaa asiaa. Testaus vaatii taitoa valita hyödyllisimmät testattavat ominaisuudet kaikista mahdollisista ominaisuuksista. Testatessa tulee myös ottaa huomioon kuinka tehokkaasti yksi testaustehtävä käy ominaisuuden läpi. Mitä tehokkaampia testaustehtävät ovat, sitä vähemmän niitä kokonaisuudessaan vaaditaan. (Fewster & Graham 1999, 4.)

Testauksen automatisointi vaatii taitoa valita luoduista testaustehtävistä vain ne joiden automatisoinnista hyödytään huomattavasti. Testauksen automatisoinnin kustannukset ovat korkeammat kuin yksittäisten testien ajaminen. Testaustehtäviä saattaa olla kymmeniä tai satoja mutta jokaisen tehtävän automatisointi olisi resurssien tuhlaamista. Mitä kauemmin samaa testiä voidaan ajaa automatisoidusti, sitä enemmän säästöjä automatisoinnista syntyy. (Fewster & Graham 1999, 5.)

Testit tulee suunnitella ennen automatisointipäätöksen tekemistä. Tämä ehkäisee automatisoimasta testejä, jotka on helppo automatisoida mutta jotka ovat heikkoja löytämään virheitä. (Kaner ym. 2002, 93.)

Automatisoidut testit tulee suunnitella toisella tavalla kuin manuaaliset testit. Automatisoiduissa testeissä tulee ottaa huomioon mihin tietokone kykenee. Testi jossa samaa prosessia toistetaan yli tuhannelle eri tiedostolle on erin-

omainen testi automatisoida. Manuaalista testiä suunnitteleva henkilö tuskin toteuttaa vastaavanlaista skenaariota. (Kaner ym. 2002, 94.)

Seuraavaksi joitain Kanerin ym. (2002, 96) esittämiä esimerkkejä skenaarioista, joissa testaamisen automatisointi on hyödyllistä tai lähes pakollista:

Kuormitustestaus. Manuaalisella testaamisella on lähes mahdotonta selvittää mitä tapahtuu jos 200 tai 2000 käyttäjää yrittävät käyttää ohjelmaa samanaikaisesti.

Suorituskykytestaus. Automatisoidut testit voi helposti asettaa mittaamaan tapahtumien välisiä vasteaikoja. Tutkimalla näitä tuloksia voidaan nähdä heikkeneekö tuotteen suorityskyky ajan myötä.

Kokoonpanon testaus. Ohjelmistojen tulee usein toimia useilla eri alustoilla sekä eri asetuksilla. Automaatiolla pystytään testaamaan mahdollisimman monta eri asetelmaa nopeasti.

Kestävyyden testaus. Mitä tapahtuu, kun ohjelmistoa käytetään päiviä tai viikkoja ilman uudelleenkäynnistyksiä? Näitä ongelmia ei välttämättä huomata lyhyen aikavälin testaamisella. Automaatio auttaa testaamaan tällaisia pitkäkestoisia skenaarioita.

Rinnakkaisuuksista johtuvat ongelmat. Jotkin virheet esiintyvät vain tietyissä olosuhteissa. Olosuhde saattaa muodostua kun kahden säikeen tai prosessin samanaikainen ajoitus varaa saman resurssin aiheuttaen kilpailutilanteen, joka johtaa virheeseen. Automaatiosta on paljon apua kun pyritään ajamaan samoja toimintoja vaihtelevilla ajoituksilla.

Yhdistelmävirheet. Jotkin virheet syntyvät usean eri toiminnon yhdistelmistä. Mahdollisimman monen yhdistelmän läpikäyminen ilman automaatiota on työlästä.

Nämä lähestymistavat keskittyvät käyttämään automaatiota luomaan uusia testejä tai toistamaan vanhoja testejä tavoilla, jotka auttavat löytämään virhei-

tä. Mikään näistä testeistä ei ole yksinkertainen toteuttaa. Testaaja saattaa joutua muuttamaan eri osia testaustyökaluista automatisoinnille sopivaksi. (Kaner ym. 2002, 96.)

Testiautomaatiossa kohdataan usein ongelmia. Fewster ja Graham (1999, 10-12) listaavat seuraavat ongelmat:

- Epärealistiset odotukset. Valmistajat usein liioittelevat työkalujen vaikutusta ja saattavat vähätellä työkalun käytön vaatimaa työtä. Jos yrityksen johdon odotukset ovat liian optimistiset, työkalujen antama realistinen etu ei saavuta odotuksia. Toistettujen testien todennäköisyys löytää uusia virheitä on paljon pienempi kuin ensimmäistä kertaa ajettavan testin todennäköisyys. Poikkeus tähän on tilanne jossa muualle järjestelmään on tehty muutoksia jotka vaikuttavat testattavaan osaan.

- Huonot testikäytännöt. Jos olemassa olevat testit ovat huonosti organisoituja ja heikkoja löytämään virheitä, ei niiden automatisointi ole suositeltavaa. Tässä tilanteessa kannattaa keskittyä ensin vanhojen testien laadun parantamiseen.

- Valheellinen turvallisuudentunne. Ohjelma ei ole bugiton vaikka automatisoidut testit eivät löytäisi siitä virheitä. Testit saattavat olla epätäydellisiä tai virheellisiä.

- Automaattisten testien ylläpito. Järjestelmän muuttuessa tulee myös testit päivittää. Testien päivitysten ollessa työläämpää kuin niiden suorittaminen manuaalisesti hylätään testiautomaatio.

- Tekniset ongelmat. Kolmannen osapuolen ostetut testityökalut eivät ole immuuneja bugeille. Joidenkin testityökalujen testaus saattaa olla rajallista. Työkalujen yhteistyö muiden järjestelmästä löytyvien työkalujen kanssa ei välttämättä toimi halutulla tavalla. Järjestelmän ympäristö saattaa vaihtua niin suurella nopeudella etteivät työkalujen ylläpitäjät pysy muutosten mukana.

3 FINDER-TOIMINNANOHJAUSJÄRJESTELMÄ

3.1 Kuvaus nykyisestä järjestelmästä

Finder on Barona Group Oy:n käyttämä toiminnanohjausjärjestelmä. Järjestelmään kirjataan työpaikkojen, työnhakijoiden ja työntekijöiden tiedot. Baronan konsultit käyvät läpi työnhakijoiden tiedot ja niiden perusteella valitsevat sopivimmat ehdokkaat avoimiin työpaikkoihin.

Järjestelmään tulleiden muutosten jälkeen suoritetaan manuaalista testausta. Testatessa tarkistetaan perusominaisuuksien toimivuus. Testaukseen on olemassa valmiiksi laadittu ohje, jota seurataan.

3.2 Ongelmat ja haasteet

Nykyisen järjestelmän ongelma on testaamiseen kuluva aika. Kun otetaan huomioon testaamiseen laadittu lista, voitaisiin testaus automatisoida.

Finder-järjestelmä käyttää dynaamisesti luotuja sivuelementtejä. Yksi testauksen automatisoinnin haasteista on näiden dynaamisten elementtien käsittely.

Testijärjestelmä tulee vaatimaan erillisen testikoneen. Testikone vaaditaan lähinnä Windows-käyttöjärjestelmän testaamiseen. Kaikista käytössä olevista kehityskoneista löytyy joko Mac- tai Linux-käyttöjärjestelmä. Windows-testikone mahdollistaa lähes kaikkien selainten testaamisen.

3.3 Tavoitteet ja vaatimukset

Työn tavoitteena on saada luotua toimiva pohja automatisoiduille testeille. Testiratkaisu mahdollistaa automatisoitujen testien koodaamisen järjestelmälle. Uusien ominaisuuksien myötä koodataan uusia testejä, jotka jokaisella ajokerralla testaavat kyseiset ominaisuudet.

Jokaisen järjestelmämuutoksen jälkeen tulee pystyä ajamaan käyttöliittymätestit joko omalla koneella tai kehitysympäristöä vasten. Testien perusteella kehittäjä näkee onko mikään perusominaisuuksista hajonnut uusien muutosten myötä.

Testien tulee pyrkiä emuloimaan loppukäyttäjää mahdollisimman hyvin. Perusominaisuudet tulee testata ja virheviestit sekä vahvistusviestit tulee tarkistaa. Järjestelmä ilmoittaa kehittäjälle testeistä, jotka eivät mene läpi.

4 SUUNNITTELU

4.1 Käyttöliittymän testausjärjestelmät

Käyttöliittymätestauksessa käytetyistä järjestelmistä valitsin mukaan verailuun järjestelmät Selenium, Watir ja Sahi. Jokainen näistä järjestelmistä mahdollistaa käyttöliittymätestaamisen loppukäyttäjän näkökulmasta. Järjestelmät pystyvät kirjoittamaan tekstikenttiin, painamaan nappeja, seuraamaan linkkejä, sulkemaan ikkunoita ja tekemään kaikkea mitä loppukäyttäjä voi tehdä.

Järjestelmä	Selenium	Watir	Sahi
Tuetut kielet			
Java	X		X
C#	X		
Python	X		
Ruby	X	X	X
Perl	X		
Php	X		X
Käyttöjärjestelmät			
Windows	X	X	X
Linux	X	X	X
Mac	X	X	X
Selaimet			
Firefox	X	X	X
Internet Explorer	X	X	X
Chrome	X	X	X
Opera	X	X	X
Safari		X	X
Versiot			
Ilmainen	X	X	X
Maksullinen			X

Kuva 3. Testausjärjestelmien ominaisuudet

4.2 Järjestelmän valinta

Finder-järjestelmä toimii Java-pohjaisesti. Testitiedostot tulevat sijaitsemaan samoilla palvelimilla kuin Finder. Yhteensopivuuden takaamiseksi on suositeltavaa, että testijärjestelmä tukee Javaa. Testijärjestelmän tulee myös tukea Windows-, Mac- ja Linux-pohjaisia käyttöjärjestelmiä. Suosituimpien selaimien tuki on myös tärkeää. Näiden kriteerien perusteella Selenium WebDrive ja Sahi nousevat mahdollisiksi testausjärjestelmiksi (ks. kuva 3). Sahin kannalta haitallista on Pro-version maksullisuus. Maksullisen version hinnaksi tulee 495 $, eli noin 380 €. Selenium tarjoaa kaikki ominaisuutensa ilmaiseksi. Seleniumilla on myös edellä mainituista järjestelmistä suurin käyttäjä ja kehittäjäkunta.

Käytettäväksi testausjärjestelmäksi valittiin Selenium.

4.2.1 Selenium

Selenium-järjestelmä mahdollistaa testien luomisen kahdella eri tavalla. Ensimmäinen tapa on Selenium IDE. Se on Firefoxiin asennettava laajennus joka mahdollistaa testien nauhoittamisen ja toistamisen selaimella. Selenium IDE:ä käytettäessä testaajan ei tarvitse koodata. (Selenium documentation, [viitattu 10.04.2012].)

Toinen tapa testien luomiseen on testiskriptien kirjoittaminen (ks. kuva 4). Tällöin suositellaan käytettäväksi Selenium WebDriveria. Skriptikielenä voi käyttää mitä vain Seleniumin tukemaa ohjelmointikieltä. Selenium WebDriver mahdollistaa monimutkaistenkin testien luomisen. Se ei kärsi samoista rajoitteista kuin Selenium IDE. (Selenium documentation, [viitattu 10.04.2012].)

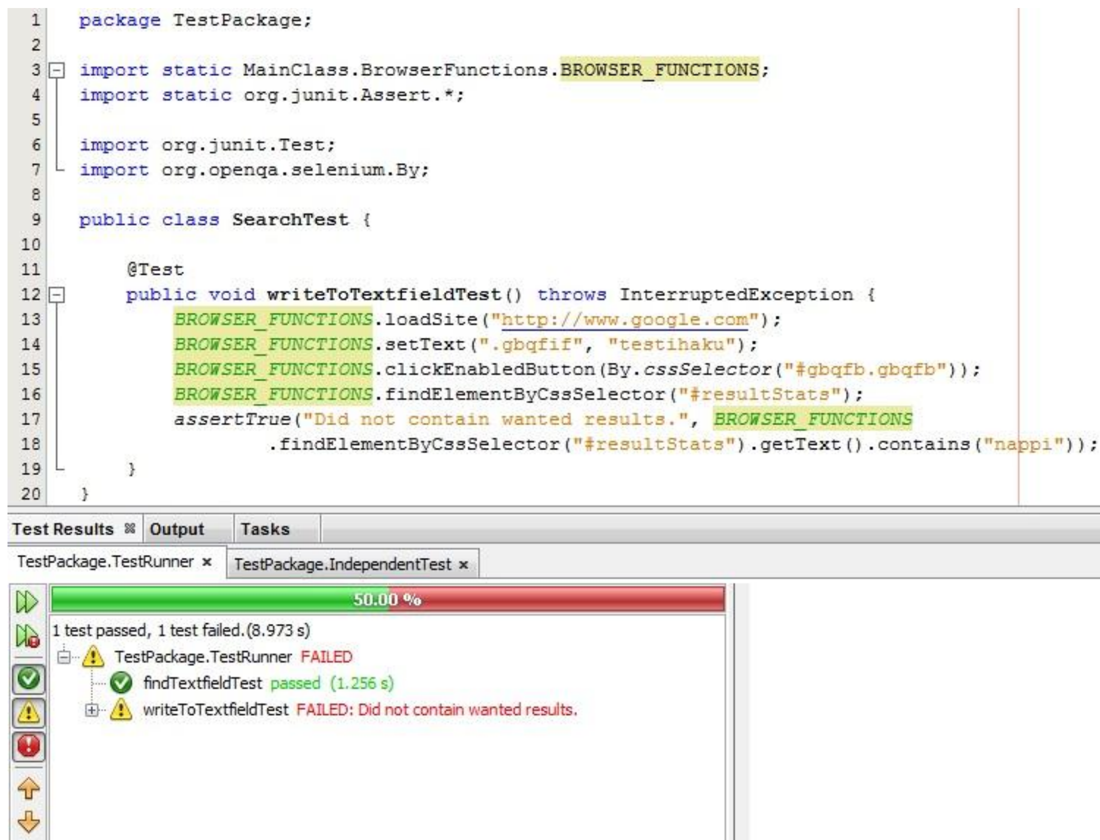
```java
package TestPackage;

import java.net.MalformedURLException;
import java.net.URL;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

public class IndependentTest {

    @Test
    public void IndependentTest() throws MalformedURLException,
                                         InterruptedException {
        DesiredCapabilities capabilities = DesiredCapabilities.firefox();
        RemoteWebDriver driver = new RemoteWebDriver
                (new URL("http://127.0.0.1:4444/wd/hub"), capabilities);
        driver.get("http://www.google.com");
        WebElement element = driver.findElement(By.cssSelector(".gbqfif"));
        element.sendKeys("testihaku");
        element = driver.findElement(By.cssSelector("#gbqfb.gbqfb"));
        element.click();
        driver.close();
    }
}
```

Kuva 4: Esimerkki Selenium scriptistä

4.2.2 JUnit

Selenium selviytyy internetsivuilta löytyvien WebElementtien tarkistelusta, mutta niiden vertailuun vaaditaan toinen sovellus. JUnit on yksikkötestityökalu, jolla voidaan vertailla toivetilaa ja saatua vastausta. Mikäli ne eivät ole keskenään yhtenevät, lähettää JUnit virheen. (JUnit FAQ, [viitattu 10.04.2012].)

Esimerkkikoodissa (ks. kuva 5) näkyvä @Test annotaatio ilmoittaa sitä seuraavan koodin olevan osa JUnit testiä. Myös koodin assertTrue osa kuuluu JUnit:ille. Tämän koodin tarkoituksena on vertailla toivottua tulosta saatuun tulokseen. Kuvasta näkyy kuinka jokin testeistä on epäonnistunut.



Kuva 5: JUnitin käyttö Selenium koodin yhteydessä

4.2.3 Hudson

Hudson on jatkuvan integraation työkalu, joka perustuu avoimeen lähdekoodiin. Se rakentaa ja testaan Java-projekteja tietyin aikavälein varmistaen projektin jatkuvan toiminnan. Virhetilanteissa järjestelmä voi ilmoittaa ongelmista esimerkiksi sähköpostilla (Dyer 2008, [viitattu 10.04.2012].) Selenium-testien suorittajana toimii firmalta jo valmiiksi löytyvä Hudson-kone. Hudson käyttää projektin rakentamiseen Mavenia, jota varten siitä löytyy sisäänrakennettu tuki. Kuva 6 esittää Hudsonia perusnäkymässä.

Kuva 6: Hudson perusnäkymä

4.2.4 Maven

Maven on työkalu, joka huolehtii projektin rakentamisesta sekä siihen liittyvien moduulien lataamisesta projektille määritetyn POM-asetustiedoston perusteella. POM-asetustiedostossa määritellään ulkoiset kirjastot joita halutaan käyttää sekä niiden versiot. Kun Maven ajetaan se lataa määritellyt kirjastot. (What is Maven?, [viitattu 27.05.2012].)

4.3 Testaussuunnitelma

Järjestelmien yhteistoiminta koostuu seuraavista osista:

- Hudson rakentaa projektin Mavenin avulla.
- Maven lataa POM-tiedostossa määritellyt kirjastot.
- Maven suorittaa JUnit yksikkötestit

- JUnit ajaa Selenium testit
- Selenium suorittaa testien määrittelemät toiminnot
- JUnit vertaa testeistä saatuja tuloksia haluttuihin tuloksiin. Tämän jälkeen JUnit lähettää tiedon testien lopputuloksista Hudson-palvelimelle
- Hudson näyttää lopputuloksen käyttäjälle selkeässä ikkunassa joka kertoo suoritettujen sekä epäonnistuneiden testien määrän (ks. kuva 5).

Selenium-testit voi suorittaa joko omalla koneella tai Selenium-palvelinta vasten. Omalla koneella ajettaessa riittää pelkkä Selenium-Java package ja testattava selain. Palvelinta vasten ajettaessa palvelinkone vaatii selenium-standalone-palvelimen sekä testattavan selaimen.

5 TOTEUTUS

5.1 Järjestelmän käyttöönotto

Järjestelmän käyttöönotossa tulee ottaa huomioon sekä testien ajaminen että testitulosten käsittely. Järjestelmä koostuu edellisessä kappaleessa läpikäydyistä ohjelmistoista Selenium, JUnit ja Hudson. Järjestelmän käyttöönotto vaati kaikkien kolmen ohjelmiston asentamista ja määrittämistä.

5.1.1 Testien ajaminen

Voidakseen käsitellä internet selaimia halutulla tavalla Selenium tarvitsee yhteyden Selenium-palvelimelle. Palvelimen IP-osoitteen määritys tapahtuu kuvan 11 osoittamalla tavalla. Palvelinyhteyden ollessa avoinna pystyvät Selenium testit lähettämään komentoja palvelimelle. Palvelin avaa selaimen ja suorittaa määritellyt toiminnot. Palvelimelta nähdään komentokehoitteella kaikki sinne lähetetyt käskyt (ks. kuva 7).

Selvimmän tuloksen saavuttamiseksi sivuston testaus on jaettu Selenium-osiin kuten käyttäjätunnuksen syöttäminen, salasanan syöttäminen ja kirjautumisnapin painaminen. Nämä osat muodostavat kokonaisen testiluokan nimeltään *kirjautuminen*. Testiluokat kuten *rekisteröidy* ja *kirjaudu* muodostavat yhdessä ajettuina kokonaisen testin. Kuva 9 esittää asiaa graafisesti.

Testejä ajettaessa on tärkeää, että testien osat ja testiluokat ajetaan oikeassa järjestyksessä. Esimerkiksi kirjautumistestissä on tärkeää, että käyttäjätunnuksen ja salasanan syöttäminen tapahtuu ennen Kirjaudu-napin painamista. Testiluokkien osalta taas täytyy varmistaa ettei kirjautuminen tapahdu ennen rekisteröitymistä. Oikean järjestyksen varmistamiseksi Selenium-testiluokkien sisälle tehdään oma rakenne testin osien järjestykselle. Mikäli

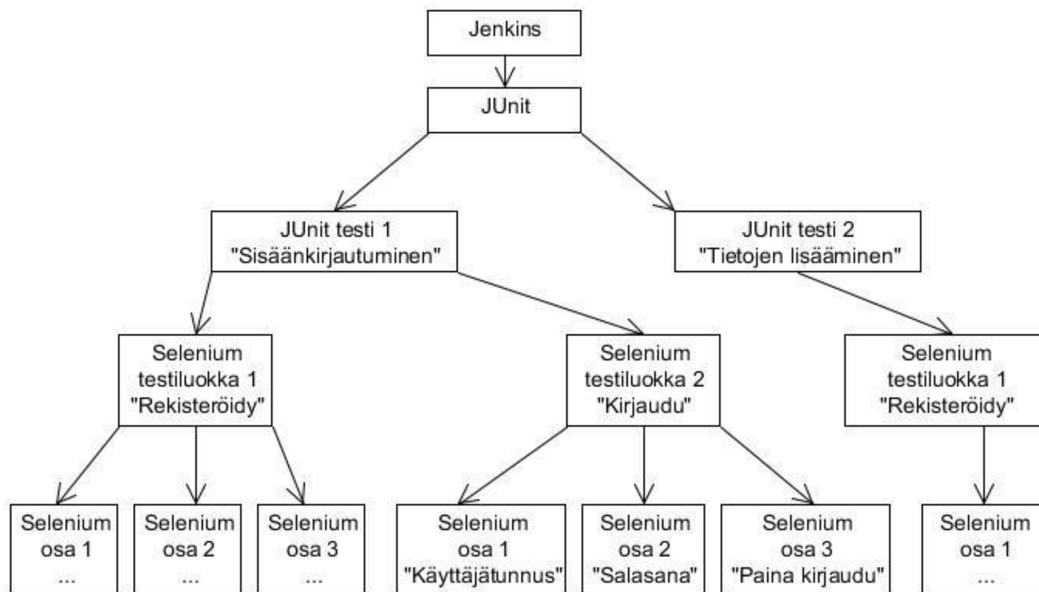
tiedoston sisälle on määritelty useampi kuin yksi JUnit testi, ne voivat suorittua missä järjestyksessä tahansa. Oikean ajojärjestyksen säilyttämiseksi jokainen Selenium-testiluokka sisältää vain yhden JUnit-testin. Näiden JUnit-testien sisällä määritellään Selenium-osien ajojärjestys (ks. kuva 8).

```
INFO: Launching a standalone server
19:35:20.844 INFO - Java: Oracle Corporation 21.0-b17
19:35:20.844 INFO - OS: Windows 7 6.1 amd64
19:35:20.860 INFO - v2.21.0, with Core v2.21.0. Built from revision 16552
19:35:20.953 INFO - RemoteWebDriver instances should connect to: http://127.0.0.
1:4444/wd/hub
19:35:20.953 INFO - Version Jetty/5.1.x
19:35:20.953 INFO - Started HttpContext[/selenium-server/driver,/selenium-server
/driver]
19:35:20.953 INFO - Started HttpContext[/selenium-server,/selenium-server]
19:35:20.953 INFO - Started HttpContext[/,/]
19:35:20.969 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@1be322
43
19:35:20.969 INFO - Started HttpContext[/wd,/wd]
19:35:20.985 INFO - Started SocketListener on 0.0.0.0:4444
19:35:20.985 INFO - Started org.openqa.jetty.jetty.Server@151c2b4
20:55:18.100 INFO - Executing: [new session: {platform=WINDOWS, browserName=fire
fox, version=null}] at URL: /session)
20:55:24.101 INFO - Done: /session
20:55:24.179 INFO - Executing: org.openqa.selenium.remote.server.handler.GetSess
ionCapabilities@23a647ea at URL: /session/1337704520891)
20:55:24.179 INFO - Done: /session/1337704520891
20:55:24.257 INFO - Executing: [get: http://www.google.com] at URL: /session/133
7704520891/url)
20:55:25.209 INFO - Done: /session/1337704520891/url
20:55:25.255 INFO - Executing: [find element: By.selector: .gbqfif] at URL: /ses
sion/1337704520891/element)
20:55:25.380 INFO - Done: /session/1337704520891/element
20:55:25.427 INFO - Executing: [is displayed: 0 org.openqa.selenium.support.even
ts.EventFiringWebDriver$EventFiringWebElement@479e4bad] at URL: /session/1337704
520891/element/0/displayed)
20:55:25.443 INFO - Done: /session/1337704520891/element/0/displayed
20:55:25.458 INFO - Executing: [close window] at URL: /session/1337704520891/win
dow)
20:55:25.458 INFO - Done: /session/1337704520891/window
```

Kuva 7: Palvelinnäkymä onnistuneen testin jälkeen

```
15  public class Registration {
16      static final String FIRST_NAME = CommonFunctions.generateString();
17      static final String LAST_NAME = CommonFunctions.generateString();
18      static final String PHONE_NUMBER = CommonFunctions.generatePhoneNumber();
19
20      @Test
21      public void shouldGoThroughRegistrationTests() throws Exception {
22          try {
23              shouldShowTermsWhenClicked();
24              shouldNotShowTermsWhenClickedSecondTime();
25              shouldNotGetThroughRegistrationWithFaultyData();
26              shouldShowJobDetailsWhenRelevant();
27              shouldGoThroughRegistrationWithCorrectData();
28              refreshPageIfUsingIE();
29          } catch (Exception e) {
30              CommonFunctions.takeScreenshot();
31              throw e;
32          }
33      }
```

Kuva 8: Esimerkki testin rakenteesta



Kuva 9. Järjestelmän rakenteen esimerkki

Jokainen JUnit-testi alkaa rekisteröitymisellä ja loppuu luodun käyttäjän poistamiseen tietokannasta. Näin saadaan aikaan toisistaan riippumattomia testejä.

Eri työkalujen hallintaan ja päivitykseen käytetään Mavenia. Maven on työkalu, joka huolehtii projekteihin liittyvien moduulien lataamisesta niille määritetyn POM-asetustiedoston perusteella. POM-asetustiedostossa määritellään ulkoiset kirjastot joita halutaan käyttää sekä niiden versiot. Kun Maven ajetaan se hakee määritellyt kirjastot. Kuvasta 10 nähdään kuinka POM-tiedostoon on määritelty JUnit ja Selenium kirjastot.

```
17    <dependencies>
18      <dependency>
19        <groupId>junit</groupId>
20        <artifactId>junit</artifactId>
21        <version>4.10</version>
22        <type>jar</type>
23      </dependency>
24      <dependency>
25          <groupId>org.seleniumhq.selenium</groupId>
26          <artifactId>selenium-java</artifactId>
27          <version>2.21.0</version>
28      </dependency>
29    </dependencies>
```

Kuva 10: POM-tiedoston määritelmät

Yksi Selenium-testien haasteista on sivujen latautumisajan vaihtelu. Joillain järjestelmillä testi pyörii ongelmitta, kun taas toisilla testit hajoavat. Tähän ei ole olemassa yksinkertaista ratkaisua Selenium-ajureiden puolelta, vaan ratkaisu täytyy ainakin osittain itse räätälöidä.

Ongelman korjaamiseen luotu luokka *RetryingExpectedConditions* ajaa toistuvasti metodia ja pyrkii suorittamaan annetut komennot (ks. kuva 11). Tässä tapauksessa *setText* funktio etsii sivulta määritellyn elementin ja lähettää siihen annetun tekstin. Lause toistetaan *RetryingExpectedCondition*:in avulla kunnes joko saavutetaan määritelty TimeOut aika, tässä tapauksessa 10 sekuntia, tai kunnes lause saadaan suoritettua onnistuneesti.

```
public WebElement setText(final By by, final String text) {
    return longWaiter().until(new RetryingExpectedCondition<WebElement>() {
        @Override
        public WebElement doApply() {
            final WebElement element = webDriver().findElement(by);
            element.clear();
            element.sendKeys(text);
            return element;
        }
    });
}

public WebElement setText(final WebElement webElement, final String text) {
    return longWaiter().until(new RetryingExpectedCondition<WebElement>() {
        @Override
        protected WebElement doApply() {
            webElement.clear();
            webElement.sendKeys(text);
            return webElement;
        }
    });
}
```

Kuva 11: Funktio jossa nähdään ratkaisu latautumisaikoihin

5.1.2 Testitulosten käsittely

Testitulosten vertaamiseen käytetään JUnitin tarjoamia työkaluja. Mikäli jokin testeistä epäonnistuu, antaa Selenium-testi virheen. JUnit poimii tämän virheen (ks. kuva 5) ja ilmoittaa asiasta Hudsonille, joka näyttää testitulokset graafisessa käyttöliittymässä (ks. kuva 12). Kuvassa 5 näkyy myös virheviesti joka tulostetaan jos *assertTrue* lause epäonnistuu.

Hudson

New Job
Manage Hudson
People
Build History
New View

Build Queue
No builds in the queue.

Build Executor Status
Status 0/2
Idle

S	W	Job	Last Success	Last Failure	Last Duration	Console
		Example project	17 sec (#5)	N/A	57 ms	
		Selenium tests	4 min 43 sec (#3)	35 sec (#4)	53 ms	

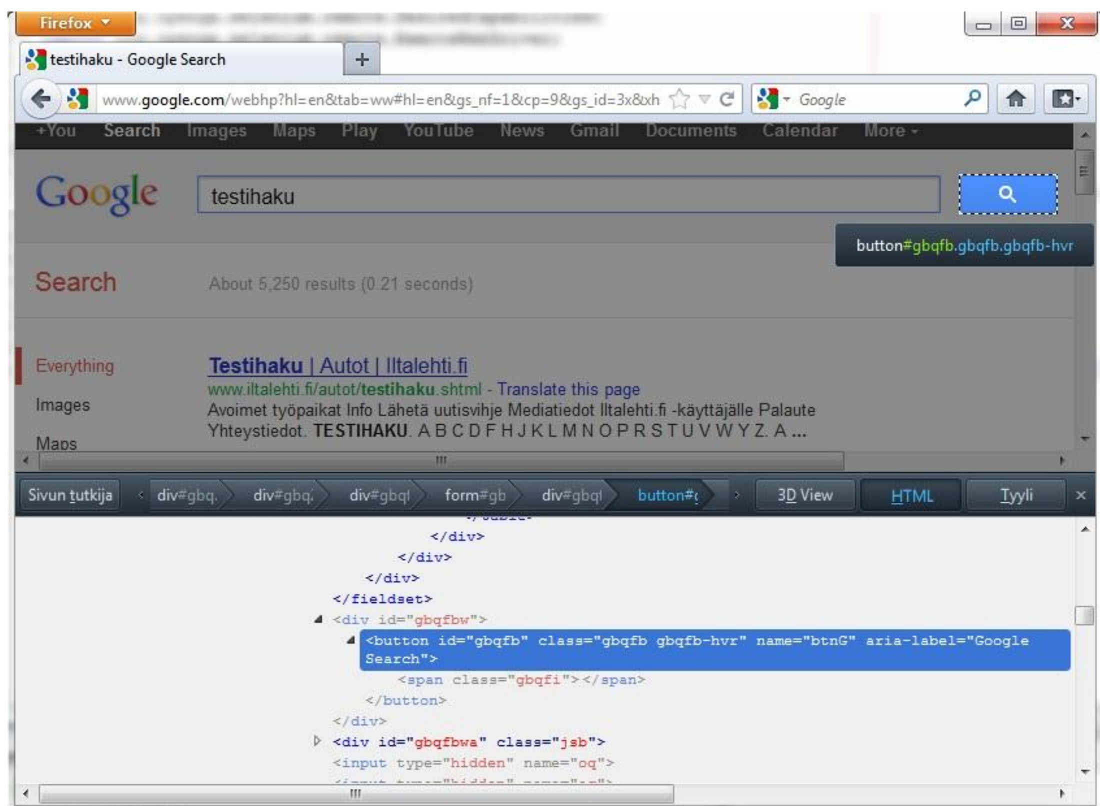
Icon: S M L

Legend · for all · for failures · for just latest builds

Kuva 12: Virheellinen testi Hudsonissa

5.2 Testien toteutus

Testejä ajettaessa tulee tietää kunkin sivuston elementin luokka, nimi tai ID. Testejä ajettaessa Selenium-kutsuille annetaan käsiteltävän elementin tiedot. Näiden tietojen perusteella Selenium etsii elementin ja suorittaa määritellyt toiminnot. Helpoin tapa saada luokka, nimi tai ID selville on käynnistää mikä tahansa selain kehitys-tilassa. Käytetään esimerkkinä Mozilla Firefox selainta. Kuvassa 13 näkyy kuinka sivustolta on valittuna ”Sivun etsijä”:n avulla etsi-nappi. Tässä tilassa on helppo nähdä napille kuuluvat ominaisuudet. Sen ID on *gbqfb*, nimi on *btnG* ja luokka on *gbqfb*. Näiden tietojen avulla elementtiä pystytään kutsumaan Seleniumissa.



Kuva 13: Firefox "Sivun tutkija"-tilassa

Testit toteutetaan JUnit Suiteina (ks. kuva 14). Suiteen määritellään ajettavat testiluokat, ja Suite ajaa ne annetussa järjestyksessä. Täten varmistetaan, että esimerkiksi rekisteröintitesti suoritetaan ennen henkilötietojen muokkaustestiä.

Yksittäiset testiluokat pyritään pitämään mahdollisimman riippumattomina muista testeistä. Testeissä tarkkaillaan vain kyseisen testin tekemien muutosten ja asioiden vaikutuksia, eikä kiinnitetä minkäänlaista huomiota muiden testien tuloksiin. Esimerkiksi henkilötietojen muokkaamista testattaessa ei tarkisteta ovatko tiedot oikein rekisteröinnin jäljiltä. Siinä käsitellään vain henkilötietojen muokkaamiseen liittyviä tuloksia, eli tarkistetaan muuttuvatko tiedot oikein kun muutokset tallennetaan.

```
1   package TestPackage;
2
3   import MainClass.DriverProvider;
4   import MainClass.SeleniumContext;
5   import org.junit.AfterClass;
6   import org.junit.BeforeClass;
7   import org.junit.runner.RunWith;
8   import org.junit.runners.Suite;
9   import org.openqa.selenium.remote.DesiredCapabilities;
10
11  @RunWith(Suite.class)
12  @Suite.SuiteClasses({Google.class, Search.class})
13  public class TestRunner {
14      @BeforeClass
15      public static void init() throws Exception {
16          SeleniumContext.setDriver(DriverProvider
17                  .getDriver(DesiredCapabilities.firefox()));
18      }
19
20      @AfterClass
21      public static void stop() throws Exception {
22          SeleniumContext.getDriver().quit();
23      }
24  }
```

Kuva 14: JUnit Suite esimerkki

Mikäli järjestelmää pystytään testaamaan rekisteröitymättä uudella käyttäjällä, pystytään välttämään riippuvuus rekisteröintitestistä. Tässä tapauksessa kuitenkin syntyy riippuvuus sisäänkirjautumistestiin. Loppukäyttäjätestien yhteydessä on kuitenkin lähes mahdotonta tehdä testeistä täysin toisista riippumattomia ilman turhaa toistoa. Testit ovat aina riippuvaisia joko rekisteröitymisestä tai sisäänkirjautumisesta. Jokaisen testisession lopuksi testihenkilö poistetaan tietokannasta. Tämä saadaan aikaiseksi ajamalla SQL-lause Javalla.

Turhan toiston välttämiseksi ja testaamisen nopeuttamiseksi luotiin useita eri luokkia jotka toteuttavat kaikille testeille yhteisiä toimintoja. Ensimmäisenä tulee *DriverProvider*-luokka joka luo ja tarjoaa muille luokille oikein määritellyn *RemoteWebDriverin* jota käytetään Selenium-palvelimelle yhdistettäessä (ks. kuva 15).

```
1   package MainClass;
2
3   import java.net.MalformedURLException;
4   import java.net.URL;
5   import org.openqa.selenium.remote.DesiredCapabilities;
6   import org.openqa.selenium.remote.RemoteWebDriver;
7
8
9   public class DriverProvider {
10      public static RemoteWebDriver getDriver(DesiredCapabilities capabilities)
11              throws MalformedURLException {
12          return new RemoteWebDriver
13                  (new URL("http://127.0.0.1:4444/wd/hub"), capabilities);
14      }
15  }
```

Kuva 15: DriverProvider-luokka

Toisena tällaisena luokkana toimii *SeleniumContext*, joka ottaa vastaan ja säilöö staattisessa muuttujassa annetun *RemoteWebDriverin* (ks. kuva 16).

```
1   package MainClass;
2
3   import org.openqa.selenium.remote.RemoteWebDriver;
4
5
6   public class SeleniumContext {
7
8       private static RemoteWebDriver driver;
9
10      private SeleniumContext() {
11
12      }
13
14      public static void setDriver(RemoteWebDriver driver) {
15          SeleniumContext.driver = driver;
16      }
17
18      public static RemoteWebDriver getDriver() {
19          return driver;
20      }
21  }
```

Kuva 16: SeleniumContext-luokka

Kolmas ja suurin yleinen luokka on *BrowserFunctions* (ks. kuva 17 & 18). Se sisältää yli 200 riviä erilaisia funktioita joilla käytännössä korvataan Seleniumin sisäisten funktioiden suora käyttö. Selenium funktioita kutsutaan *BrowserFunctions*:in sisällä mutta niihin sisältyy paljon muutakin joka helpottaa testien ajamista. *BrowserFunctions* on luokka jossa ratkaistiin sivujen latautumisaikoihin liittynyt ongelma.

Kaikki Selenium-testit käyttävät *RemoteWebDriver*:iä *BrowserFunctions*:in kautta. Metodi *webDriver* hakee *RemoteWebDriver*:in funktion sisäiseen käyttöön. Koska kaikki testit käyttävät luokassa määriteltyjä funktioita, käyttävät ne ajuria tätä kautta.

```
1   package MainClass;
2
3   import org.openqa.selenium.By;
4   import org.openqa.selenium.SearchContext;
5   import org.openqa.selenium.WebElement;
6   import org.openqa.selenium.remote.RemoteWebDriver;
7   import org.openqa.selenium.support.ui.Wait;
8
9   import java.net.MalformedURLException;
10  import java.net.URL;
11  import java.util.List;
12
13  import static MainClass.SeleniumContext.getDriver;
14
15  public class BrowserFunctions {
16
17      public static final BrowserFunctions BROWSER_FUNCTIONS
18              = new BrowserFunctions();
19      private long suspendTime = 100;
20
21      private BrowserFunctions() {
22      }
23
24      public RemoteWebDriver webDriver() {
25          return SeleniumContext.getDriver();
26      }
27
28      public void suspend() {
29          try {
30              Thread.sleep(suspendTime);
31          } catch (InterruptedException e) {
32              throw new UnsupportedOperationException();
33          }
34      }
35
36      public void setSuspendTime(long suspendTime) {
37          this.suspendTime = suspendTime;
38      }
39
40      public void loadSite(final String url) {
41          webDriver().get(url);
42          fullScreen();
43      }
```

Kuva 17: BrowserFunctions esimerkki 1

```
45      public void fullScreen() {
46          runScript("if (window.screen){window.moveTo(0, 0);"
47                  + "window.resizeTo(window.screen.availWidth,"
48                  + "window.screen.availHeight);};");
49      }
50
51      public void runScript(final String script) {
52          webDriver().executeScript(script);
53      }
54
55      public WebElement untilFound(final By by) {
56          return longWaiter()
57                  .until(new RetryingExpectedCondition<WebElement>() {
                @Override
59              public WebElement doApply() {
60                  return webDriver().findElement(by);
61              }
62          });
63      }
64
65      public Boolean untilClassNotFound(final By by, final String string) {
66          return longWaiter()
67                  .until(new RetryingExpectedCondition<Boolean>() {
                @Override
69              public Boolean doApply() {
70                  return !getDriver().findElement(by)
71                          .getAttribute("class").contains(string);
72              }
73          });
74      }
75
76      public List<WebElement> untilFound(final By by, final int expectedAmount) {
77          return longWaiter().until(new RetryingExpectedCondition<List<WebElement>>() {
                @Override
79              public List<WebElement> doApply() {
80                  final List<WebElement> elementList = webDriver().findElements(by);
81                  return elementList.size() == expectedAmount ? elementList : null;
82              }
83          });
84      }
```

Kuva 18: BrowserFunctions esimerkki 2

Useamman kehittäjän tiimissä saattaa tulla päällekkäisyyksiä testejä ajettaessa. Jos kaksi henkilöä ajaa samaan aikaan testejä, tai toinen henkilö aloittaa testien ajamisen samalla kun toinen henkilö ajaa niitä, syntyy helposti tietokannassa päällekkäisyyksiä. Finder-järjestelmä ei siedä kahta henkilöä joilla on samat nimet ja syntymäajat, tai sama sähköpostiosoite. Tällaisten konfliktien välttämiseksi luotiin funktion joka luo satunnaisia merkkijonoja etu- ja sukunimille sekä e-mailille. Sähköpostien loppuosa kuitenkin pysyy aina samana (@testiselenium.fi). Näin varmistetaan mahdollisuus poistaa helposti tietokannasta kaikki ylijääneet testihenkilöt.

Testihenkilöt voitaisiin poistaa BeforeClass-annotaatiossa, eli ennen yhdenkään testin ajamista. Tämä ei kuitenkaan ole mahdollista satunnaisten merkkijonojen käyttöönoton jälkeen. Merkkijonot luodaan kun niitä kutsutaan ensimmäistä kertaa. Käyttäjätiedot sisältävät merkkijonot ovat tyyppiä static final, eli ensimmäisen kutsun jälkeen niiden arvo ei voi enää muuttua. Koska merkkijonot arvotaan jokaisen ajon alussa, on mahdotonta poistaa tietoja tietokannasta BeforeClassissa, sillä testillä ei ole tietoa mikä nimi arvottiin testiä viimeksi ajettaessa. Tietojen poistaminen olisi mahdollista käyttämällä hyväksi sähköpostin post-fixiä, mutta tällöin päällekkäisten testien ajo olisi mahdotonta. Lause nimittäin poistaisi sen hetkisen testin tiedot, jolloin testi kaatuisi.

Tavallisessa tapauksessa tietokantaan ei jää testihenkilön tietoja edes silloin kun testi epäonnistuu. Tämä varmistetaan ajamalla tietokantakutsu AfterClass-annotaatiolla. AfterClass suoritetaan aina testien jälkeen riippumatta siitä onnistuvatko vai epäonnistuvatko testit. Ainoa tapaus jossa testihenkilön tiedot voivat jäädä tietokantaan on kun kehittäjä itse katkaisee testin. Tällöin suoritetaan kill-komento, jonka seurauksena AfterClassia ei suoriteta.

E-mailin post-fixin pysyessä aina samana, @testiselenium.fi, pystytään ylimääräiset tiedot poistamaan tietokannasta helposti SQL-lauseella:

```
1  SELECT * FROM tietokanta WHERE email LIKE '%testiselenium%';
```

Kuva 18. Tietokannan käsittely

Jotta selenium voi suorittaa toimintonsa tulee sille ensin kertoa mitä web elementtiä se käsittelee, sekä kriteerit joilla elementti etsitään. Varmin tapa löytää elementit on suorittaa niiden etsiminen CSS Selectoreilla. Selectorille voi määritellä mm. elementin id:n, luokan tai tyypin. Esimerkkeinä:

```
1  SeleniumContext.getDriver().findElementByCssSelector("#tämäonid");
2  SeleniumContext.getDriver().findElementByCssSelector(".tämäonluokka.jatämätoinenluokka");
3  SeleniumContext.getDriver().findElementById("tämäonid");
```

Kuva 19. CSS Selectorien käyttöä

WebElementtien löydyttyä voidaan niille antaa komentoja kutsumalla funktioita:

```
1  SeleniumContext.getDriver().findElementById("tekstikenttä").sendKeys("auto");
2  SeleniumContext.getDriver().findElementById("nappi").click();
```

Kuva 20. Funktiokutsuja

Seleniumille saatiin testausalustaksi Windows XP virtuaalikone.

Virtuaalikoneen käyttöönotossa ilmeni muutama hidaste. Virtuaalikone sijaitsee palvelimella, jolla on monta virtuaalikonetta käynnissä. Virtuaalikoneiden IP-osoitteet resetoituvat joka kerta kun palvelin käynnistetään uudelleen, mm. viikoittaisten huoltotöiden yhteydessä.

Tästä muodostui ongelma sillä Selenium-testien olisi kuulunut yhdistää eri IP:lle kuin sillä hetkellä määritelty. Uudelleenkäynnistys myös palauttaa Windows XP:n tilaan missä siitä on tehty boottikopio. Tämän johdosta kaikki asetukset sekä asennukset pyyhkiytyvät pois. Virtuaalikoneella on kuitenkin yksi verkkoasema, joka pysyy pyyhkiytymisten välillä. Tälle tallennettiin Selenium-palvelimen tiedostot. Käyttöönoton nopeuttamiseksi asemalle sijoitettiin myös Firefoxin ja Chromen asennustiedostot, joista tehtiin pikakuvake työpöydälle.

Aluksi IP-osoitteiden päivittäminen hoidettiin manuaalisesti. Myöhemmin operaatio automatisoitiin rekisteröimällä DNS-osoite DynDNS-palvelusta. Selenium-koneelle sijoitettiin Dyn Updater, joka IP-osoitteen muutoksen havaitessaan päivittää sen DynDNS-palvelimelle internettiin. Nykyään testit viittaavat DNS-osoitteeseen (esim. baronaselenium-dyndns.com). Pitääkseen IP-osoitteen tuoreena tulee testaajan huolehtia Dyn Updaterin käynnistämisestä

aina kun virtuaalikone pyyhitään. DynDNS-palvelua ei ole varsinaisesti tarkoitettu tällaiseen käyttöön, mutta pienellä säätämisellä se saatiin näyttämään sisäverkon IP:itä ulkoverkon IP:n sijasta. DynDNS-palveluun tallennettu osoite on sisäverkon osoite, eikä sitä pystytä käyttämään hyväksi mistään muualta kuin Baronan sisäverkosta, jossa sekä testikone että Hudson sijaitsevat.

6 ARVIOINTI

Opinnäytetyön toiminnallista osuutta työstettäessä kului eniten aikaa testijärjestelmän pohjan luomiseen. Seleniumin mukana tulevat funktiot eivät riittäneet monimutkaisempien testien toteuttamiseen ilman turhaa toistoa. Näiden testien toteuttamista varten jouduttiin luomaan uusia funktioita käyttäen Selenium mahdollistamia toimintoja kekseliäästi. Kunnollisen pohjan luomisen jälkeen testien kirjoittaminen helpottui huomattavasti.

Testien ajaminen ulkoisella testikoneella ei sujunut saumattomasti. Windows-pohjainen testikone resetoitui viikoittain oletusasetuksiinsa. Tämän seurauksena koneelle oli asennettava tietyt selaimet uusiksi ja Internet Explorerin asetuksia oli säädettävä testauksen mahdollistamiseksi. Ilman asetuksien muuttamista Internet Explorer näytti virheen koskien epäluotettua sertifikaattia. Tämä ilmoitus esti testien ajamisen.

Työn toteutusosa saatiin haluttuun tilaan. Järjestelmälle kirjoitettiin esimerkkitestejä ja valmis pohja mahdollistaa uusien testien lisäämisen helposti. Kun otetaan huomioon, että nämä automaattisesti ajettavat testit suoritettiin ennen manuaalisesti tietyin väliajoin, syntyy järjestelmän käyttöönotosta sitä enemmän säästöjä mitä kauemmin järjestelmä on käytössä. Testit voidaan nyt ajoittaa automaattisesti ajettavaksi haluttuina ajankohtina. Oletusasetuksena testit ajetaan aina kun järjestelmässä havaitaan muutoksia.

Olen opinnäytetyön lopputulokseen tyytyväinen. Kohdeyrityksen kannalta kehitys sujui kivutta sillä kaikki kehityksessä käytetyt resurssit olivat joko ilmaisia tai jo ennestään yrityksellä käytössä.

7 LÄHTEET

Clark, M. 2006. JUnit FAQ. [WWW-dokumentti]. [Viitattu 10.04.2012]. Saatavissa:
http://junit.sourceforge.net/doc/faq/faq.htm

Dumas, J. Redish, J. 1999. A Practical Guide to Usability Testing. Bristol: Intellect Ltd.

Dyer, D. 2008. Why are you still not using Hudson? [WWW-dokumentti]. [Viitattu 10.04.2012]. Saatavissa:
http://blog.uncommons.org/2008/05/09/why-are-you-still-not-using-hudson/

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

Finder Yritystieto, Barona Group Oy. [WWW-dokumentti]. Fonecta. [Viitattu 11.04.2012]. Saatavissa:
http://www.finder.fi/Henkil%C3%B6st%C3%B6n%20vuokrausta%20ja%20v%C3%A4lityst%C3%A4/Barona%20Group%20Oy/HELSINKI/toiminta/292773

Hetzel, B. 1993. The Complete Guide to Software Testing, Second Edition. New York: John Wiley & Sons, Inc.

Kaner, C. Bach, J. & Pettichord, B. 2002. Lessons Learned in Software Testing, A Context-Driven Approach. New York: John Wiley & Sons, Inc.

Kaner, C. Falk, J. & Nguyen, H. C. 1993. Testing Computer Software, Second Edition. New York: John Wiley & Sons, Inc.

Myers, G. 1979. The Art of Software Testing. New York: John Wiley & Sons, Inc.

Portfolioyhtiö Barona. [WWW-dokumentti]. Sponsor Capital Oy. [Viitattu 11.04.2012]. Saatavissa:
http://www.sponsor.fi/fi/portfolioyhtioet/59-barona

Rubin, J. Dana, C. 2008. Handbook of Usability Testing, Second Edition. New York: John Wiley & Sons, Inc.

Selenium Documentation. 2012. [WWW-dokumentti]. [Viitattu 10.04.2012]. Saatavissa:
http://seleniumhq.org/docs/

What is Maven? 2012. [WWW-dokumentti]. [Viitattu 27.05.2012]. Saatavissa:
http://maven.apache.org/what-is-maven.html