Aleksi Ahvamaa

KÄYTTÖLIITTYMÄTASON AUTOMAATTINEN TESTAUS
CASE: ENERSIZE OY

Tietojenkäsittelyn koulutusohjelma
Ohjelmistosuunnittelun suuntautumisvaihtoehto
2011

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

KÄYTTÖLIITTYMÄTASON AUTOMAATTINEN TESTAUS
case: Enersize Oy

Ahvamaa, Aleksi
Satakunnan ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Lokakuu 2011
Ohjaaja: Nieminen, Hans
Sivumäärä: 48
Liitteitä: -

Asiasanat: testaus, automaatio, käyttöliittymätestaus, testauksen automatisointi, testaustyökalut, ketterät kehitysmenelmät

Tämän opinnäytetyön toimeksiantaja oli Enersize Oy. Enersize Oy:n toimiala on cleantech ja yritys toimittaa energiatehokkuus ratkaisuja teollisuusyrityksille maailmanlaajuisesti.

Tämän opinnäytetyön tarkoituksena oli löytää toimiva toimintamalli yrityksessä kehitettävän sovelluksen käyttöliittymän toiminnallisuuden todentaminen ja parantaminen testaamalla sitä automaattisesti. Toimintamallin toteutuksen olisi oltava yhteensopiva jo tehtävien automatisoitujen yksikkötestien kanssa sekä muiden sovelluskehitystiimissä olevien työntekijöiden helposti omaksuttavissa. Toimintamallin lisääminen kehitysprosessiin ei saisi olla liian työlästä eikä sen opetteluun kulua liikaa resursseja. Työn aikana haluttiin tutkia onko kannattavaa testata koko sovelluksen toiminnallisuus ja jos ei kannata, niin mitkä toiminallisuudet ovat niitä, jotka tulee testata ja mistä syystä.

Opinnäytetyön teoriaosuudessa käydään läpi testauksen perusperiaatteita ja käytäntöjä, jotka on hyvä tietää. Lisäksi tarkastellaan testauksen automatisointiin liittyviä asioita sekä kerrotaan työkaluista ja ketteristä kehitysmenetelmistä, jotka ovat valtaamassa alaa vanhemmilta menetelmiltä ja siten myös liittyvät projekteissa suoritettavaan testaukseen.

Työn tuloksena saatiin kehitettyä toimintamalli siihen, miten kehitettävän sovelluksen käyttöliittymän testausta tullaan viemään läpi. Toimintamallin käyttäminen tulee parantamaan sovelluksen laatua ja luotettavuutta. Työn aikana huomattiin että mikäli on valmiina olevaa toiminnallisuutta tehtynä, niin kannattaa aloittaa testaus tärkeimmistä toiminnallisuuksista liiketoiminnan kannalta ja uusista kehitteillä olevista ominaisuuksista eikä yrittää tehdä testejä kaikelle valmiina olevalle ensiksi. Työn kehittäminen jatkuu vielä tässä opinnäytetyössä aikaan saadun tuotoksen monipuolistamisella kohti kokonaisvaltaisempaa käyttöliittymän testauksen automatisointiratkaisua.

AUTOMATED GRAPHICAL USER INTERFACE TESTING
case: Enersize Oy

Ahvamaa, Aleksi
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Business Information Systems
October 2011
Supervisor: Nieminen, Hans
Number of pages: 48
Appendices: -

Keywords: testing, automation, graphical user interface testing, automated testing, testing tools, agile development

Employer of this thesis was Enersize Ltd. Enersize Ltd. works in cleantech industry and delivers energy efficiency solutions to industrial companies worldwide.

Purpose of this thesis was to find a working pattern to authenticate functionality and improve quality by automaticly testing application that is developed in the company. Pattern would have to be compatible with already automated unit tests that are done and executed during development process. Solution should also be easy to absorbed by other members of the software development team, so it would not be too troublesome to add into development process and wouldn't take too much resources in the process. During the project it was also researched that if it would be worthwhile to test the applications whole functionality and if not then so what would those funcionalities to test be and what are reasons for that.

In theoretical part of this thesis basic principles and methods of testing that would be preferred to know is introduced to reader. I also go over some matters that includes automating testing, tools for test automation and agile development methods that are conquering space from traditional methods and that way are related to testing that is done in software development projects.

As a result of this thesis were a pattern for application under development graphical user interface testing executing which is improving quality and reliability of the application. During the project it was noticed that if there is functionality done before graphical user interface testing is started, then it would be preferred to start testing from those functionalities that are important for business and features coming in future rather than trying to test all the done functionalities before making tests for new features. The development work will continue after this thesis yield towards more overal user interface test automation solution.

SISÄLLYS

1 JOHDANTO

Tämä opinnäytetyö käsittelee Enersizelle tekemääni käyttöliittymän testaus konseptia tai toimintamallia. Työharjoitteluni loppuvaiheessa keskustelimme yhdessä yrityksessä toimineen ohjaajani kanssa mahdollisesta vaihtoehdosta jatkaa töitä yrityksessä opinnäytetyön parissa. Ohjaajallani oli mielessään jo aihe opinnäytetyölle, minkä voisin toteuttaa. Minulta kysyttiin, että haluaisinko alkaa toteuttamaan yritykselle käyttöliittymän testauksen konseptia, johon kuuluisi suunnittelua, itse testien tekemistä sekä mahdollisesti pienimuotoista koulusta.

Tarkoitus oli toteuttaa toimintamalli, jolla myös muut tiimin jäsenet pystyisivät luomaan näitä testejä. Otin työn mielelläni vastaan ja mielestäni aihekin oli mielenkiintoinen ja syventäisi osaamistani, sillä olin jo aikaisemmin tehnyt seminaarityöni aiheesta testaus. Seminaarityöni käsitteli testausta yleiseltä kannalta, mutta ei ottanut kantaa testauksen automatisointiin taikka sen tuomiin haasteisiin ja hyötyihin.

Opinnäytetyötä lähdettiin viemään läpi projektina, joka alkaisi kesän 2011 aikana ja konsepti tai toimintamalli tulisi ottaa käyttöön vuoden loppuun mennessä. Tulin toteuttamaan työni käyttäen avoimen lähdekoodin ohjelmistoja. Lähtökohtana oli suunnitella ja toteuttaa käyttöliittymän testaukseen soveltuva toimintamalli, jota tultaisiin hyödyntämään yrityksen sovelluskehityksessä parantamaan kehitettävän web-sovelluksen luotettavuutta ja toimintavarmuutta.

Valmiin järjestelmän tulisi olla nopeasti opittava, helppokäyttöinen, eikä se saisi olla liian raskas käyttää. Henkilökohtaisena tavoitteenani oli oppia lisää testauksen automatisoinnista ja syventää testausosaamistani edelleen jatkuvan integraation mukaan tuomisella.

2 TYÖN LÄHTÖKOHDAT

2.1 Toimeksiantaja

Opinnäytetyöni toimeksiantajana toimi yritys nimeltä Enersize. Enersize Oy on cleantech -alan yritys. Yrityksen toiminta on keskittynyt teollisuuden energiatehokkuusratkaisujen toimittamiseen maailmanlaajuisesti sekä yleiseen energiatehokkuuteen liittyvän tietouden edistämiseen. Toimipisteet yritykseltä löytyy Ulvilasta ja Helsingistä, joista pääpaikkana toimii Ulvila.

2.2 Tehtävän kuvaus

Opinnäytetyön tarkoitus oli suunnitella ja kehittää konsepti tai toimintamalli, jolla pystytään testaamaan kehitteillä olevan web-sovelluksen käyttöliittymää ja sen toimintaa automatisoiduin testein. Työn aloitushetkellä tehtiin sovelluksen käyttöliittymän testaus aina manuaalisesti, joka tarkoitti uusien ominaisuuksien läpi käymistä eri selaimilla käsin ja siihen vierähtäisi aina oma aikansa. Käytännössä tämä tarkoitti sitä, että testaaminen jätettiin pisteeseen ennen julkaisua. Kun testaus tehtiin juuri ennen julkaisua, saattoi osa lisätyistä tai muutetuista ominaisuuksista jäädä testaamatta. Sovelluksen toimintaa testattiin jo automaattisilla yksikkötesteillä, mutta se ei ulottunut vielä käyttöliittymätason toiminnallisuuteen.

Huomattiin myös se, että testauksen jääminen tehtäväksi ennen julkaisua aiheutti muutamien pienien virheiden pääsyn seulan läpi, jotka sitten jouduttiin korjaamaan nopeasti julkaisun jälkeen niiden tullessa ilmi. Nämä pienet virheet toiminnallisuudessa eivät olisi todennäköisesti päässeet läpi ohjelman mukana, mikäli käyttöliittymän toimintaa olisi testattu automaattisilla testeillä.

Opinnäytetyö päätettiin toteuttaa projektina jotta sitä voitaisiin jatkaa myös sen jälkeekin kun opinnäytetyön osuus loppuu, mikäli tarve sitä vaatii. Projekti alkaisi kesän aikana ja jatkuisi opinnäytetyön osalta saman vuoden loppupuolelle. Tavoitteena oli se, että projektin päättyessä olisi toimintamalli tai konsepti otettu käyttöön ja sitä sovellettaisiin kehitystyössä. Tuloksena haluttiin pystyä parantamaan kehitettävän web-sovelluksen laatua, käytettävyyttä ja lisätä toimintavarmuutta.

Haluttiin myös tutkia sitä, miten testausta kannattaa lähteä suorittaamaan sekä mitä osia ohjelmasta halutaan testata. Onko jotain mitä ei kannata testata ja jos ei, niin miksi. Työlle asetettiin myös joitain toivomuksia, jotka tulisivat toimimaan ohjaavina reunaehtoina työn edetessä. Näitä toivomuksia olivat sellaiset asiat kuin: 1. Uuden toimintamallin tulisi olla helposti tai nopeasti opittavissa, jotta ei kulu turhaa aikaa uuden opetteluun, 2. Toimintamallin tulisi olla yhteensopiva nykyisen kehitysmallin kanssa, jolloin se voidaan integroida suoraan osaksi kehitysprosessia.

3 YLEISTÄ TESTAUKSESTA

Mitä testauksella halutaan saada aikaan? Yksinkertaistettuna testaus on virheiden etsintää ohjelmasta taikka ohjelmistosta ja niiden korjaamista. Tällä päästään siihen, että kehitettävän ohjelmiston laatu nousee sekä toimintavarmuus paranee. Mikä käsitetään virheeksi? Virhe voi olla tila taikka tapahtuma ohjelmassa, joka poikkeaa sen oletetusta toiminnasta. Esimerkiksi ohjelman toiminta on jossain tilanteessa poikkeavaa ja sille tehdyn määrittelyn vastaista. Täytyy kuitenkin ottaa huomioon mahdollisuus määrittelyn virheellisyydestä. Ohjelma voi toimia niin kuin on haluttu ja aiheuttaa virhetilanteen, joka siinä tilanteessa luultavasti johtuu virheellisesti tehdystä määrittelystä.

Testauksen avulla pyritään löytämään ohjelmistossa olevat virheet mahdollisimman aikaisin. Mitä myöhemmin virhe löydetään, niin sitä kalliimmaksi sen korjaaminen tulee. Jos mahdollinen virhe löydetään määrittelyn aikana, ei sen korjaaminen maksa välttämättä mitään. Ohjelman koodauksen ja testauksen jälkeen löydetty sama virhe voi maksaa 10 – 100 rahayksikköä. Mikäli saman virheen löytää asiakas, voi korjauskustannukset nousta helposti jopa tuhansiin taikka miljooniin rahayksiköihin. Testausta suorittamalla säästetään mahdollisissa kuluissa, jotka virhetilanteet synnyttää ja sen lisäksi parannetaan ohjelman toimivuutta ja luotettavuutta.

Vaikka testausta suoritettaisiin miten paljon tahansa, ei pystytä ohjelmaa koskaan testaamaan täydellisesti ja olla täysin varmoja sen virheettömyydestä. Testauksen tavoitteena on hyvä olla mahdollisimman monen virheen löytäminen, koska koskaan

ei voida olla aivan varmoja ohjelman täydellisestä virheettömyydestä. Mikäli lähdetään metsästämään aivan kaikkia mahdollisia virheitä, niin testausta voitaisiin jatkaa niin pitkään kuin resursseja riittää.

3.1 Testausprosessi

Kuten muukin ohjelmistokehitys, niin myös testaus etenee askel kerrallaan. Perinteisessä ohjelmistokehityksessä testaus tapahtuu usealla tasolla (kts. kuva 1). Kuvan V-mallissa on testaus jaettu moduulitestaukseen (module testing), integrointitestaukseen (integration testing) ja järjestelmätestaukseen (system testing).

Usein kuulee sanottavan, että ”Hyvin suunniteltu on jo puoliksi tehty”, joka mielestäni pitää osittain paikkansa myös ohjelmistoprojekteissa. On hyvä suunnitella jollain tasolla se, miten ohjelmaa aiotaan suunnitella ja mitä työkaluja siihen tullaan käyttämään. Suunnittelua koskevien päätöksien ei tulisi kuitenkaan olla kiveen hakattuja, koska mikään ei ole niin varmaa ohjelmistoprojekteissa kuin se, että muutoksia tulee aivan varmasti vastaan.

Muutoksiin tulisi pystyä reagoimaan mahdollisimman nopealla sykkeellä, johon ei aikaisemmilla kehitysmalleilla pystytty. Vesiputousmalli on hyvä esimerkki, koska siinä kaikki pyritään määräämään ennalta. Tällaiset toimintamallit ovat auttamatta liian jäykkiä ja joustamattomia nykyisen ohjelmistokehityksen tarpeisiin koska ne eivät mahdollista muutosten tekemistä nopeasti, mikäli mahdollistavat niitä ollenkaan.

Esimerkkinä voisin käyttää sellaista tapausta, että ohjelmistokehittäjä Olli löytää meneillään olevan projektin aikana uuden työkalun tai tavan tehdä sovelluksen testaus. Tällä uudella menetelmällä voitaisiin projektin testaus suorittaa nopeammin ja vähemmillä resursseilla kuin tällä hetkellä on käytössä menetelmällä. Olli esittää asian eteenpäin esimiehellensä Esalle, joka tyrmää ehdotuksen vedoten sellaisiin seikkoihin kuten: ”Näin meillä on aina tehty”, ”Nämä asiat on jo lyöty lukkoon määrittelyssä” tai ”Tämä projekti viedään läpi näillä työkaluilla ja seuraavaan projektiin voidaan harkita otettavaksi käyttöön tämä uusi menetelmä.

Esa haluaa saada määrittelyt dokumentoituna uudesta työkalusta tai -menetelmästä ennen mahdollista päätöstä. Tästä seuraa se, että Olli palaa työpisteelleen ja jatkaa projektin testausosuuden läpi viemistä vanhoilla työkaluilla ja luultavasti haluttuja dokumentteja testausmenetelmän muuttamiseksi ei koskaan tehdä. Tästä seuraa se, että yrityksen testaustoiminnan mahdollinen tehostuminen tai kehittyminen ei koskaan toteudu.

Toinen ääripää tästä samasta esimerkistä on se, että Ollin esimies Esa kuuntelee ehdotuksen ja lähtee siihen mukaan. Ollille annetaan lupa muuttaa testausmenetelmä uuteen omassa projektissaan ja sen onnistuessa parantamaan testaustoimintaa, voitaisiin se ottaa käyttöön myös muissa yrityksen projekteissa. Tässä tapauksessa Olli palaa tekemään testausosuuden uusilla työkaluilla ja huomataan miten paljon helpommin pystytään tekemään samat asiat tekemään mielekkäämmin ja siihen tarvittavien resurssien määrä pienenee.

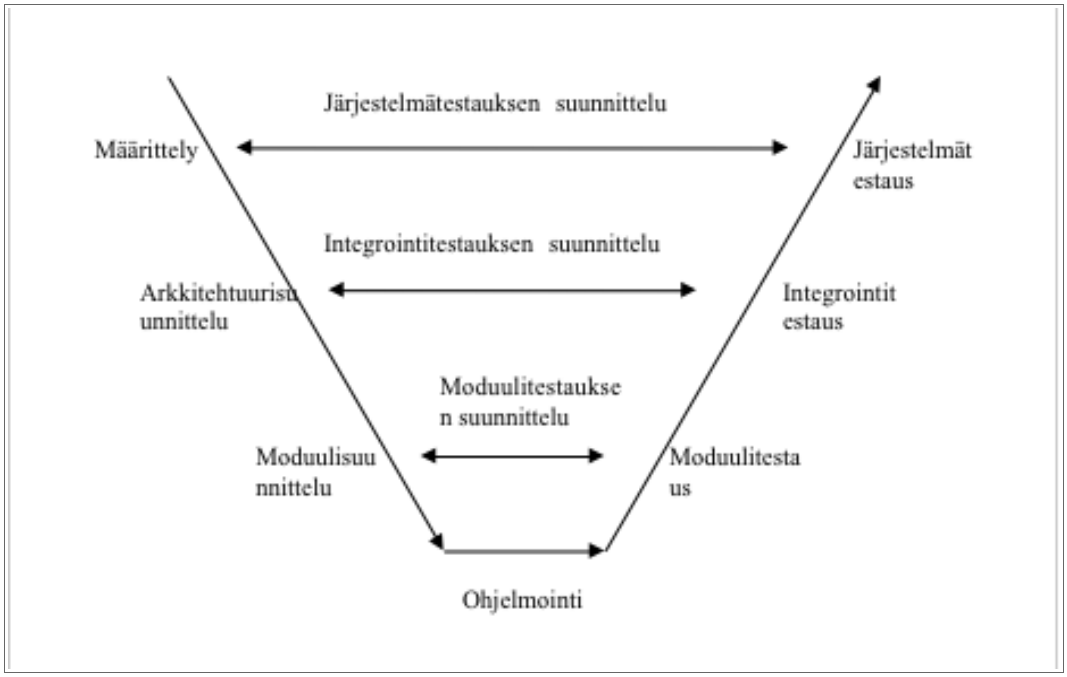
Tämän seurauksena tätä uutta työkalua siirretään käytettäväksi pian myös muissa projekteissa. Näin yritys pystyy käyttämään säästetyt resurssit johonkin muuhun. Tämän esimerkin tarkoituksena on herättää ajatuksia siitä, pystyttäisiinkö asiat hoitamaan tehokkaammin, kuitenkaan laatua laskematta. Paljon saataisiin aikaan jo pelkästään sellaisella toimenpiteellä, että pyrittäisiin vähentämään riippuvuutta määrittelyistä ja dokumentoinnista. Tällöin pystyttäisiin reagoimaan nopeammin eteen tuleviin muutoksiin.

Esimerkki ei tarkoittanut sitä, etteikö testausta tulisi suorittaa systemaattisesti ja suunnitellusti. Systemaattisesti suoritettu testaus säästää aikaa sekä antaa selkeämmän kuvan ongelma-alueista. Todellinen haaste on selvittää testausmenetelmät ja niiden kattavuus, jolloin päästään testattavan ohjelmiston sovellusteknisiin yksityiskohtiin. Ymmärtämällä sovelluksen teknisiä yksityiskohtia, pystytään käytettävä menetelmä valitsemaan paremmin.

V-mallissa suoritetaan testauksen suunnittelu aina testauksen tasoa vastaavalla suunnittelutasolla. Eli ohjelmiston määrittelyn osana tehdään suunnitelma järjestelmätestauksen osalta. Kun sitten taas integrointitestaus suunnitellaan samalla

kun tehdään arkkitehtuurisuunnitelma ja samaa logiikkaa käyttäen voidaan olettaa, että moduulisuunnittelun yhteydessä moduulitestaus suunnitellaan. Tästä päästään siihen, että testauksista saatavia tuloksia tulee verrata kyseisen suunnittelutason dokumentteihin. Jos ei halutusta tuloksesta ole olemassa minkäänlaista dokumentaatiota, niin testauksen suorittaminen vaikeutuu, koska ei ole mitään mihin saatuja tuloksia voitaisiin verrata.

Kuten V-mallista käy myös ilmi, että testauksen suunnittelu tulisi aloittaa jo aikaisemmin kuin on tehty ensimmäistäkään testattavaa ohjelmakoodiriviä. V-mallin mukaan täytyy kyseisen testaustason testaussuunnitelmat sekä testitapaukset olla valmiina, jotta testausvaihe pystyttäisiin viemään mahdollisimman nopeasti läpi. Testitapausten avulla pystytään varmistamaan se, ollaanko tuotetta rakentamassa oikein.



Kuva 1. Testauksen V-malli. (Haikala & Märijärvi 2006, 289).

Testaus suoritetaan V-mallin mukaisessa järjestyksessä alhaalta ylöspäin. Syy tähän on se, että on helpompaa löytää sekä korjata yksittäisestä moduulista löytynyt virhe kuin suuremmasta kokonaisuudesta. Kun pienemmät yksiköt on testattu, on helpompaa siirtyä suurempien kokonaisuuksien testaamiseen.

3.2 Testauksen käsitteitä

Seuraavien kappaleiden aikana tulen käsittelemään testauksen peruskäsitteistöä ja kertomaan niistä hieman tarkemmin. Tarkoitus on antaa lukijalle perustietoa testauksesta ja siihen liittyvistä toimista.

Moduulitestauksella tarkoitetaan moduulien testausta, jotka koostuu pienistä ohjelmayksiköistä. Ohjelmayksikkö voi koostua esimerkiksi funktioista ja niistä muodostuvista pienistä kokoelmista. Käytännösssä moduulit pitävät sisällään n. 100 – 1000 riviä ohjelmakoodia. Moduulitestauksella halutaan löytää selkeät virheet, mutta myös niiden lisäksi yksiköiden ja moduulien toiminnan määrittelyjen väliset ristiriidat ja korjata ne.

Ennen moduulitestauksen suorittamista, voidaan joutua toteuttamaan ympäristöä simuloivia osia, esimerkiksi testiajureita taikka tynkämoduleita, näitä kutsutaan erillisesti testipedeiksi. Ajurit auttavat toteuttamaan palvelujen kutsumista sekä tulosten tarkastelua. Tynkämodulien tarkoitus on korvata testattavasta moduulista puuttuvat moduulit.

Integrointitestauksella tarkoitetaan yhteenkoottujen moduulien ja moduuliryhmien välisten rajapintojen toiminnan testaamista. Testauksen aikana saatuja tuloksia voidaan verrata tekniseen määrittelyyn. Integrointitestausta tehdään yleisesti samaan aikaan moduulitestauksen kanssa. Integrointitestauksessa on vaikea saavuttaa kunnollista testikattavuutta, koska ohjelmistoilla on tapana paisua iän myötä.

Kun ohjelmisto on laajentunut, niin on kaikkien tuotettujen koodirivien läpikäyminen huomattavasti vaikeampaa. Integrointitestauksessa pyritään löytämään toistensa kanssa yhteensopimattomat moduulit ja korjata ne. Testaus voidaan suorittaa joko aloittaen alimman tason moduuleista ja jatkaa siitä ylöspäin taikka sitten lähteä ylhäältä ja jatkaa alemmille tasoille.

Seuraavana on vuorossa järjestelmätestaus, jossa testataan jo koko järjestelmää ja siitä saatuja tuloksia verrataan määrittely- sekä asiakasdokumentaatioon.

Järjestelemätestausta tulisi suorittaa ohjelmankehitystyöstä mahdollisimman riippumattomat henkilöt, jolloin voidaan olla varmempia testien oikeellisuudesta.

Silloin testaushenkilöillä ei ole tietoa siitä, miten ohjelma on rakennettu ja miten sen tulisi reagoida eri tilanteissa. Järjestelmätestaukseen voi olla sisällytetty myös kenttätestaus sekä hyväksymistestaus. Tässä testausvaiheessa suoritetaan testejä myös ei-toiminnallisille ominaisuuksille esimerkiksi: kuormitus-, luotettavuus-, käytettävyystestien muodossa.

Alfa- ja betatestaus ovat testausvaiheita joissa asiakas tai mahdollinen asiakas ovat mukana. Alfa-testaus on asiakkaan suorittamaa testausta, joka tehdään toimittajan tiloissa ja siinä testataan tehdyn ohjelmiston toimintaa ja ominaisuuksia, joita sitten verrataan asiakkaan vaatimuksiin. Tätä jatketaan niin kauan kunnes tuotteen ominaisuudet on hyväksytty molempien osapuolien toimesta. Beta-testaus on sitä, kun ohjelmisto on jo valmis ja sitä jaetaan potentiaalisille asiakkaille käytettäväksi.

Asiakkaat käyttävät ohjelmistoa omissa tiloissaan ja halutessaan parantaa ohjelmiston toimintaa, ilmoittavat kehittäjille mahdollisista virheistä ja parannusehdotuksista. Tässä vaiheessa löydettyjä virheitä voidaan vielä korjata ennen kuin ohjelmisto lähtee viralliseen julkaisuun.

Regressiotestaus on testausvaihe, mikä on hyvin suurella todennäköisyydellä automatisoitu. Muuten sen läpivieminen voi tulla kalliiksi. Regressiotestauksella tarkoitetaan sitä kun jossakin testausvaiheessa huomataan virhe ja se korjataan, niin myös muut testivaiheet tulee tehdä uudestaan. Tällä varmistetaan sitä, että toisen virheen korjaaminen ei ole rikkonut ohjelmistoa tai jotain sen muuta toimintoa toisessa paikassa.

Mikäli testausta ei ole automatisoitu ja jokaisen virheen jälkeen jouduttaisiin manuaalisesti käymään läpi muut testivaiheet, joka veisi paljon kallista aikaa. Aika pystyttäisiin käyttämään paremmin hyödyksi ohjelmistonkehittämisessä ja hyvin todennäköisesti virheitä löytyisi paljon enemmän. Virheitä löytyisi enemmän, koska aikaa ei olisi riittänyt testivaiheiden kunnolliseen läpikäymiseen.

Käytettävyystestauksen tarkoitus on testata sitä että ohjelmiston käyttäjä pystyy suorittamaan mahdollisimman hyvin ja helposti, hänelle annetut tehtävät järjestelmän kanssa. Tätä vartenhan ohjelmistoa ollaan toteuttamassa. Testattavina kohteina ovat käyttöliittymä ja sen toiminta.

Yleensä näitä testauksia on suoritettu pienellä joukolla henkilöitä, jotka ovat järjestelmän tulevia käyttäjiä. Koekäyttäjät kutsutaan erilliseen tilaisuuteen, jossa he suorittavat halutut testit ryhmissä taikka yksittäin. Koekäyttäjiä pyydetään ajattelemaan ääneen testiä tehdessään, koska yleensä nämä testit videoidaan käyttäjäkohtaisesti. Kun testit on videoitu käyttäjäkohtaisesti, niin ne pystytään analysoimaan paremmin.

Nykyään pystytään käyttöliittymän toimintaa testaamaan hyvin jo siihen tarkoitetuilla automatisointityökaluilla ja käyttäjänkin toimia pystytään simuloimaan tiettyyn pisteeseen saakka. Tällaisten koetilaisuuksien suurin anti on kuitenkin saatu palaute.

Vaikka miten pystyttäisiin simuloimaan käyttäjän toimia, niin siihen ei pystytä käyttäjien ajatuksien tasolla. Tällöin jokin outo ratkaisu, joka ohjelmistonkehittäjästä ei käy mitenkään järkeen voi olla käyttäjälle sitten taas aivan luontainen. Myöskään käyttöliittymän ulkoasun toimivuutta on vaikea testata ja sen kehittämiseen tarvitaan käyttäjien mielipiteitä, jotta päästäisiin parempaan lopputulokseen ja ohjelmistosta kehittyisi hyvin toimiva helposti käytettävä.

4 TESTITAPAUSTEN VALINTA

4.1 Staattinen ja Dynaaminen analyysi

Staattisen analyysin tarkoitus on etsiä ohjelmasta puutteita ja virheitä ilman että sitä suoritetaan. Staattisen analyysin suorittaminen voi tapahtua joko ohjelmallisesti

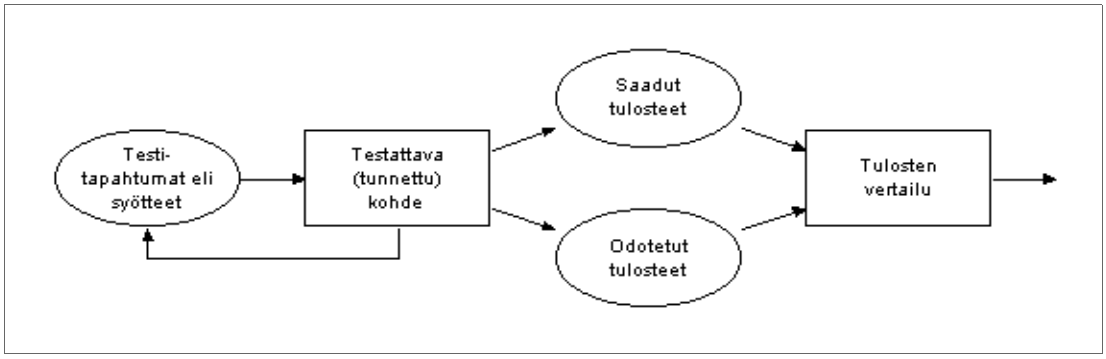
taikka käsin. Analyysin avulla voidaan löytää virheitä esimerkiksi funtioiden, muuttujien taikka osoittimien väärinkäytöstä.

Dynaaminen analyysi sitten taas suoritetaan silloin kun ohjelma on käynnissä ja sen toimintaa testataan. Dynaaminen analyysin suorittamiseen tarvitaan siis toimiva ohjelma taikka ympäristö missä näitä testejä pystytään suorittamaan. Samalla voidaan testata sellaisia ominaisuuksia joiden testaaminen staattisen analyysin kanssa olisi vaikeaa, osaa jopa mahdoton testata.

4.2 White box

White box -testauksessa testaaja pääsee tutustumaan ohjelmakoodiin ennen varsinaista testausta, josta on hyötyä testaajalle testien suorittamisessa. Tutustuttuaan ohjelmakoodiin, voi testaaja tehdä johtopäätöksiä siitä miten ohjelma toimii ja mahdollisesti johtaa ohjelmassa virhetilanteeseen. Testaaja pystyy myös muokkaamaan suorittamaansa testausta siten, että mahdollisimman paljon virhetilanteita tulisi esiin. Toisaalta tässä mallissa testaaja pystyy myös valitsemaan ne arvot tai tilanteet, joissa tietää ohjelman toimivan normaalisti.

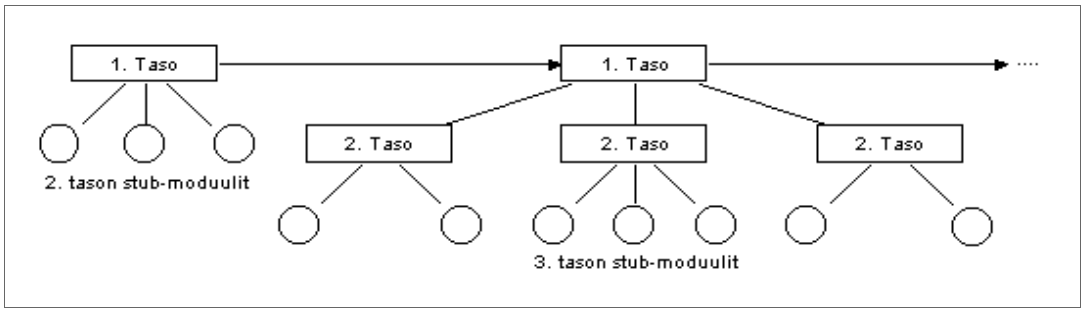
Tällöin mahdollisia virheitä ei löydetä välttämättä niin tehokkaasti, koska testaaja voi pyrkiä vain suorittamaan testit mahdollisimman nopeasti ja virhetilanteita ei synny niin paljoa. White box testauksessa pyritään siihen, että kaikki mahdolliset ohjelmapolut tulisi käytyä läpi testien aikana ja sen mukaan valitaan testitapaukset näille testeille.



Kuva2. White box -testaus(Kautto 1996.)

4.2.1 Top-down

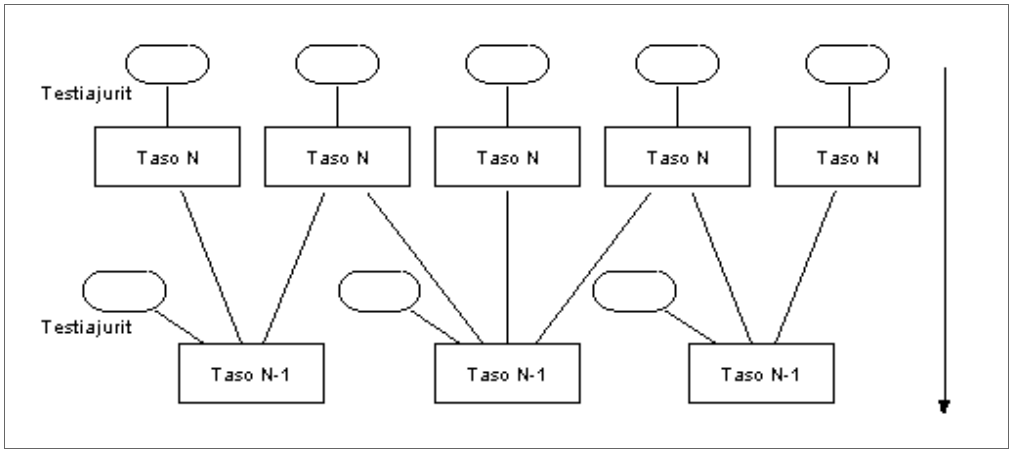
Top-down strategiaksi kutsutaan lähestymistapaa, joka aloitetaan sovelluksen moduuleista. Moduulit sijaitsevat korkeammalla tasolla ja siitä edetään systemaattisesti alemmille tasoille ja niiden moduuleihin. Tätä lähestymistapaa kutsutaan top-down -strategiaksi. Tässä strategiassa voidaan simuloida alempien tasojen moduuleita rakentamalla ”tyhmiä” moduuleita tai niiden pätkiä, joiden avulla voidaan testaus suorittaa. Näihin mahdollisiin lisättyihin moduuleihin ei toteuteta mitään monimutkaista toiminnallisuutta, vaan ne pyritään pitämään mahdollisimman yksinkertaisina. Aina kun testaus etenee seuraavalle tasolle, voidaan korvata tehdyt ”tyhmät” moduulit oikeilla moduuleilla.



Kuva 3. Top-down -testaus (Kautto 1996.)

4.2.2 Bottom-up

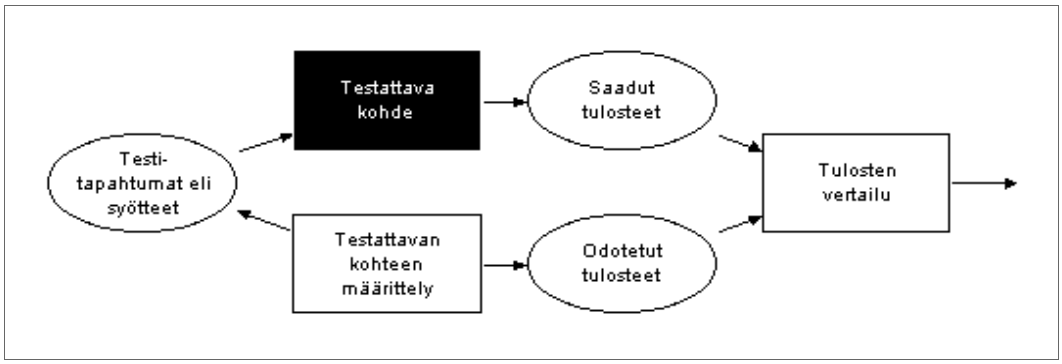
Testaus voidaan myös aloittaa alimman tason moduuleista ja siitä jatkaa systemaattisesti ylemmille tasoille, tätä kutsutaan bottom-up -strategiaksi. Toisin kuin top-down strategiassa, niin tässä strategiassa luodaan ajureita. Ajureiden avulla tehdään ja suoritetaan halutut testit. Yleisenä käytäntönä on tehdä testauksen mahdollistava ympäristö, jota käytetään kaikkien moduulien yhteisenä testiajurina. Käytetään siis yhtä ajuria eikä luoda montaa ajuria eri moduuleille. Suurimpana ongelmana bottom-up -testauksessa on se, että alimman tason moduulit pitää suunnitella ja määritellä ennen kuin testaus voidaan aloittaa. Toisena ongelmana pidetään sitä, että ohjelma ei ole valmis, ennen kuin kaikki sen moduulit on testattu ylintä tasoa myöden.



Kuva 4. Bottom-up -testaus (Kautto 1996.)

4.3 Black box

Testaaja tietää vain sen, mitä ohjelman pitäisi tehdä. Kun hän antaa tietyn syötteen, hän saa tietyn tulosteen takaisin. Testaajalla ei ole tietoa siitä, miten ohjelma on päätynyt antamaansa tulosteeseen.



Kuva 5. Black box testaus (Kautto 1996.)

Black box -testauksessa testitapaukset valitaan ekvivalenssiluokista, jotka koostuvat eri syöteavaruuksista. Ajatuksena on se, että oletetaan ohjelman toimivan samalla tavalla kaikkien saman luokan edustajien kanssa. Ohjelman toimiessa yhdellä luokan edustajalla, voidaan sen olettaa toimivan myös muilla luokan edustajilla. Luokkien jako voidaan esimerkiksi suorittaa seuraavasti: 1. kelvolliset syötteet, 2. liian pienet taikka suuret syötteet ja 3. epäkelvot syötteet(ei kokonaisluvut).

Tapauksen esimerkkiohjelmana toimii laskuohjelma, jonka voidaan olettaa toimivan oikein, jos sille syötetään 1+1 ja saadaan tuloksena 2. Voitaisiin siis olettaa että ohjelma toimisi oikein myös arvoilla 7+4 tai 9+16. Testeihin voitaisiin myös valita arvoja luokkien rajoilta. Jos testi epäonnistuu raja-arvolla, niin se tulisi

epäonnistumaan myös muilla rajojen sisällä olevilla arvoilla. Tällaisia tapauksia kutsutaan raja-arvoanalyyseiksi.

Testaajalla voi olla kokemuksia erityyppisistä ongelmia aiheuttavista syötteistä, jotka eivät kuulu edellä mainittuihin syötteisiin. Nämä syötteet ovat aiheuttaneet aikaisemmin ongelmia ohjelmissa, jolloin testaaja osaa ottaa myös nämä huomioon testiä tehdessä. Testaaja voi siis listata kaikki mahdolliset mieleen tulevat testitapaukset, joilla nykyisen testin kohde saataisiin epäonnistumaan.

5 RIITTÄVYYDEN ARVIOINTI

Onko jotain keinoa, millä pystyttäisiin määrittämään se milloin on ohjelmaa testattu tarpeeksi? Testausta voitaisiin jatkaa niin pitkään kuin vain aikaa ja rahaa riittää, joten yleensä päädytään tekemään kompromissiin tuotteen vikojen aiheuttamien kustannusten ja ohjelman myöhästymisestä aiheutuvien tuottojen menetyksen välillä. Minun mielestäni tämä on jossain suhteessa jo väärä lähtökohta, koska on hyvin todennäköistä että kehitettävästä tai kehitetystä tuotteesta tulee löytymään virheitä.

Eikä kaikkia virheitä ole mahdollista löytää ilman, että tuote julkaistaisiin ja päästetään muutkin käyttämään sitä. Liian paljon tapahtuu sitä, että tuotetta hiotaan, testataan, hiotaan, testataan. Tätä oravanpyörää toistetaan x kertaa, jonka jälkeen tuote julkaistaan ja sieltä löydetään silti sellaisia virheitä mitkä eivät aikaisemmin tullut vastaan.

Mielestäni tulisi mahdollinen kehityksessä oleva tuote saada mahdollisimman pian saada julkaistua ja saada sille käyttäjiä, joiden palautteen perusteella voidaan ohjelman toiminnallisuutta parantaa ja vastaantulevia virheitä korjata. Samalla myös mahdolliset käyttäjät pääsevät sisälle tuotteeseen ja näkevät sen mihin suuntaan tuote kehittyy palautteen ansiosta. Palatakseni vielä vähän taaksepäin, niin testauksen määrää voidaan yrittää arvioida erilaisilla mutkikkuusmitoilla ja riittävyyttä erinäisillä kattavuusmitoilla ja virheiden kylvämisellä.

5.1 Mutkikkuusmitta

Analysoimalla ohjelmamoduulin koodin rakennetta pyritään päättelemään sen monimutkaisuutta. Voidaan yrittää paikantaa paljon testausta tarvitsevat moduulit. Tällaisia mutkikkuusmittoja ovat mm. Halsteadin mitta ja McCaben syklomaattinen mitta. (Haikala & Märijärvi 2006, 294.)

”Halsteadin mitta määräytyy seuraavasti. Lasketaan ohjelmassa olevien operaattoreiden ja operandien yhteinen lukumäärä N (operaattoreita ovat varatut sanat ja +,-m jne.). Tämän jälkeen lasketaan ohjelmassa käytettyjen eri operaattoreiden ja operandien yhteinen lukumäärä n. Halsteadin mitta ohjelmalle on tällöin Nlog2n.”(Haikala & Märijärvi 2006, 294.)

”McCaben syklomaattinen numero lasketaan yleensä erikseen ohjelman jokaiselle funktiolle. Sen arvo saadaan lisäämällä funktion kontrolliverkon haarautumiskohtien lukumäärään ykkönen. Saatu luku kuvaa funktion kontrolliverkon monimutkaisuutta. Testauksen kannalta tarkasteltuna sen arvoa voi pitää minimimäärää testitapauksia, joilla ko. funktiota on testattava.” (Haikala & Märijärvi 2006, 294.)

5.2 Riittävyyden arvioinnissa käytettyjä kattavuusmittoja

Käyttämällä erilaisia kattavuusmittoja, pyritään varmistamaan sitä, että kaikki testiohjelmiston osat tulisi testatuksi kattavasti. Niiden avulla pystytään myös osoittamaan, että testausta on suoritettu riittävä määrä. Seuraavaksi tulen esittelemään joitain kattavuuksia, joita käytetään ohjelmiston koodin testauksessa.

Lausekattavuuden avulla lasketaan sitä, kuinka moni ohjelmistossa oleva ohjelmakoodirivi tulee suoritetuksi vähintään kerran. Täydelliseen lausekattavuuteen on erittäin vaikeaa päästä. Vaikka täydellisen lausekattavuuden saavuttaminen voi vaikuttaa kohtuullisen helpolta, niin silti yleisesti saavutetaan vain n. 90% lausekattavuus.

Päätöskattavuudella tarkoitetaan sitä, kun ohjelmistokoodin jokaisen ehtorakenteen päätöksen arvot saadaan toteutumaan vähintään kerran testien aikana. Ehtokattavuudella tarkoitetaan sitä, kun ehtorakenteen kaikki osaehdot saavat

molemmat arvonsa. Moniehtokattavuudella taas tarkoitetaan sitä, että testaus suoritetaan kaikkien osaehtojen kaikilla eri kombinaatiovaihtoehdoilla.

Polkutestauksen ajatuksena on, että kehitetty ohjelma on yksi suuri verkko, jonka solmuina toimivat sen haarautumiskohdat. Tämän verkon lävitse yritetään etsiä kaikki mahdolliset reitit ohjelman läpikäymiseen. Käytännössä mahdollisia reittejä on ääretön määrä ja kaikkia löydettyjäkään polkuja ei pystytä käyttämään. Seuraus tästä on se, että täydelliseen polkukattavuuteen ei koskaan päästä.

Virheiden kylväminen on tapa, jossa ohjelmakoodiin syötetään virheitä tahallisesti. Tällä tavalla pystytään testauksen aikana löydettyjen virheiden määrästä päättelemään suurinpiirtein niiden virheiden määrä, jotka ovat jääneet ohjelmaan. Jos ohjelmassa on X kylvettyä ja Y oikeaa virhettä ja testauksen aikana kumpiakin löydetään määrät X ja Y.

Oletetaan, että todellisista ja kylvetyistä virheistä on löydetty osuus on yhtä suuri, niin pystytään arvioimaan todellisten virheiden määrä. Tätä voi viedä vielä pidemmälle, mikäli tekee ohjelmasta eri versioita, joissa kylvetyt virheet eivät ole samoja edellisten versioiden kanssa. Silloin tätä tapaa voidaan kutsua mutaatiotestaukseksi.

Käytännössä virheiden kylväminen ei ole käytetty tapa, koska se aiheuttaa kohtuullisen paljon ylimääräistä työtä. Aina kun tällaista tapaa käytetään, on tärkeää tehdä tarkat suunnitelmat. Miten varmistetaan kaikkien tahallisesti kylvettyjen virheiden poistaminen ohjelmasta ja miten se todistetaan. Kylväminen kuitenkin tapahtuu ihmisen toimesta ja vaikka kuinka jokainen kylvetty virhe dokumentoitaisiin, niin inhimillinen virhe voi tapahtua dokumentoitaessa taikka virheiden poiston yhteydessä. Itse en suosittelisi kyseistä tapaa käytettäväksi ohjelmistotestauksessa.

6 YLEISTÄ AUTOMATISOIDUSTA TESTAUKSESTA JA MENETELMISTÄ

Kerroin aiemmin ohjelmiston täydellisesti kattava testaus on erittäin haastavaa ja käytännössä mahdotonta. Tämä johtuu siitä, että nykyään ohjelmistot ovat niin laajoja kokonaisuuksia. On arvioitu että, ohjelmistoprojektin budjetista noin 25 – 50 prosenttia kuluu testaukseen, siihen sisältyy kehittäjien tekemä manuaalinen ja automatisoitu testaus. Huolimatta siitä, että testauksen osuus ohjelmistoprojektin budjetista on näin merkittävä, niin Yhdysvalloissa julkaistujen ohjelmistojen virheet aiheuttivat vuonna 2002 59,5 miljardin dollarin laskun. Tehdyn tutkimuksen mukaan voitaisiin vuosittain säästää 22,2 miljardia dollaria vain sillä, että parannetaan testaamista. (Mäki. 2002.)

Yleensä ratkaisu testauksen kattavuuden suurentamiseksi on automatisoitu testaus. Automatisoinnin avulla pystytään suorittamaan testejä enemmän, nopeammin ja pidempiä aikoja kuin tavallisella manualisella testauksella. Testauksen pystyy automatisoimaan, mutta testitapausten luonti sekä testitulosten analysointi on tehtävä ihmisten toimesta. Milloin kannattaa automatisoida? Minun mielestäni testaus kannattaa aina automatisoida, koska sillä säästetään aikaa ja resursseja johonkin toiseen vaiheeseen. Automatisoinnilla saadaan myös etua siitä, että voidaan suorittaa testitapausten testaaminen useita kertoja eri ohjelmiston versioille. Vaikka yhden testitapauksen kirjoittaminen veisi aikaa yhtä paljon kuin testitapauksen suorittaminen kolmekymmentä kertaa manuaalisesti, niin pitkässä juoksussa se on taloudellisesti kannattavampaa automatisoida.

Mitä mahdollisia ongelmia on testauksen automatisoinnissa? Yleensä aina uuden teknisen ratkaisun tultua, uskotaan että se tulee ratkaisemaan kaikki nykyiset ja tulevat ongelmat. Näin se ei kuitenkaan ole tässäkään asiassa. Mikäli testaajilla ei ole kokemusta testaamisesta, on mahdollista että testit tehty huonosti. Huonosti tehdyillä testeillä ei löydetä haluttuja virheitä ohjelmasta. Automatisoidun testauksen uskotaan löytävän virheitä paljon uusia virheitä, mutta pääasiassa virheet löytyvät tehtyjen testien ensimmäisellä ajokerralla, jonka jälkeen ne korjataan. Virheet korjataan ja

testit ajetaan uudestaan, eikä virheitä enää löydy ennen kuin tehtyä ohjelmakoodia muutetaan ja sitä testataan testien avulla.

Vaikka testit menisivätkin lävitse, niin siltikään ei voida olla aivan varmoja, ettei virheitä ole. Kuten jo aikaisemmin mainitsin, niin myös itse testit voivat olla virheellisiä taikka epätäydellisiä testaamaan haluttua asiaa. Täytyy myös muistaa se, että kun ohjelmakoodia muutetaan, niin tulee myös muutettua ohjelmakoodiosuutta koskevat testit muokattava vastaamaan muutettua koodia. Siksi onkin tärkeää, että automatisoitujen testien ylläpidosta tehdään mahdollisimman helppoa, jotta siitä ei tule ns. pakkopullaa sovelluksen kehittäjille taikka testaajille.

Mitä tarvitaan, jotta testausta voidaan lähteä automatisoimaan? Ensimmäinen tapa miten voidaan lähteä automatisoimaan testausta, tarvitsee olla valmiina tarkat sekä yksityiskohtaiset testitapaukset ja niistä odotetut tulokset. Tulokset perustuvat vaatimusmäärittelyihin ja suunnitteludokumentteihin. Mielestäni em. tapa soveltuu enemmän perinteisen mallilla tehtävän ohjelmiston testauksen automatisointiin, mutta nykyisten käytössä olevien kehitysmenetelmien kanssa sen käyttäminen on hidasta ja jäykkää. Myöskään em. tapa ei mielestäni sovellu täysin web-sovelluksen käyttöliittymän testauksen kanssa käytettäväksi.

Toinen lähestymistapa on se, että aloitetaan suoraan tekemään testejä sekä ohjelmistoa ilman mitään suurempaa vaatimusmäärittelyitä taikka suunnitteludokumentteja tai pyritään pitämään ne minimissään. Kun määrittely- ja suunnitteludokumentointi pyritään pitämään minimissä, pystytään reagoimaan tuleviin muutoksiin. Tällaista menetelmää kutsutaan myös ketteräksi ohjelmistokehitykseksi, joka koostuu useasta kehitysmenetelmästä mm. Extreme Programming (XP), Scrum ja Pragmatic Programming.

Ketterillä kehitysmenetelmillä pyritään vähentämään ohjelmistokehityksen riskejä jakamalla se lyhyisiin iteraatioihin, jotka ovat kestoltaan yhdestä neljään viikkoon. Jokaiseen iteraatioon kuuluu uusien toimintojen suunnittelu, vaatimusanalyysi, testaus, ohjelmointi sekä mahdollinen dokumentointi. Iteraation loppuessa pyritään aina siihen, että ohjelmisto on julkaisukelpoinen riippumatta siitä miten paljon uutta

toiminnallisuutta siihen on lisätty ja mietitään projektin prioriteetteja, joiden avulla pystytään päättämään seuraavan iteraation sisällöstä. Ketteriin menetelmiin kuuluu suora viestintä, joka tapahtuu mielellään kasvokkain ja dokumenttien kirjoittamista pyritään vähentämään.

Ketterissä menetelmissä dokumentaatiolla ei ole samaa arvoa kuin perinteisimmissä malleissa, vaan ensisijaisestti edistymisen mittarina toimii toimiva sovellus tai ohjelmisto. Kun dokumentoinnilla ei ole niin suurta osaa tai arvoa, niin yleisiä harhaluuloja ovat esimerkiksi: että kehitys muuttuisi jotenkin kurittomammaksi tai että projektin aikana ei tehtäisi suunnittelua ollenkaan. Suunnittelua tehdään koko projektin ajan, mutta toisin kuin perinteisemmissä kehitysmenetelmissä, ollaan suunnitelmia valmiita muuttamaan nopeallakin aikataululla, mikäli niin vaaditaan.

6.1 Kehitysmenetelmiä

TDD eli Test-Driven development on ohjelmistokehitysmenetelmä, jossa luodaan ensiksi testitapaus tai -tapaukset ja vasta sitten tehdään taikka muutetaan ohjelmakoodia ja sen toimintaa. Ennen kuin ensimmäistäkään riviä uutta toiminnallisuutta tehdään, tulee olla sitä vastaavat testit tehtynä. Tällä menetelmällä pyritään varmistamaan sitä, että ohjelmisto toimii oikein. Samalla kun kirjoitetaan testejä, saadaan jatkuvasti laajeneva verkosto, jonka avulla uusien toimintojen tekeminen sekä vanhojen muokkaaminen on turvallisempaa.

Kirjoitettuja testejä suorittamalla pystytään helposti havaitsemaan uusia virheitä, jotka voivat mahdollisesti johtua aikaisemman virheenkorjauksesta. TDD kulkee yleensä käsikädessä ketterien kehitysmenetelmien kanssa ja sen tarkoituksena on auttaa suunnittelussa ja toteutuksessa, jotta saadaan aikaan vaatimukset täyttävä ohjelmisto. TDD:tä käytettäessä ohjaudutaan malliin, jossa ajatellaan ensin ja tehdään vasta sen jälkeen. Kun tehdään testit ensiksi, voidaan niiden pohjalta tehdä toiminnallisuus ohjelmistoon testien sisältämien vaatimusten mukaan.

TDD koostuu käytännössä kolmesta osasta. Ensimmäinen näistä on se, että kirjoitetaan testi taikka useita testejä ja ajetaan ne. Testit eivät mene läpi. Toiseksi tehdään testejä vastaava ohjelmakoodi ja sitä voidaan muokata niin kauan, että testit

menevät läpi. Kolmannessa vaiheessa refaktoroidaan tuotettu koodi eli siivotaan ja rakennetaan se uudestaan. Näitä kolmea vaihetta toistetaan niin kauan kuin on tarvetta.

Refaktorointi on tärkeä osa tätä kehitysmenetelmää ja sillä pyritään yksinkertaistamaan ohjelmakoodin rakennetta sisältä niin, että ulkoinen toiminta ei muutu ollenkaan. Ohjelmakoodin siistimisen yhteydessä etsitään ja poistetaan taikka korjataan ns. tuplakoodeja sekä ”purkkaviritelmiä”, joilla ohjelmiston jokin toiminto on saatu aikaiseksi. Näillä toimenpiteillä saadaan ohjelmakoodista selkeämpää, jolloin sitä on helpompi ymmärtää sekä ylläpitää ja nämä samat em. asiat koskevat myös tehtyjä testejä.

Jotkut saattavat ajatella, että on hullua lähteä purkamaan ja muuttamaan jo valmiiksi toimivaa koodia. Muutettu koodi saattaa rikkoa jonkin osan ohjelmasta, mutta tätä ei mielestäni tulisi pelätä koska lähes kaikkeen tehtyyn koodiin löytyy jossain vaiheessa helpompi ja paremmin toimiva ratkaisu. Uuden ratkaisun käyttöönottaminen yleensä parantaa ohjelman toimintavarmuutta sekä laatua. Mielestäni olisi hulluutta jättää tuo asia tekemättä vain siksi, ettei haluta koskea jo tehtyyn koodiin vaikka se olisikin toimivaa.

Miksi ei käytettäisi TDD:tä? Kuten muutkin muutokset rutiineihin aiheuttavat yleensä vastarintaa kehittäjien joukossa. Joitain yleisiä olettamuksia on, että TDD olisi jotenkin kehitystä hidastavaa vaikka sillä pyritään juuri kehitysprosessin parantamiseen. TDD sanotaan vaikeuttavan kehittäjän pääsyä ns. flow -tilaan vaikka ainakin omasta mielestäni on helpompi päästä flow -tilaan ohjelmakoodia tehdessä, jos on jotain mitä vasten tulevaa koodia voi testata ja millaisia vaatimuksia on tullut esille.

Sanotaan myös, että testien kirjoittaminen olisi vaikeaa taikka työlästä. Tämäkään ei pidä paikkaansa koska testien kirjoittamisen voi aloittaa helpommista ja pienemmistä osista ja niitä tekemällä oppii tekemään myös vaikeampia testejä. Ehkä tämän olettamuksen takana on ajatus, että testien kirjoittaminen tuo paljon ylimääräistä

työtä. Voi toki olla niinkin, että halutaan vain vastustaa muutosta siitä syystä ettei siihen ole totuttu.

ATDD eli Acceptance Test Driven Development on osa tai lisä äsken mainitsemaani test-driven developmentiin, joka auttaa sovellukehittäjiä rakentamaan korkealaatuisia ohjelmistoja jotka täyttävät myös liiketoiminnalliset tarpeet. Äsken maintittu TDD auttaa varmistamaan teknisen laadun. ATDD auttaa sovelluskehitysprojekteja koordinoimaan toimintaansa siten, että pystytään tuottamaan juuri sellainen sovellus kuin asiakas haluaa, milloin haluaa. ATDD:ssa pyritään pienentämään kehittäjien mahdollisuutta toteuttaa vaadittuja toiminnallisuuksia vain puolittain. Tärkeä osa menetelmää on se, että sitä toteutetaan ryhmänä ja kommunikointi ryhmän sisällä on hyvin tärkeää

BDD eli Behaviour Driven Development on kehitysmenetelmä, joka kannustaa kommunikointiin kehittäjien, laadun varmistajien ja muiden projektiin osaaottavien kesken. Tällä menetelmällä pyritään löytämään mahdollisimman selkeä kuva kaikkien projektiin kuuluvien henkilöiden asiakas mukaan luettuna, että miten ohjelmiston tai sovelluksen tulee käyttäytyä keskustelujen avulla. Aikaisemmin mainitsemaani TDD:tä laajennetaan testitapausten kirjoittamista tavallisella kielellä, jota myös ei tekniset henkilöt pystyvät lukemaan. Näin kehittäjät voivat keskittyä enemmän siihen, miksi ohjelmakoodi tulisi tehdä eikä teknisiin yksityiskohtiin. Pystytään minimoimaan käännökset teknisen kielen ja muiden eri ryhmien välillä.

6.2 Continuous integration

Continuous integration (CI) toteuttaa jatkuvia laadun tarkkailu prosesseja, pieniä vaikutuksia joita sovelletaan usein. Prosesseilla pyritään parantamaan sovelluksen laatua ja pienentämään aikaa, joka kuluu sen julkistamiseen taikka toimitukseen. Laatua parannetaan korvaamalla käytäntö, jota käytetään perinteisissä menetelmissä. Perinteisissä menetelmissä laadun tarkkailu aloitetaan vasta kun koko kehitys on tehty, jolloin se vie aikaa ja vaikeus kasvaa huomattavasti. Tällöin projektin alussa tulee kiinnittää tarpeeksi huomiota integrointiin ja sen tuomiin haasteisiin.

CI:ssa sovellusta suoritetaan ja integroidaan kokoajan. Sovelluksen kehitystä ohjataan sellaiseksi, että se on julkaistavissa nopeasti. Mitä suurempi projekti on kyseessä ja sen parissa työskentelee useita tiimejä, niin CI helpottaa varmistumaan eri tiimien komponenttien yhteensopivuudesta. Kun eri tiimien tekemiä komponentteja aloitetaan yhdistämään, niin ei pitäisi suuren suuria yllätyksiä tulla vastaan. CI:n vaiheisiin kuuluu ne toimenpiteet, jotka tarvitaan tuotteen julkaisemiseen.

Yksinkertaisimmillaan CI tarkoittaa uusimman sovellusversion hakemista, kääntämistä ja sen jälkeen siirtämistä asennushakemistoon tai palvelimelle. Sitten on myös monimutkaisempia järjestelmiä, jotka suorittavat samassa yhteydessä automatisoidut testit, kääntää tehdyn ohjelmakoodin ja alustaa palvelimen sekä asentaa servicet paikoilleen.

7 TYÖKALUT

7.1 Selenium

Selenium on avoimen lähdekoodin ohjelmisto, jolla pystytään testaamaan selainpohjaisten sovellusten käyttöliittymää ja sen toimintaa. Tällä hetkellä se pitää sisällään seuraavat projektit: Selenium IDE, Selenium Remote Control, Selenium WebDriver sekä Selenium Grid. Näistä tulen erikseen kertomaan seuraavissa otsikoissa. Selenium oli ensimmäinen työkalu, johon tutustuin tämän opinnäytetyön puitteissa ja täytyy sanoa, että alusta lähtien se teki hyvän vaikutuksen laajuudellaan ja helposti omaksuttavana kokonaisuutena.

7.1.1 Selenium IDE

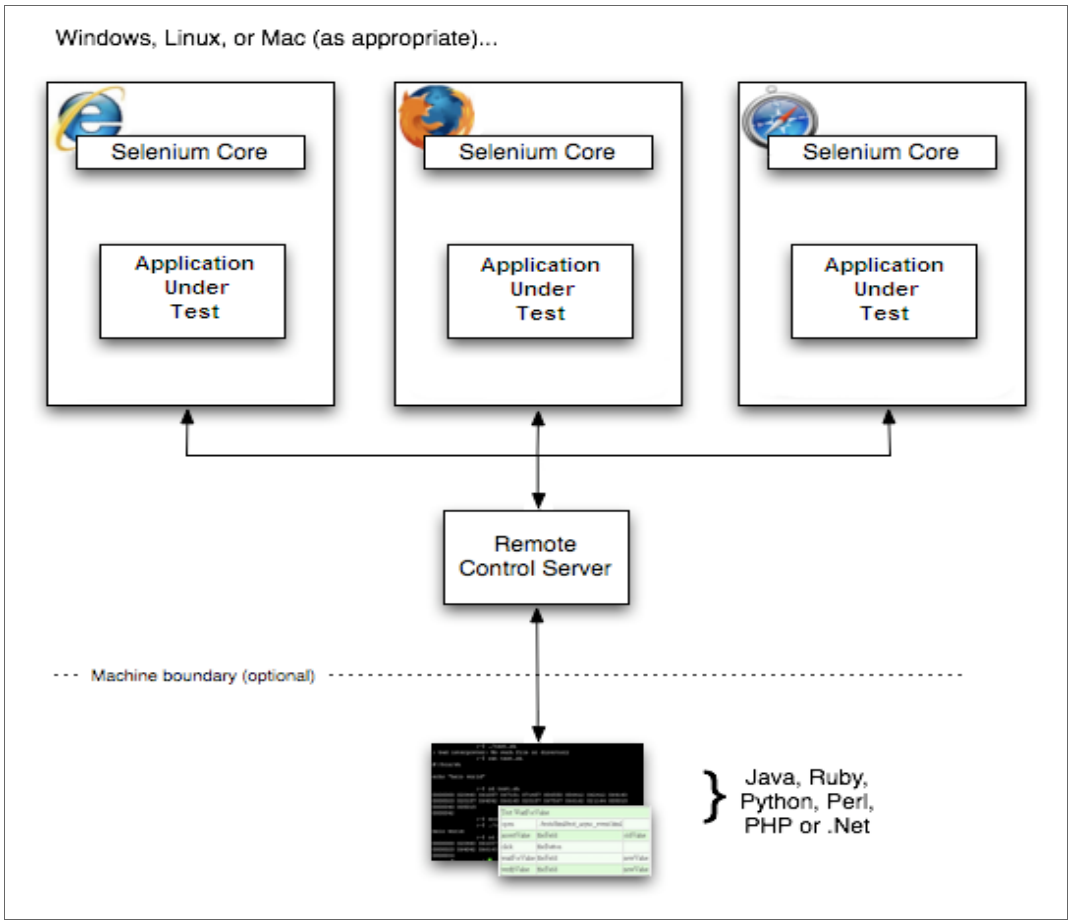
Selenium IDE on lisäosa Mozillan Firefox selaimelle, joka on tällä hetkellä versiossa 1.2.0. Tällä työkalulla pääsee helposti alkuun web-pohjaisen sovelluksen käyttöliittymän testauksessa. Se on helppokäyttöinen, joten sen opetteluun ei kulu paljoa aikaa. Lisäosan asennuksen jälkeen pystyy nauhoittamaan sekä toistamaan

sillä tehtyjä testitapauksia yksittäin taikka sitten niistä voi muodostaa testsuiten, johon erilliset testitapaukset liitetään.

Nauhoitetut testit voidaan tallentaa esimerkiksi html tai ruby -script muodossa. Mikäli testejä halutaan muokata toisessa ympäristössä taikka toisella työkalulla, voidaan tallennetut testit muuntaa eri muotoon. Esimerkkeinä c#, jUnit ja python. Itse aloitin työn käytännön osuuden juuri Selenium IDE:lla, jolla nauhoitin perustestisetin ja samalla pääsin tutummaksi Seleniumissa käytettävän kielen kanssa.

7.1.2 Selenium 1 (Remote Control)

Selenium 1 eli RC koostuu kahdesta komponentista, joista ensimmäinen on Selenium server. Sillä pystytään avaamaan ja sulkemaan selaimia. Serverin kautta myös ajetaan Selenium komennot javascriptin avulla selaimelle, joka palauttaa vastaukset takaisin serverille. Toinen komponentti on kirjastot, jotka toimii rajapintana eri ohjelmointikielien ja serverin välillä.



Kuva 6. Selenium RC

Hieman tarkemmin Selenium serveristä. Serveri vastaanottaa komentoja käytetyltä testiohjelmalta tulkitsee ne ja raportoi testien tulokset takaisin testiohjelmalle. Serveri paketoi Selenium Coren ja automaattisesti injektoi sen selaimeen. Tämä tapahtuu silloin, kun testiohjelma avaa selaimen. Selenium Core on javascript ohjelma tai oikeastaan kokoelma javascript funktioita, jotka tulkitsee ja suorittaa Selenium komentoja käyttäen hyväksi selaimeen rakennettua javascript tulkkia. Serveri saa Selenium komennot testiohjelmalta käyttämällä yksinkertaisia HTTP GET eli POST kutsuja. Tämä tarkoittaa sitä, että Selenium selain testit pystytään automatisoimaan millä tahansa HTTP kutsuja lähettävällä ohjelmointikielellä.

Kirjastot tarjoavat ohjelmointituen, jonka avulla pystytään ajamaan Selenium komentoja eri ohjelmista. Jokaiselle tuetulle ohjelmointikielelle on omat kirjastot, mitä pystytään käyttämään. Ne tarjoavat ohjelmointirajapinnan (API), jolla näitä komentoja pystytään ajamaan. Kirjastot ottavat vastaan Selenium komennon ja syöttävät sen Selenium serverille, joka prosessoi siitä tietyn tapahtuman taikka testin ohjelmaa vasten. Se myös vastaanottaa tulokset ja varastoi ne ohjelmassa muuttujaan sekä raportoi testin onnistumisesta tai epäonnistumisesta. Mahdollisessa virhetilanteessa voi kirjasto myös tarjota apua virhetilanteen korjaamiseksi.

Haluan myös mainita, että vaikkakin Selenium 1 on edelleen käytetty kieli ja tapa web-sovellusten testien aikaansaamikseksi, niin suosittelen siirtymään käyttämään Selenium 2:sta. Selenium 2:n kehitys sekä tuki jatkuu tulevaisuudessa ja Selenium 1 tulee jäämään taka-alalle.

7.1.3 Selenium 2 (WebDriver)

Selenium 2 eli WebDriver, on pääasiassa uudistettu versio vanhasta Remote Controllista ja suurimpana uudistuksena on integroituna oleva WebDriver API. WebDriver API tarjoaa vaihtoehtoisen ja helppokäyttöisemmän ohjelmointirajapinnan sekä parantaa useita rajoitteita, joita vanhassa versiossa oli. Tarkoituksen on ollut kehittää objekti-orientoitunut rajapinta, joka tarjoaa lisätuen suurelle määrälle selaimia ja helpottaa työskentelyä kehittyvien web-sovellusten testausongelmien kanssa.

Koska WebDriver käyttää aivan eri teknologiaa kommukoimiseen selaimen kanssa, niin Selenium serverin käyttäminen ei ole pakollista. Webdriver tekee suoria kutsuja selaimelle käyttäen hyväksi selaimen natiivia tukea. Mikäli aiot käyttää myös toiminnallisuuksia, jotka tarvitsevat Selenium rc:in, niin silloin tulee käyttää myös Selenium serveriä. Toinen syy Selenium serverin käyttöön on se, että halutaan käyttää Selenium Grid:iä testien suorittamiseen jaetusti.

Mikäli sinulla on vanhemmalla Selenium versiolla tehtyjä testejä, niin ne pystytään muuttamaan uuden version vaatimaan muotoon. Täytyy kuitenkin ottaa huomioon, että testien konvertoiminen ei aina käy aivan kivuttomasti. Syitä miksi vanhat testit kannattaisi muuttaa uuteen muotoon: saataisiin pienempi ja kompaktimpi rajapinta. Työskenteky helpottuisi sekä käyttäjän toimintoja voitaisiin paremmin jäljitellä.

Uudemmalla versiolla pystytään jäljittelemään myös kehittyneempiä käyttäjän toimintoja sekä saadaan tuki selaimien kehittäjiltä. Opera, Mozilla ja Google ovat kaikki aktiivisesti mukana WebDriverin kehityksessä ja tämä tarkoittaa sitä, että usein tuki WebDriverille on lisätty selaimeen itseensä. Kun tuki WebDriverille on lisätty suoraan selaimeen, niin pystytään testit suorittamaan nopeammin ja ne ovat vähemmän herkkiä rikkoutumaan. Oli tuki sitten suoraan selaimessa taikka käytössä palvelun kautta, niin niitä kutsuttiin yhteisellä nimityksellä WebDrivereiksi.

7.1.4 Selenium Grid

Selenium Grid on tarkoitettu testien ajamiseen samanaikaisesti, jolloin testeihin kuluvaa kokonaisaikaa saadaan lyhennettyä. Grid on kehitetty käytettäväksi Selenium RC:n kanssa, mutta ilmeisesti Grid 2 versio on työn alla ja siihen olisi sisällytetty myös tuki WebDriverille. Grid 2 -versiosta ei ole vielä tehty virallista julkistusta, mutta kenties sekin tässä lähitulevaisuudessa tullaan näkemään.

7.2 WatiR

WatiR on myös avoimen lähdekoodin ohjelmisto, jolla kyetään testaamaan web-pohjaisia sovelluksia selaimen avulla. Käytännössä se on kokoelma Ruby kirjastoja. Kirjastot sallivat testien kirjoittamisen, jotka ovat helppolukuisia ja ylläpidettäviä.

WatiR ”ajaa” selaimia samalla tavalla kuin sitä käyttävä tekisi, se klikkaa linkkejä tai nappeja sekä syöttää tietoja lomakkeille taikka etsii tiettyä tekstiä sivulta. Ruby kirjastot antavat tuen kehitettävälle sovellukselle riippumatta siitä, millä teknologialla se on kehitetty. WatiRille on myös lisätty sama WebDriver, joka löytyy Seleniumista ja sen avulla WatiR tukee selaimista Chromea, Firefoxia, Internet Exploreria, operaa ja htmlunittia(selain, jossa karsittu käyttöliittymätoiminta).

On olemassa myös työkalut nimeltä WatiN ja WatiJ, jotka mahdollistavat testauksen vain eri kielillä. Nämä ovat rajapintoja .net ja java ohjelmointikielille. WatiN tukee selaimista Internet Exploreria ja Firefoxia. WatiJ tukee Internet Exploreria, Firefoxia sekä Safaria windows, mac os x sekä linux ympäristöissä.

WatiR vaikuttaa hyvin varteenotettavalta työkalulta niille, jotka tuntevat ruby ohjelmointikieltä entuudestaan ja sen kanssa toimiminen tuntuu luontevalta. Pienen testikäytön jälkeen voin hyvillä mielin suositella tätä työkalua web-sovellusten automaattiseen testaukseen. Omalla kohdalla viime kädessä se, että en tätä työkalua valinnut opinnäytetyöni tekemiseeni on se, että en entuudestaan tunne ruby ohjelmointikieltä. Kielen opetteluun olisi kulunut liikaa aikaa ja käyttöönottaminen olisi aiheuttanut jonkin verran enemmän työtä kehitysympäristössä.

7.3 jUnit

JUnit on testaamiseen tarkoitettu kehitysympäristö, jolla pystytään tekemään ja ajamaan yksikkötestejä java luokille. jUnit tarjoaa perusluokan, jota kutsutaan TestCase luokaksi ja sitä voidaan laajentaa testisarjojen tekemiseen. Assertion kirjastoa käytetään tulosten arviointiin yksittäisissä testeissä sekä useissa sovelluksissa, jotka ajavat testejä mitkä on luotu.

Työssäni tulen käyttämään kyseistä työkalua kirjoittamieni testien ajamiseen aluksi omassa testiympäristössäni, joka myöhemmin laajennetaan myös muiden tiimin jäsenien omiin testiympäristöihin. Myöhemmin on mahdollista luoda oma ympäristö, jota kaikki kehittäjät pystyvät käyttämään. Tämä työkalu valittiin siitä syystä, että se on jo käytössä sovelluksen kehityksessä ja sillä ajetaan tehtyjä luokkia vasten kirjoitettuja yksikkötestejä.

7.4 Saucelabs

Saucelabs on yritys, joka tarjoaa tuotteena mahdollisuutta tehdä käyttöliittymän testausta pilvipalvelussa. Yrityksen tuotteita ovat Sauce Scout, Sauce Connect ja Sauce on-demand. Sauce Scout on samankaltainen työkalu, kuin jo aiemmin mainittu Selenium IDE. Sillä pystyy nauhoittamaan ja ajamaan nauhoitettuja testejä. Scout eroaa IDE:stä omalla koneella selaimeen avautuvan virtuaaliympäristöllä, johon avataan haluttu selain testien suorittamista varten.

Hyvänä puolena voi pitää laajaa selain valikoimaa ja sitä, että testit nauhoitetaan videoksi. Nauhoitetut testit ovat uudelleen katseltavissa. Huonona puolena voisi pitää palvelun hitautta, joka johtuu siitä että sitä käytetään selaimen kautta. Selaimen kautta käytettäessä syntyy väkisinkin pieni viive ja myös se, että testit tehdään manuaalisesti aiheuttavat yhdessä hieman tahmaavan käyttökokemuksen. Ideana ja palveluna Scout on mielestäni hyvä. Palveluna se voisi olla vieläkin parempi, jos hitauteen tai viiveeseen saataisiin parannusta.

8 KÄYTÄNNÖN OSUUS

8.1 Työkalun valinta

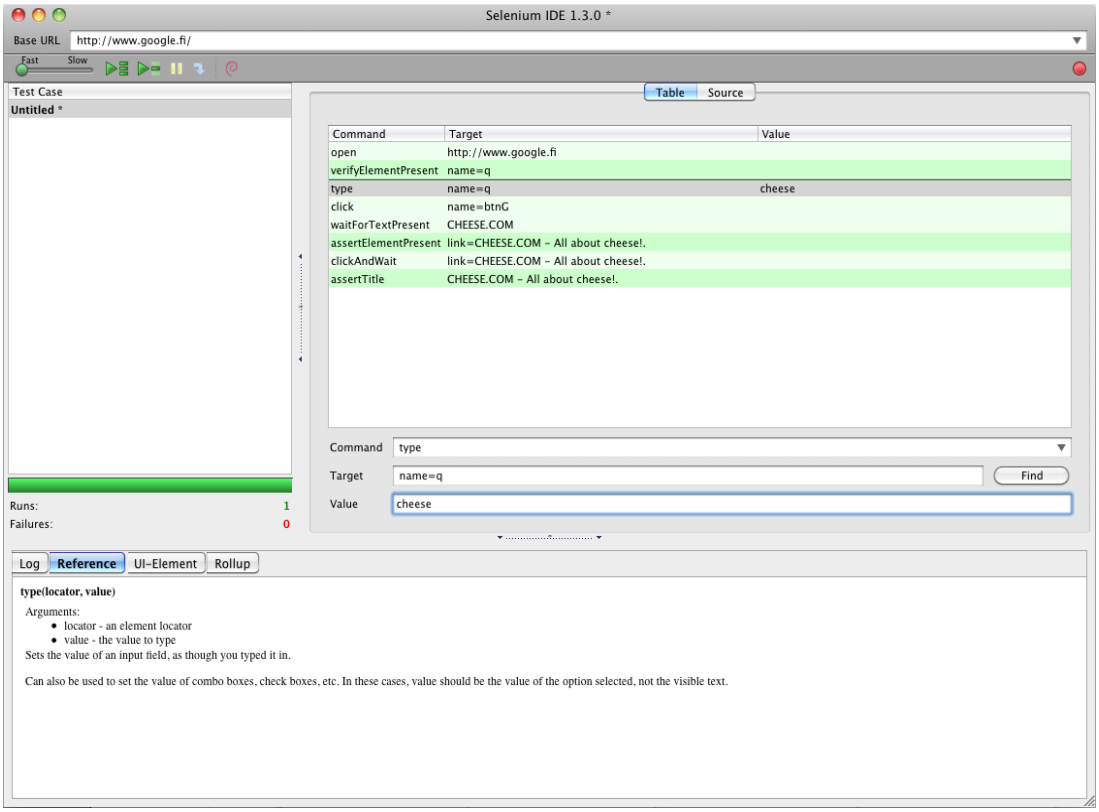
Työn käytännön osuus lähti liikkeelle projektin muodossa heinäkuun loppupuolella, kun lähdin selvittämään millaisia työkaluja tämän työn suorittamiseen olisi tarjolla. Täytyy sanoa, että viikon tutkimisen jälkeen en ollut löytänyt montaakaan varteenotettavaa vaihtoehtoa maksullisten työkalujen joukosta. Avoimen lähdekoodin ohjelmistoista niitä löytyi enemmän. Kun olin mahdolliset ohjelmat käynyt lävitse, niin valitsin kaksi niistä pienelle testikierrokselle. Työkalut olivat nimeltään Selenium ja WatiR. Molemmat valitsemani ohjelmat ovat ilmaisia käyttää ja mielestäni toiminnallisuuksiltaan, jos ei nyt huippuluokkaa, niin ainakin ylempää keskitasoa.

Seuraavassa vaiheessa piti tehdä valinta näiden kahden eri tuotteen välillä, joten lähdin tekemään vertailua. Ensimmäisenä vuorossa oli WatiR, jonka asennus ja

käyttöönotto sujuivat pienistä ongelmista kohtuullisen helposti. Ongelmia tuli vastaan ohjelman konfiguraatioiden saamisessa kuntoon, mutta niistä selvittiin pienellä selvitystyöllä eteenpäin. Pääsin kokeilemaan edes jonkinlaisten testien kirjoittamista. Tässä kohtaa tulee vastaan ensimmäinen suurempi ongelma, koska tässä käytetään ohjelmointikielenä Rubya, jota en ollut aikaisemmin opetellut, niin niiden muutamienkin testien aikaansaaminen kesti yllättävän kauan.

Jouduin käyttämään mielestäni liian paljon aikaa oikean syntaksin etsimiseen kuin itse testin vaiheiden miettimiseen ja niissä olevien ongelmien ratkaisuun, tämä tuntui turhauttavalta. Piti ottaa huomioon myös tarpeet ja vaatimukset, jotka tässä projektissa on asetettu. Vaatimuksia vasten WatiR ei loistanut, koska jo muutaman testin jälkeen jouduin toteamaan kielen ja komentojen opetteluun menevän aivan liikaa aikaa. Ei voi myöskään olettaa muiden kehittäjien tehneen ruby ohjelmointia, niin lähes sama aika menisi heiltä opetteluun. Jos komentoja joudutaan paljon opettelemaan, on se aika muusta kehitystyöstä pois ja vaikka itse olisin antamassa neuvoja, niin siihen menisi silti oma aikansa.

Toisena vuorossa oli Selenium, jonka testaustyökalu valikoimasta tutustuin ensimmäisenä IDE:en ja sillä pääsin nauhoittamaan muutamia testejä selaimessa. Edelliseen verrattuna, tuntui tämän työkalun käyttö paljon helpommalta. Helppous oli alussa sitä, kun pääsee itse suoraan selaimessa nauhoittamaan haluamansa testin ja sen jälkeen tallentaa sen haluamassaan muodossa tietokoneelle kehitysympäristöön. Seuraavalla sivulla olevassa kuvassa näytetään Selenium IDE:n käyttöliittymää.



Kuva 7. Selenium IDE.

Vaikka pääsin kohtuullisen nopeasti kiinni Selenium IDE:en ja testitapausten tekeminen oli helppoa, niin halusin silti kokeilla miten varsinainen Selenium toimii. Onko sen käyttäminen yhtä helppoa kuin IDE:n sekä miten IDE:llä tehdyt testit konvertoituisivat varsinaiselle ohjelmalle. Varsinaisen Seleniumin asennus ei ollut vaikea tehtävä. Ensimmäiseksi asetettiin riippuvuus projektin konfiguraatio tiedostoon, jonka avulla Maven lataa java asiakasohjelma kirjastot ja siihen liittyvät riippuvuudet. Kun edellinen on tehty, niin voidaan maven projekti tuoda haluttuun kehitysympäristöön. Kehitysympäristönä voi käyttää vaikka IntelliJ:tä taikka Eclipse:ä kuten tässä tapauksessa.

Toiseksi luodaan kansiot Selenium projektitiedostoille ja sen jälkeen voitaisiin periaatteessa alkaa testien tekeminen. Tässä kyseissä tapauksessa käytettiin jo valmiina olevaa testiympäristöä ja Selenium projekti lisättiin valmiiseen infraan, jolloin saatiin jonkinlainen lokaali testiympäristö käytettäväksi. Infralla tarkoitan tässä opinnäytetyössä konfiguraatioinfrastruktuuria, joka mahdollistaa testien suorittamisen. Tässä vaiheessa oli myös selvää, että testit tultaisiin siirtämään jossain

vaiheessa omaan infraansa, joka mahdollisesti käyttää omaa tietokantaa ja niihin tehtyä testidataa.

Tein vielä muutaman testin IDE työkalulla, jotka toin konvertoituna varsinaiseen Seleniumiin ja tutkin sen komentoja ja jUnit konvertoinnin mukana tulleita valmiita metodeita. Alusta lähtien komennot vaikuttivat helposti ymmärrettäviltä sekä yksinkertaisilta. Mielestäni pääsin koodiin ja syntaksiin aika nopeasti kiinni ja vastaan tulleet ongelmat eivät olleet enää itse koodiin liittyviä, vaan sovellusten toimintaan tai -logiikkaan liittyviä. Useimmiten ne olivat ajoitukseen liittyviä eli kun koitettiin simuloida mahdollisen käyttäjän toimia, niin testiohjelma suorittaa komennot nopeammin mitä ihminen manuaalisesti testattaessa. Virhetilanteet syntyivät siitä, kun selain ei pysynyt testiohjelman tahdissa ja sivuilta halutut elementit eivät ehtineet latautua valmiiksi ennen testiohjelman kutsua.

Sivulla olevien elementtien paikantamiseen on kaksi tapaa, ensimmäisenä tarjotaan xpath:ia, jota kokeilin aluksi mutta se on vaikeasti ymmärrettävää. Toiseksi vaihtoehdoksi löysin css:n, jonka käyttö on varmasti kehittäjille luontevampaa. Valtaosa sovelluskehittäjistä on sitä tehnyt aikaisemmin ja se löytää halutut elementit nopeammin kuin ensin mainittu xpath. Valinta näiden kahden välillä oli selkeä, koska oikeastaan kaikki vaatimukset ja ominaisuudet täyttyivät toisessa näistä työkaluista.

	CSS	XPATH
Kaikki P elementit	p	//p
Elementti ID:n mukaan	#foo	//*[@id='foo']
Elementit luokan mukaan	.foo	//*[contains(@class,'foo')]
Elementit attribuutin mukaan	a[title]	//a[@title]
P elementin jälkeläinen	div:first-child	//p/*/[0]
Seuraava elementti	p + *	//p/following-sibling::*[0]

Kuva 8. XPATH vs CSS.

8.2 Seleniumin käyttöönotto ja testien teko

Käytettäväksi työkaluksi valitsin siis Seleniumin. Ihan ensimmäisenä suunniteltiin tehtäväksi testisetti, joka sisältäisi korkeintaan kymmenen testitapausta. Testien tulisi kattamaan sovelluksen toimintaa ja ominaisuuksia mahdollisimman laajasti. Ajatuksena oli myös tehdä tästä testisetistä useasti suoritettava, jotta nähdään onko

uudet muutokset rikkonut jotain tärkeää toiminnallisuutta. Erikseen luotaisiin myös setti, joka ajaisi kaikki tehdyt testit lävitse. Ajankäytöllisistä syistä sen ajaminen joka kerta muutosten jälkeen olisi liian raskasta, joten sitä suoritettaisiin harvemmin.

Testisetin testitapausten valitsemisen jälkeen alkoi niiden nauhoittaminen manuaalisesti Selenium IDE:n kanssa. Näiden testitapausten tekemiseen sekä hiomiseen kului aikaa yllättävänkin paljon vaikka työkalu onkin helposti opittava. Näiden nauhoitusten aikana tuli opittua paljon lisää itse Selenium kielestä ja sen tarjoamista mahdollisuuksista.

Kun testisetti oli hiottu kuntoon, oli aika siirtää se ajettavaksi kehitysympäristössä. Testiympäristössä tultaisiin kaikki muutkin käyttöliittymätason testit suorittamaan. Aluksihan oli päädytty ratkaisuun, että ei vielä oteta kantaa mahdollisiin testidatoihin. Keskittyttiin saamaan testit toimimaan lokaalisti yhdellä koneella. Pienellä työllä saatiin tehdyt testit toimimaan käytössä olevan testitietokannan datoja vastaan.

Vaikka testit oli saatu hyvään kuntoon ja konvertoitua haluttuun muotoon, jouduttiin niitä vielä muokkaamaan. Kaikkia IDE:n toiminnallisuudet ei ollut tuettuna Selenium 2 versiossa ja osaa ohjelma ei osannut konvertoida oikein. Testien korjaamiseen meni jälleen aikaa ja erilaisten ongelmien ratkaisuja oli edessä. Seuraavissa kuvissa esimerkki konvertoidusta selenium IDE testitapauksesta sekä muokattu versio samaisesta testitapauksesta.

```
// Testiluokka
public class TestNoAccessLinks {
	private WebDriver driver; //Luodaan webdriver instanssi
	private String baseUrl=""; //Luodaan muuttuja mahdolliselle vaiko osoitteelle
	private StringBuffer verificationErrors = new StringBuffer();
	@Before
	// Suoriteaan ennen testiä
	public void setUp() throws Exception {
		driver = new FirefoxDriver(); //luodaan ja käynnistetään uusi Firefox driver
		driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS); //tehdään 30
sek implisiittinen odottaja komennoille
	}

@Test
// Testi
public void testNoAccessLinks() throws Exception
{
	// Haetaan login.html sivu
	driver.get("login.html");

	// Etsitään ja tyhjennetään sahkoposti id:llä löytyvä elementti
	driver.findElement( By.id( "sahkoposti") ).clear();
	// Kirjoitetaan sahkoposti elementtiin sahkoposti
	driver.findElement( By.id( "sahkoposti") ).sendKeys( "testiposti@..." );
	// Etsitään ja tyhjennetään salasana id:llä löytyvä elementti
	driver.findElement( By.id( "salasana") ).clear();
	// Kirjoitetaan salasana elementtiin salasana
	driver.findElement( By.id( "salasana") ).sendKeys( "**********" );
	// Etsitään elementtiä, jolla on kirjaudu luokka ja painetaan siitä kirjautuaksemme sisään
	driver.findElement( By.cssSelector( ".kirjaudu") ).click();
	// Painetaan testitili linkistä
	driver.findElement( By.linkText( "Testitili") ).click();
	// Varmistetaan, että vaihd salasana teksti on sivulla
	assertTrue( isElementPresent( By.linkText( "vaihda salasana" ) ) );
	/* Haluttiin varmistaa onko ettei sivulla ole näkyvissä kahta linkkiä, mutta näitä komentoja
ei osattu konvertoida */
	// ERROR: Caught exception [ERROR: Unsupported command [isTextPresent]]
	// ERROR: Caught exception [ERROR: Unsupported command [isTextPresent]]
	assertFalse(isElementPresent( By.cssSelector( "img[alt=muokkaa]" ) ) );
	//Kirjaudutaan ulos painamalla linkkitekstiä kirjaudu ulos
	driver.findElement( By.linkText( "kirjaudu ulos" ) ).click();
}
```

Kuva 9. Esimerkki konvertoidusta Selenium IDE testitapauksesta

```
@After
// Testin jälkeen suljetaan driver ja otetaan mahdolliset virheet kiinni
public void tearDown() throws Exception
{
        driver.quit();
        String verificationErrorString = verificationErrors.toString();
        if ( !"".equals( verificationErrorString ) )
        {
                fail( verificationErrorString );
        }
}
// Metodi, jolla tarkistetaan onko elementti löydettävissä
private boolean isElementPresent( By by )
{
        try
        {
                driver.findElement( by );
                return true;
        }
        catch ( NoSuchElementException e )
        {
                return false;
        }
}
}
```

Kuva 10. Esimerkki konvertoidusta Selenium IDE testitapauksesta jatkuu.

```
@Test
public void testNoAccessLinks() throws Exception
{
	//Varmistetaan ettei tämä tämä selain ole kirjautunut järjestelmään
	driver.get( baseUrl + "/logout.action" );
	//Haetaan login.action sivu
	driver.get( baseUrl + "/login.action" );
	//Odotetaan elementtia id:llä sahkoposti
	WaitElementWithBy( By.id(  "sahkoposti" ) );
	//Etsitään ja painetaan elementistä, jolla on id kieli
	driver.findElement( By.cssSelector( "#kieli" ) ).click();
	//Odotetaan elementtiä linkkitekstillä English
	WaitElementWithBy( By.linkText( "English" ) );
	//Etsitään ja painetaan elementistä, jolla on linkkiteksti English
	driver.findElement( By.linkText( "English" ) ).click();

	//Etsitään ja tyhjennetään sahkoposti id:llä löytyvä elementti
	driver.findElement( By.id( "sahkoposti" ) ).clear();
	//Kirjoitetaan sähköposti elementtiin sahkoposti
	driver.findElement( By.id( "sahkoposti" ) ).sendKeys( "testiposti@..." );
	//Etsitään ja tyhjennetään salasana id:llä löytyvä elementti
	driver.findElement( By.id( "salasana" ) ).clear();
	//Kirjoitetaan salasana elementtiin salasana
	driver.findElement( By.id( "salasana" ) ).sendKeys("*********");
	//Etsitään elementtia, jolla kirjaudu luokka ja painetaan sitä kirjautuaksemme sisään
	driver.findElement( By.cssSelector( ".kirjaudu" ) ).click();

	//Painetaan Testitili linkkitekstistä
	driver.findElement( By.linkText( "Testitili" ) ).click();
	//Odotetaan elementtiä id:llä asetukset
	WaitElementWithBy( By.id( "asetukset" ) );
	//Etsitään ja painetaan elementistä, jonka id on asetukset
	driver.findElement( By.id( "asetukset" ) ).click();
	//Odotetaan linkkitekstiä Vaihda salasana
	WaitElementWithBy( By.linkText( "Vaihda salasana" ) );
	//Tarkistetaan onko linkkiteksti Vaihda salasana näkyvillä
	assertTrue( isElementPresent( By.linkText( "Vaihda salasana" ) ) );

	//Tarkistetaan ettei linkkiteksti Valvonta-asetukset ole näkyvillä
	assertFalse( isElementPresent( By.linkText( "Valvonta-asetukset" ) ) )
	//Tarkistetaan ettei elementti luokalla kaavioedit ole näkyvillä;
	assertFalse( isElementPresent( By.className( "kaavioedit") ) );
	//Kirjaudutaan ulos painamalla elementtiä, jonka linkkiteksti on kirjaudu ulos
	driver.findElement( By.linkText( "Kirjaudu ulos" ) ).click();
}
```

Kuva 11. Edellä näytetty Selenium testitapaus muutettuna

Korjailujen ja ongelmien ratkaisujen jälkeen, jotka tehtiin pelkästään Mozillan Firefox selaimen versionumero viidelle. Tuli ajankohtaiseksi lisätä selaimia näitä testejä käyttämään. Tutkimisen ja pienen keskustelun jälkeen päädyin siihen, että

seuraavaksi lisättävä selain olisi Google Chrome versio 13. Chromen lisääminen sujuisi kivuttomimmin eri vaihtoehdoista, jotka olivat Internet Explorerin versio kahdeksan, Safarin versio viisi tai juuri valittu Google Chrome. Internet Explorerin lisääminen olisi aiheuttanut enemmän työtä, koska sen käyttöönotto tarvitsee windows ympäristön ja erillisen tietokoneyksikön. Myöskin se, että mahdollisiin tietokannan data-asioihin ei oltu otettu tässä vaiheessa sen enempää kantaa vaikutti päätökseen. Safarin kohdalla kävi ilmi, ettei sille ollut tehty Selenium 2 mahdollistavaa WebDriver driveria, joten tästä syystä päätin jättää Safarin vielä odottamaan lisäystä.

Myöskään Googlen Chromelle ei oltu tehty varsinaisia WebDriver drivereita, mutta Selenium 2 käyttö oli mahdollistettu suoritettavan palvelun avulla. Palvelun ohjelma pitää ladata koneelle, jonka jälkeen sitä voidaan käyttää. Firefoxin ja Chromen WebDriverin lataus metodi on hyvin samanlainen, mutta poikkeaa siten että kun Firefox lataa WebDriverinsa luokkakirjastosta, niin Chromen WebDriver käynnistetään palveluna. Samalla tavalla myös pysäytetään Chromen palvelu ja Firefoxilla WebDriverin ajo, jolloin selaimet sulkeutuvat.

Chromen lisäämisen jälkeen otettiin käyttöön jo aikaisemmin tehty apuluokka, joka mahdollisti tarvittavien metodeiden käytön molempien testattavien selaimien testien aikana. Tässä vaiheessa tuli ilmi se, että testit pystyttiin ajamaan läpi vain yhdellä selaimella kerrallaan ja että jokaisen testin jälkeen selain avautui uudelleen. Testeihin menevä aika piteni huomattavasti, jolloin lähdin toteuttamaan testejä toisella tavalla. Tarkoituksena oli saada testit suoritettua molemmilla selaimilla yhdellä ajokerralla ilman, että selainta käynnistettäisiin uudestaan joka testitapauksen alussa. Sain testien suorittamisen toteutumaan niin, että selaimet avautuivat testiluokan alussa ja sulkeutuivat kun viimeinen testiluokan testi oli suoritettu.

Edellä mainittu ominaisuus pystyttiin mahdollistamaan aluksi siten, että testit tehtiin kaikki samaan luokkaan jota ajettiin apuluokan avulla. Apuluokka pystyi käsittelemään WebDriverien lataamisen ja lopettamisen, jolloin saatiin yhden luokan testit ajettua kahdella selaimella lävitse. Tässä vaiheessa kun selaimet avattiin aina luokan suorittamisen alussa ja ne pysyivät auki aina siihen asti kunnes luokan

viimeinen testi oli suoritettu, jolloin ei enää kulunut turhaa aikaa niiden avaamiseen eri testien alussa.

Parannettavaa riitti tämän jälkeenkin ja seuraava askel oli, että testiluokat siirrettiin osaksi testsuitea/-ryhmää. Tekemällä testeistä ryhmä, mahdollistettiin useampien testiluokkien ajamisen samanaikaisesti. Luulin että tulisimme jatkossa käyttämään tätä tapaa testien tekemiseen ja ajamiseen, mutta vielä yksi tapa löytyi ja siitä paremmin hieman tuonnempana. Jatkoin testien tekemistä ja niissä ilmentyvien ongelmien ratkaisemista, jotka lähinnä liittyivät edelleen WebDriverien komentojen liian nopeaan suorittamiseen. Nopeudesta johtuen testattavat toiminnot sivuilla eivät olleet vielä valmiit. Elementtien paikallistamisessa oli myös ongelmia.

Esimerkkinä voisin mainita sellaisen tapauksen, että tarvitsee taulukosta tarkistaa jossakin tietyssä solussa oleva arvo, taulukon rivi saadaan helposti ”kiinni”. Ongelma alkaa siinä vaiheessa kun pitää vielä riviltä saada jonkin tietyn sarakkeen arvo. Tätä em. ongelmaa jouduin miettimään aika pitkään ja palaamaan siihen muutamia kertoja ennen ratkaisun löytymistä.

Ensimmäinen ratkaisuni oli sellainen, että kaikki tietyn rivin sarakkeiden arvot haetaan taulukkoon ja taulukon kaikki arvot käydään läpi. Tässä ratkaisussa pystytään koko rivin arvot tarkastamaan samalla kertaa. Kohtasin tässä ratkaisussa toisen ongelman, koska osa rivin arvoista palautui tyhjinä. Palautusarvot olivat tyhjiä, vaikka niissä saattoi olla vaikka kuvakkeita ja näiden tyhjien arvojen käsittely ei onnistunut.

```
//Metodi, jolle annetaan parametrina haluttu taulukko ja rivin selektori
protected List<String> getTableRowValues( final String tableSelector, final String rowSelector )
{
	//Etsitään haluttua taulukkoa ja tehdään siitä web element
	WebElement table = driver.findElement( By.cssSelector( tableSelector ) );

	//Sijoitetaan muuttujaan halutun rivin css selector
	String selector = String.format( "%s > td", rowSelector );

	//Haetaan taulukosta halutun rivin arvot
	List<WebElement> rowValues = table.findElements( By.cssSelector( selector ) );

	//Lisätään jokaisen sarakkeen arvo listaan
	List<String> values = new ArrayList<String>();
	for( WebElement rowValue : rowValues )
	{
		values.add( rowValue.getText() );
	}

      //Palautetaan lista rivin arvoista
      return values;
}

Testiluokassa käytetään näin:

assertEquals( "2009/05/08 22:44, 2, Alert, Huomio", getTableRowValues( ".display",
".selenium_TestiIlmoitus_3" ) );

/* Tarkistetaan onko display luokan taulukon TestiIlmoituksen numero 3 rivin sarakkeissa arvot
"2009/05/08 22:44", "2", "Alert", "Huomio" */
```

Kuva 12. Esimerkit taulukon rivin solujen hakemisesta.

```
/*
Metodi, jolla varmistetaan tietyn taulukon tietyn solun arvo. Parametreina voidaan antaa
informatiivinen viesti(ei pakollinen), odotettu arvo, haluttu taulukko, taulukon rivi ja taulukon
sarake.
*/
protected void assertTableColumn( String message, String expected, String tableSelector, int row,
int col )
{
	//Etsitään haluttua taulukkoa ja tehdään siitä web element
	WebElement table = driver.findElement( By.cssSelector( tableSelector ) );

	//Listataan kaikki taulukon rivit web elementteinä
	List<WebElement> rows = table.findElements( By.tagName( "tr" ) );

	//Haetaan haluttu rivi kaikista riveistä
	WebElement tRow = rows.get( row );

	//Listataan kaikki taulukon tietyn rivin sarakkeet web elementteinä
	List<WebElement> cols = tRow.findElements( By.tagName( "td" ) );

	//Haetaan haluttu sarake rivin sarakkeista
	WebElement tCol = cols.get( col );

	Tarkistetaan, että saatu arvo vastaa odotettua
	Assert.assertEquals( resolveMessage( message ), expected, tCol.getText() );
}

Testiluokassa käytetään näin:

assertTableColumn( "Huomio", ".datatable", 1, 3 );
/*
Tarkistetaan onko datatable luokan taulukon ensimmäisen rivin kolmannen sarakkeen arvona teksti
Huomio
*/
```

Kuva 13. Esimerkki taulukon tietyn rivin sarakkeen arvon hakemisesta.

Edellisen kuvan ratkaisu on se, johon päädyin pienen pohdinnan jälkeen ja toteutus on mielestäni ensimmäistä tapaa parempi. Pystytään hakemaan taulukosta yksittäisen solun arvo ja näin välttyä mahdollisilta tyhjien sarakkeiden aiheuttamilta virheiltä. Tässä tavassa annetaan parametreina taulukosta haluttu rivi, sarake, odotettu arvo ja saatu arvo. Odotettua arvoa verrataan saatuun arvoon ja mikäli ne eivät täsmää, niin palautetaan virheviesti. Virheviestissä kerrotaan mitä arvoa odotettiin ja mitä oikeasti saatiin. Arvojen täsmätessä palautetaan true ja jatketaan seuraavaan testikomentoon.

8.3 Johtopäätöksiä

Projektin aikana pidettiin katselmuksia, joissa yhtenä osana käytiin läpi tämän projektin edistymistä, ongelmia ja mitä olisi tehtävälistalla seuraavaksi. Yhdessä näistä keskusteltiin seuraavista asioista: Safarilla testaus, testidata, Internet Explorerin lisääminen testiselaimiin ja siihen liittyvät toimenpiteet.

Tutkin Safari selaimen WebDriverin tilannetta ja näytti siltä, että ilman Applen panosta ja tukea ei sitä luultavasti tultaisi julkaisemaan ainakaan lähitulevaisuudessa. Muutamia vapaaehtoisia tekijöitä oli ilmoittautunut tehtävään, mutta ilman applen tukea olisi WebDriverin driverin tekeminen epätodennäköistä. Ilman Safarin WebDriveria voitaisiin testaus suorittaa kuitenkin Seleniumilla käyttäen vanhempaa versiota eli Remote Controllia ja muutamia testejä oli Safarille näin jo tehtykin.

Testejä tehdessä pitää ottaa huomioon kaksi erilaista lähestymistapaa ja logiikan rakennusta, joka hidastaa ja vaikeuttaa testien tekoa. Tällä tavalla joutuisimme tekemään jotain testitapauksen osia kahteen kertaan, joissa toinen olisi WebDriverilla ja toinen rc versiolla suoritettavaa testiä varten. Toisekseen näiden kahden version syntaksi ei ole täysin samanlainen, jolloin jouduttaisiin opettelemaan molempia erikseen ja miettimään miten eri toiminnallisuudet rakennetaan testiin sen mukaan kumpaa versiota käytetään.

Päätettiin että tullaan keskittymään pelkkään WebDriver versioon Seleniumista, joka tarkoitti samalla myös sitä, ettei Safari selaimella pystytä testejä tekemään. Tämä päätös oli mielestäni oikea, koska se yksinkertaistaa testien tekemistä huomattavasti

ja kuitenkin tulevaisuudessa Seleniumin kehitys ja tuki tulee siirtymään WebDriverille Remote Controllista. On selkeämpää kun voidaan keskittyä vain yhteen tapaan tehdä ja ajaa testejä ilman että tarvitsee tuhlata aikaa miettimiseen, mitkä testin osat tarvitsee kirjoittaa kahdella eri tavalla. Myös niiden toimintalogiikka on erilainen, eli jos käytössä on Selenium 2 WebDriver, niin tehdään näin mutta muuten tehdään toisin.

Edellä mainituista syistä johtuen myös ylläpidettävyys helpottuu, kun ei tarvitse huolehtia eri syntakseista tai tuplakoodeista. Mikäli Safarille tulee tulevaisuudessa julkaistaan WebDriver driver, niin se tullaan lisäämään testejä suoritettavien selaimien joukkoon mitä pikimmiten. Toinen mahdollinen vaihtoehto on se, että Safarilla suoritettavat testit siirrettäisiin suoritettavaksi pilvipalvelussa, esim. Saucelabs. Tämä siis siinä tapauksessa, jos Safarilla testaus koetaan niin tärkeäksi, että niitä ei voi jättää huomioimatta. Esimerkiksi huomataan että jostain syystä juuri tällä selaimella sovelluksessa esiintyy paljon virhetilanteita.

Testidataan otettiin kantaa sen verran että nykyinen tapa tullaan muuttamaan, koska se ei takaa sitä, että data olisi aina sama. Nykyisellään siis käytössä ovat oman koneeni lokaalit datat, jotka mitä luultavimmin poikkeavat muiden kehittäjien datoista. Näin ei siis voi olla, koska silloin ei voida olla täysin varmoja, että johtuuko testin epäonnistuminen oikeasta virheestä vai virheellisestä datasta. Ratkaisuna voisi olla se, että tehdään käyttöliittymän testeille omat datat, jotka ajetaan testien alussa tietokantaan. Kun datat ajetaan tietokantaan testien alussa voidaan varmistua joka suorituskerralla siitä, että testattava data on sitä mitä halutaan.

Lähdettiin toteuttamaan ratkaisua testidata pulmaan. Kuten aikaisemmin mainitsin, niin nykyisellään testit olivat luotettavia vain omalla koneellani tehtynä. Tähän haluttiin muutos, koska tarkoituksena oli saada muutkin kehittäjät testejä kirjoittamaan. Tätä lähdettiin toteuttamaan siten, että siirrettiin testidatat erilleen vanhasta infrasta. Vanhassa infrassa käsiteltiin myös yksikkötestien testidatoja, joten nähtiin parhaimmaksi että nämä kaksi erotellaan. Luotiin siis uusi ympäristö käyttöliittymätesteillä ja sinne tehtiin uusi infra, jossa määriteltiin kehitysympäristön

konfiguraatioita ja käytettävään tietokantaan liittyviä asetuksia. Näiden lisäksi tehtiin muutoksia käyttöliittymän basetest -luokkaan, jota kaikki testiluokat laajentaa.

Basetest:iin lisättiin käytettävien tietokantojen asetukset ja sinne tullaan lisäämään halutut testidata, joista ajetaan dataa tietokantoihin. Kun testidata ajetaan tietokantaan testien alussa, voidaan olla varmistua sen olevan samanlaista joka testikerralla. Seuraavassa kuvassa on esimerkki yhdestä testidata tiedostosta sekä esitetään miten data ladataan tietokantaan.

```
Protected DataSource dataSource;

@Before
/*
Ennen testien ajamista suoritetaan initDb metodi, jossa konfiguroidaan tietokannat ja ajetaan
testidatat
*/
public void initDb() throws Exception
{
        database(); //kutsutaan metodia, jossa tietokannat konfiguroidaan
        DatabaseUtils.loadDataSet( "test/data/test_datatable.xml", dataSource ); /* Ladataan
test_datatable.xml tiedostosta testidataa tietokantaan */
}

//test_datatable.xml sisältö
//tietokantatauluun test_datatable sijoitetaan neljä riviä, sarakkeina id, testdate, type ja teksti
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
        <test_datatable id="1" testdate="2008-05-08 16:24:00" type="1" title="News"
teksti="Uutinen" />
        <test_datatable id="2" testdate="2011-02-02 08:20:00" type="1" title="News"
teksti="Uutinen2" />
        <test_datatable id="3" testdate="2099-05-08 22:44:15" type="2" title="Alert"
teksti="Huomio" />
        <test_datatable id="4" testdate="2011-09-03 13:12:30" type="1" title="News"
teksti="Uutinen" />
</dataset>
```

Kuva 14. Esimerkki testidata xml -tiedostosta.

Käyttöliittymätesteille tehtiin oma tomcat -instanssi, jossa testejä tullaan suorittamaan. Käytännössä se voitiin kopioida jo olemassa olevasta tomcat serveristä ja muuttamalla asetuksia se on käyttövalmis. Asetuksia muutettiin sen takia, koska aikaisemmin siirrettiin käyttöliittymätestit omaan infraansa. Näin saatiin varmistettua se, ettei varsinaisen testiympäristö ja käyttöliittymän ympäristön kanssa sekaisin.

Testejä suoritettiin testsuitena, johon oli liitetty testattaviksi halutut testiluokat. Testisuitea suoritettiin ensin omana yksittäisenä testinään aluksi. Kun testejä oli suoritettu jonkin aikaa itsenäisesti, haluttiin ne lisätä osaksi jo ajettavia yksikkötestejä. Käyttöliittymätestejä ajettiin yhdessä yksikkötestien kanssa ja huomattiin kaksi eri ongelmaa. Ensimmäinen huomattiin, että suoritettaessa käyttöliittymätestit näkyivät useampaan kertaan. Syynä tähän oli se, että testsuite jakoi aina yhden testiluokan testit jokaiselle selaimelle.

Tietyn testiluokan testien näkyminen kahteen kertaan ei varsinaisesti ollu ongelma, koska se ei haitannut testien toimintaa mutta se ei ollut se toimintapa mikä haluttiin. Toinen asia mikä havaittiin, oli testien virheellinen toiminta. Testiluokan tultua päätökseen selaimet sulkeutuivat, mutta eivät jostain auenneet enää uuden testiluokan alussa. Näiden kahden ongelmatilanteen takia haluttiin löytää jokin toinen ratkaisu testien suorittamiseen.

Ratkaisuksi löytyi jUnitin theory -annotaatio. Siinä testiluokan testitapaukset muutetaan teorioiksi. Teorioita suoritetaan datapointeilla. Yksi datapoint on yhtä kuin yksi WebDriver taikka ajettava selain. Testit suoritetaan siten, että datapointit vuorollaan suorittaa testiluokan teoriat. Näin saadaan testien tuplana näkyminen pois sekä varmistutaan siitä, että selaimet aukeaa testiluokan aluksi ja sulkeutuvat sen lopussa.

Uuden testien suorittamistavan käyttöönoton jälkeen alkoi testikomentojen yksinkertaistaminen. Selenium testien kirjoittaminen on ei ollut vaikeaa taikka kielen opetteluun ei mennyt kauaa aikaa, mutta tätäkin haluttiin vielä helpottaa. Kirjoitin joukon apumetodeita, joiden avulla testien kirjoittaminen olisi entistä helpompaa. Apumetodit sisälsivät Selenium komentoja, joita käytettäisiin parametrien avulla, jolloin testintekijän ei tarvitse tietää Selenium komentoja ja miten sitä käytetään, vaan hän voi valita apumetodin ja antaa sille haluamansa parametrit. Tällä tavalla pystyttiin helpottamaan testien tekemistä sekä oppimista nopeutettua.

Tämä tarkoitti myös sitä, että kaikki tähän mennessä tehdyt testit tuli refaktoroida käyttämään näitä uusia metodeita. Refaktoroinnin aikana ei tullut mainittavampia ongelmia vastaan. Joitakin apumetodeita piti korjata, koska ne eivät toimineetkaan niin kuin olin ajatellut. Lisäksi apumetodeita tuli lisää, kun uusia tarpeita niille ilmeni. Refaktorointiin kuului myös apumetodeiden kommentointi. Kommentoinnilla pyritään selkeyttämään metodia ja sen toimintaa nykyisille ja mahdollisille uusille kehitystiimin jäsenille.

8.4 Projektin jatko

Projekti oli tullut tämän opinnäytetyön kohdalla päätöspisteeseen, vaikkakin projekti itsessään jatkuu loppuvuoteen asti. Projektin alkuperäinen päättymisaika määriteltiin vuoden loppuun. Mielestäni tässä kohtaa on hyvä ajankohta projektin kannalta ottaa pieni tarkastelu siitä mitä on tehty ja mitä tulevaisuudessa halutaan tehdä. Nyt pystytään luomaan käyttöliittymätestejä koko kehitystiimin voimin helposti. Tehtyjä testejä voidaan ajaa kahdella selaimella lävitse ja etsimään näin virheitä muualtakin kuin service rajapinnalta.

Testien kirjoittamiseen avuksi tehdyt metodit helpottavat ja nopeuttavat testien tekoa siinä määrin, että niitä pystyisi tekemään jo hyvin pienellä opettelulla. Niin sanottuja erikoistapauksia lukuunottamatta toimintamalli ei vaadi tutustumista varsinaiseen Seleniumiin, toki sen ymmärtämisestä ei haittaakaan ole. Myöskin, jos kehitystiimiin liittyisi jossain vaiheessa uusia jäseniä, niin heidän kohdallaankaan ei tarvita suurta määrää koulutusta, jotta pääsee mukaan näiden testien tekemisessä ja suorittamisessa.

Jatkossa tullaan ainakin Internet Explorer lisäämään selain testattavien selaimien listaan. Silloin saadaan vieläkin kattavammin sovelluksen toimintaa testattua. Toinen asia mitä mielestäni olisi hyvä ainakin miettiä ja mielellään myös toteuttaa, on testien ajaminen rinnakkaisesti. Testejä ajettaessa rinnakkain suoritetaan aina yksi testi samanaikaisesti kaikilla selaimilla, mikä lyhentäisi testeihin menevää aikaa huomattavasti. Säästetyn ajan huomaisi jo alkuvaiheissa, mutta erityisesti silloin kun testejä alkaa olemaan useita kymmeniä tai satoja ja halutaan suorittaa kaikki testit

esimerkiksi ennen uuden version julkaisua. Silloin on erittäin tärkeää ettei testien ajamiseen mene kohtuuttoman paljon aikaa.

Kolmantena asiana haluaisin tuoda esille asian, jota on hyvä miettiä ja käydä läpi jokaisessa projektissa, jossa aiotaan testata sovelluksia selaimien avulla. On tärkeää päättää se, että millä selaimilla halutaan testata. Tärkeää on myös miettiä se, että mitkä selain versiot halutaan testata. Nykyään selainten versiot päivittyvät huomattavasti nopeampaa tahtia kuin ennen ja uusia versioita julkaistaan jopa kuuden viikon välein.

Nopea päivitystahti aiheuttaa päänvaivaa testaajille, koska vanhalle versiolle tehty testi ei välttämättä toimi uudessa versiossa taikka päinvastoin uuteen tehty testi ei toimi vanhemmalla versiolla. Tähän samaan asiaan liittyy myös se, että miten vanhoille selainten versioille halutaan antaa tuki. Mitä vanhemmalle versiolle halutaan sovelluksella antaa tuki, niin sitä hankalampaa tulee testien ylläpidosta, koska pitää ottaa eri versioiden erilaisuus huomioon. Tämän asian ratkaisuun voisi tietysti käyttää jo aikaisemmin mainitsemaani Saucelabs:n tarjoamaa palvelua. Testit voitaisiin ajaa pilvipalvelussa myös selainten eri versioilla ilman, että joudutaan uhraamaan omia resursseja.

9 LOPPUSANAT

Opinnäytetyön projekti oli mielestäni onnistunut, vaikka aivan kaikkia osia ei saatukaan valmiiksi. Projekti siis jatkuu kuluvan vuoden loppuun ja silloin sen on määrä valmistua kokonaisuudessaan. Opinnäytetyön tuloksena syntyi hyvä toimintamalli käyttöliittymän toiminnan testaukseen, joka on nopeasti opittava sekä helppo ottaa käyttöön. Työn tulos on otettu käyttöön osa Enersize Oy:n sovelluskehitystä.

Projektia toteutettiin ketterien kehitysmenetelmien avulla, joka mahdollisti nopean reagoinnin muutostarpeisiin. Toteutusta muutettiin projektin aikana useampaan kertaan, mikä olisi saattanut aiheuttaa ongelmia perinteisemmällä

kehitysmenetelmällä. Myöskin projektista tuotetun dokumentaation määrä on hyvin pieni. Tämän opinnäytetyön lisäksi ainoat dokumentaatiot ovat asennus- ja käyttöönottodokumentti sekä käyttöliittymän testaamiseen liittyvät vinkit sekä parhaat toimintatavat. Nämäkin dokumentit ovat osa toimeksiantajan sisäistä wiki -palvelua, josta ne ovat helposti yrityksen henkilöstön saatavilla.

Työstä saatu palaute on ollut positiivista sekä rakentavaa, josta on hyvä lähteä projektia viemään eteenpäin. Seuraavassa suora lainaus yhdestä kehitystiimin jäseneltä tulleesta palautteesta: ”Hyvin tehdyn taustatutkimuksen ja hyvien apumetodien myötä UI testauksen aloittaminen on erittäin helppoa ja palkitsevaa. Itseasiassa hyvin nopeasti (jopa muutamien testien tekemisen jälkeen) tekee testausta mielummin UI testiluokkien kautta kuin itse napsauttelemalla. Ja luonnollisesti siitä on huikea etu sovelluksen luotettavuuden ja laadun kannalta jatkossa.”

Lopuksi haluaisin kiittää Enersizea mahdollisuudesta tämän työn toteuttamiseen. Lisäksi haluan kiittää muita kehitystiimin jäseniä heidän tähän asti antamastaan tuesta sekä näkökulmista tämän opinnäytetyön suhteen.

LÄHTEET

Ahonen, E: Ohjelmistotestaus. Hämeen ammattikorkeakoulu. Opinnäytetyö. http://data.hamk.fi/~lseppane/courses/ohjelmistotestaus/Ohjelmistotestaus_Elina_Ahonen.pdf

Haikala, I & Märijärvi, J. 2006. Ohjelmistotuotanto. Jyväskylä. Gummerus.

Kautto, T: Ohjelmistotestaus ja siinä käytettävät työkalut. Seminaariesitelmä. Viitattu 16.10.2011. http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus/

Patton, R 2005. Software Testing. Sams publishing.

Pohjolainen, Pentti. 2003. Ohjelmiston testauksen automatisointi. Kuopion yliopisto. Pro gradu -tutkielma.

http://www.cs.uku.fi/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf

Mäki, Erkki. 2002. USA:n kauppaministeriö: Ohjelmistovirheet maksavat miljardeja. Viitattu 18.10.2011.
http://www.digitoday.fi/data/2002/07/01/ohjelmistovirheet-maksavat-miljardeja/20026986/66

Peippo, Raili. 2009. Prototyyppilaitteiden automatisoitu rautapohjaisen suorituskyvyn testaus. Jyväskylän ammattikorkeakoulu. Opinnäytetyö.

Black, R. 2007. Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. Indianapolis, Indiana: Wiley Publishing, Inc.

Li, K. & Wu, M. 2004. Effective Software Test Automation: Developing an Automated Software Testing Tool. Alameda: SYBEX Inc.

Resig, John. 2005. Xpath and CSS selectors. Viitattu 25.10.2011.
http://ejohn.org/blog/xpath-css-selectors/