LAUREA AMMATTIKORKEAKOULU Uuden edellä

Testiautomaatiotyökalun käytettävyyden arviointi

Helkamäki, Mirka & Ikola, Tiina

2011 Laurea Kerava

Laurea-ammattikorkeakoulu
Kerava

Testiautomaatiotyökalun käytettävyyden arviointi

Mirka Helkamäki, Tiina Ikola
Tietojenkäsittelyn koulutusohjelma
Opinnäytetyö
Toukokuu, 2011

Laurea-ammattikorkeakoulu **Tiivistelmä**
Kerava
Tietojenkäsittelyn koulutusohjelma

Mirka Helkamäki, Tiina Ikola

Testiautomaatiotyökalun käytettävyyden arviointi

Vuosi 2011 Sivumäärä 64

Tämän opinnäytetyön tavoitteena oli tutkia testiautomaatiotyökalua nimeltä Robot Framework ja löytää siitä mahdollisia käytettävyysongelmia. Tavoitteena oli lisäksi tutustua testiautomaatioon osana ohjelmistotestausta sekä löytää sen hyödyt ja haittapuolet ohjelmistotestauksessa. Käytettävyysongelmien etsimiseen käytettiin apuna heuristista arviointia. Työ jakautuu teoriaosuuteen ja käytännön osuuteen.

Työn teoriaosuudessa käsitellään ensin yleisesti käytettävyyttä, heuristista arviointia ja käytettävyystestejä. Teoriaosuuden toisessa osassa keskitytään ohjelmistotestaukseen sekä testiautomaatioon. Tässä käsitellään myös eroavaisuudet automaatiotestauksen ja manuaalitestauksen välillä. Eri testiautomaatiotyökalut ja kuinka valita juuri sopiva omaan tarkoitukseen on käsitelty tässä osuudessa.

Opinnäytetyön käytännön osuudessa keskityttiin tuomaan esille mahdolliset käytettävyysongelmat Robot Framework –automaatiotestaustyökalusta. Tässä osuudessa on tarkasteltu Nielsenin heuristista listaa apuna käyttäen Robot Frameworkin ja sen käyttöliittymän (RIDE) toiminnallisuuksia. RIDE:n toiminnallisuuksia on havainnollistettu kuvin, joista näkyvät perusominaisuudet.

Tutkimuksen edetessä huomattiin, että Robot Framework on suhteellisen käyttäjäystävällinen eikä käytettävyysongelmia löytynyt kuin muutama. Ensimmäinen ongelma oli muuttujien värit itse testitapauksissa. Muuttujien värinä oli vihreä, vaikka testitapaus ei välttämättä vielä edes tunnistanut kyseistä muuttujaa. Tämä saattaa aiheuttaa ongelmatilanteita käytettävyydessä: vihreä yleensä tarkoittaa oikeanlaista toimintaa.

Toinen ongelma keskittyi RIDE-ohjelman omaan testitapausten ajoon. Jos testitapausajon lopettaa kesken, avaa ohjelma kuitenkin määrittelemättömän määrän uusia selainikkunoita. Tämä koettiin erittäin harmittavaksi ongelmaksi ja sen huomattiin haittaavan käytettävyyttä.

Näiden ongelmien lisäksi ohjelmasta ei löytynyt heuristisen arvioinnin mukaan muita ongelmia, ja ohjelma todettiin käytettävyydeltään hyväksi. Näin ollen todettiin, että Robot Framework toimii hyvin selaintestauksen apuvälineenä ja on varteenotettava työkalu, kun halutaan laadultaan ja käytettävyydeltään hyvää apuvälinettä automaatiotestaukseen.

Asiasanat: käytettävyys, automaatio, testaus, testauslaitteet

Laurea University of Applied Sciences
Laurea Kerava
Business Information Technology

Abstract

Mirka Helkamäki, Tiina Ikola

Usability Evaluation of Test Automation Tool

Year 2011 Pages 64

The goal of this Bachelor's thesis was to examine a test automation tool named Robot Framework and find possible usability problems. The goal was to get to know test automation as a part of software testing and find its benefits and disadvantages in software testing. Heuristic evaluation was used as help in finding usability problems. The thesis consists of a theory section and a practical section.

The first part of the thesis´s theory section deals first with usability in general, heuristic evaluation and usability testing. The second part focuses on software testing and test automation. This part discusses also differences between test automation and manual testing. Further, different test automation tools are dealt with as well as how to choose a suitable one meet one´s own needs.

The thesis´s practical section concentrates on the possible usability problems in Robort Framework test automation tool. In this part Nielsen´s heuristic list was used as a help to test different functionalities in Robot Framework and its user interface (RIDE). RIDE´s functionalities have been demonstrated with pictures in which the main functionalities can be noticed.

As the research went forward it was noticed that Robot Framework is relatively user friendly and only a few usability problems were detected. The first problem was the colors of variables in test cases. The variables were green although the test case did not necessarily recognise the variable in question. This may cause problems in usability: green usually means a right kind of action.

The second problem emerged during running the test cases in RIDE. If you stop running the test case in the middle of the run the program will open several new browser windows. This was considered an annoying problem and it was noticed to disturb usability.

Apart from these two problems no other problems could be found via heuristic evaluation and therefore the program was discovered user friendly. Therefore it was noticed that Robot Framework works well in browser testing and is a considerable tool in test automation when you want a tool of good quality and usability.

Keywords: usability, automation, testing, test tools

Sisällys

1 Johdanto

Testauksen automatisointi on tärkeää jokaisen käyttöliittymän, Internet-sivun tai sovelluksen testauksessa. Sen avulla testausta saadaan nopeutettua ja esimerkiksi yölliset sivujen päivitykset eivät vaadi testausresursseja, jos vain automaatiotestit testaavat sovelluksen.

Pelkät sivustojen eri sivuja testaavat testit ovat puuduttavia testaajille. Testaajien työtä voidaan helpottaa, jos edellä mainitut testit hoidetaan testiautomaation kautta. Tällöin testaajat voivat keskittyä testaamaan muita ominaisuuksia, kuten sovelluksen sisältöä riippuen testikontekstista. Testiautomaatiossa on etuna se, että samaa asiaa voidaan testata uudestaan ja uudestaan. Testitapaukset tallennetaan koneelle ja niitä voidaan ajaa yhden napin painalluksella.

Opinnäytetyössä käydään läpi sekä heuristisen arvioinnin, käytettävyystutkimuksen, testauksen että testiautomaation teoriaa. Työssä selvitetään eroja automaatiotestauksen ja manuaalitestauksen välillä. Tarkoituksena on löytää vastauksia, mihin testiautomaatiota kannattaa käyttää ja kuinka sen suunnittelu tulisi hoitaa.

Heuristisen arvioinnin osalta pyritään selvittämään, miten arviointi tehdään, minkälaisia resursseja se vaatii ja minkälaisen pätevyyden se arvioijalta vaatii. Käytettävyystestaus ja –testit ovat osana käytettävyyteen perehtyvää teoriaa.

Tutkimuksen tarkoituksena on tehdä heuristinen arviointi testiautomaatiotyökalusta Robot Framework. Heuristisen arvioinnin avulla etsitään ongelmakohtia työkalusta ja tutkitaan sen toimivuutta. Tarkoituksena on myös miettiä parannusehdotuksia mahdollisiin ongelmakohtiin.

2 Käytettävyys

Käytettävyydellä kuvataan sitä, kuinka hyvin ja sujuvasti käyttäjä pääsee haluamaansa päämäärään käyttäen sovelluksen eri toimintoja. Käytettävyys lyhykäisyydessään on käyttäjän eli ihmisen ja tietokoneen vuorovaikutusta. Käytettävyydellä tutkitaan ominaisuuksia, jotka tekevät tietystä tuotteesta joko hyvä tai huonon. Erään mielipiteen mukaan käytettävyys on tutkimustyön työkalu, jonka juuret ovat klassisessa tutkimuksellisessa menetelmäopissa. (Wiio 2004, 13–14; Rubin & Chisnell 2008, 21.)

Käytettävyys koostuu Kuutin (2003, 13) mukaan viidestä eri osa-alueesta, jotka ovat muistettavuus, opittavuus, tehokkuus, miellyttävyys sekä pieni virhealttius. Käytettävyyden tavoitteina Wiio (2004, 30) näkee käytön tehokkuuden sekä opittavuuden helppouden. Hänen mukaan hyvä ohjelma vaatii käytettävyydeltään neljä eri asiaa. Sen tulee olla ymmärrettävä,

vaivaton, kattava sekä esteettisesti miellyttävä. Ymmärrettävyys kattaa sen, että käyttäjän on helppo päätellä miten ja millä toiminnoilla hän pääsee haluamaansa päämäärään. Samoin ymmärrettävästä ohjelmasta on helppo päätellä mitä sillä ylipäänsä tehdään. Vaivaton sovellus taas antaa käyttäjän suoriutua mahdollisimman helposti tehtävistään, koska vaivalloinen toimenpide on samalla myös aikaa vievä. Vaivatonta on suoriutua tehtävistä mahdollisimman yksinkertaisella tavalla. Kattava sovellus taas on Wiion (2004, 31) mukaan sellainen joka tarjoaa kaikki mahdolliset ja mahdottomat toiminnot ja tiedot, joita käyttäjä tarvitsee. Jotta sovelluksesta saisi esteettisesti miellyttävän, suunnitellaan se niin, että se viestittää käyttäjälle laatua ja osaamista.

Tärkeän käytettävyydestä tekee se, että sen avulla saadaan tuotteesta luotettava käyttää. Luotettavat tuotteet saavatkin suurimmat käyttäjäryhmät puolelleen. Kukapa tahtoisi käyttää epävakaata ohjelmaa joka voi koska tahansa saada aikaan virhetilanteen. Hyvä käytettävyys on myös kustannustekijä niin käyttäjälle kuin tuotteen valmistajallekin. Käyttäjälle se lisää tuottavuutta ja se vähentää käyttäjätuen tarvetta. Tuotteen valmistajalle se taas muun muassa nostaa tuotteen markkina-arvoa. Kaiken kaikkiaan hyvä käytettävyys on osa tuotteen laatua. (Ovaska 2004; Eulenberger-Karvetti 2005.)

3 Heuristinen arviointi

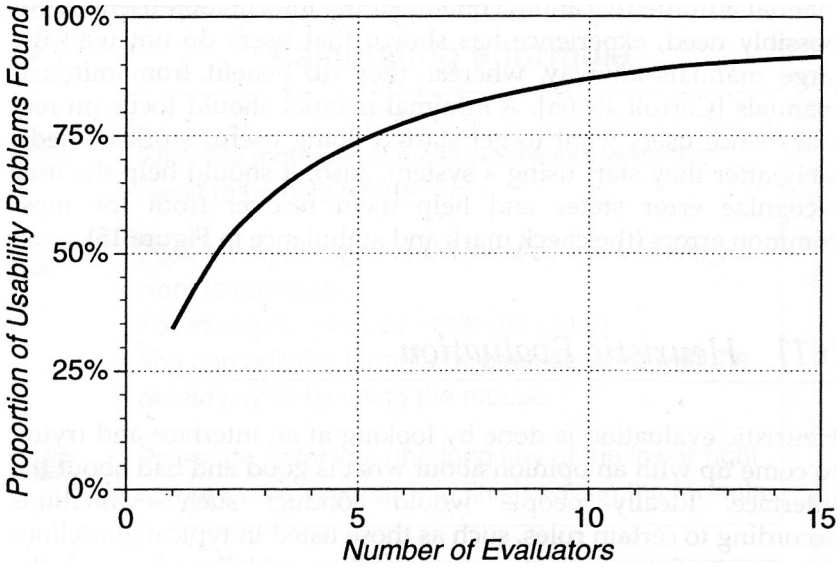
Heuristisella arvioinnilla tarkoitetaan kokemukseen perustuvaa arviointia. Se on käytettävyyden arviointia ilman käyttäjää ja sen tarkoitus on löytää tuotteen käytettävyysongelmat. Apuna tässä käytetään usein jonkinlaista heuristista muistilistaa kuten Nielsenin lista. (Nummiaho 2003; Käyttötuotteen heuristinen arviointi.)

Heuristinen arviointi perustuu erilaisiin heuristiikkoihin, jotka ovat erilaisia listoja ohjeista ja säännöistä. Ehkä yleisimmin käytetty heuristinen arviointimenetelmä on Nielsenin lista, joka sisältää kymmenen eri kohtaa johon vastataan. Toinen tunnettu on Schneidermanin lista, jossa taas on kahdeksan eri kohtaa. Myös pidempiä listoja löytyy, mutta arviointi on helpompaa tehdä lyhyellä listalla kuin pitkällä. (Kuutti 2003, 47.)

Heuristisen arvioinnin tavoitteena on löytää esimerkiksi käyttöliittymään liittyvät käytettävyyden ongelmat. Usein heuristisia arviointeja tehdäänkin koko suunnittelu- ja kehitysprosessin aikana, jotta siinä löydettävät ongelmat voidaan korjata ennen kuin tuote pääsee tuotantoon. (Nielsen 1993, 155.)

Heuristisessa arvioinnissa yksi arvioija voi löytää ainoastaan 35 % kaikista sovelluksen käytettävyysongelmista. Mitä enemmän arvioijia osallistuu arviointiin, sitä suurempi prosentti saadaan. On suositeltavaa, että arvioinnissa käytettäisiin vähintään kolmea arvioijaa.

Optimaalisimmassa arviointitilanteessa on viisi arvioijaa, koska viiden ylittävä määrä ei enää nosta ongelmien löytymisprosenttia. (Nielsen 1993, 156; Kuutti 2003, 48.)



Kuvio 1: Heuristisen arvioinnin avulla löydetyt ongelmat suhteessa arvioijien määrään (Nielsen 1993, 156).

3.1 Heuristisen arvion tekeminen

Heuristinen arviointi toteutetaan itsenäisesti, jotta ei kenenkään arvio perustuisi kenenkään toisen arviointiin. Vasta siinä vaiheessa kun kaikki arvioijat ovat tehneet arviointinsa loppuun, saavat arvioijat keskustella keskenään itse arvioinnista sekä siinä löytyneistä ongelmista. Tästä loppukeskustelusta laaditaan yhteenveto löydöksistä sekä lista löydetyistä ongelmista ja käytettävyyspuutteista. (Nielsen 1993, 157; Kuutti 2003, 48–49.)

Listassa löydetyistä ongelmista on suora viittaus sääntöön, jota se heuristiikassa rikkoo. Vaikka heuristisen arvioinnin tarkoitus on löytää ongelmia, ei se silti ota kantaa niiden korjaamiseen. Vikojen korjaus ja käytettävyyden parantaminen jää täysin ohjelman tekijöiden vastuulle. (Kuutti 2003, 49.)

Heuristisen arvioinnin pystyy tekemään kuka vain omaamatta aikaisempaa kokemusta heuristiikasta tai arvioinnista. Tosin tilanteessa, jossa käytetään kokematonta arvioijaa, voidaan löytää vain 22 % havaittavista käytettävyyden ongelmista. (Kuutti 2003, 48–49.)

Tyypillisesti heuristinen arviointi kestää noin tunnista kahteen tuntiin. Tosin, jos arvioitavaa on paljon, voi myös arviointiaika kasvaa. Tällaisissa tilanteissa on kuitenkin suositeltavaa, että arviointi pilkotaan pienempiin osiin. (Nielsen 1993, 158.)

Yleisenä periaatteena on, että jokainen arvioija päättää itse kuinka haluaa edetä oman arviointinsa kanssa. On kuitenkin suositeltu, että käyttöliittymä ja sen rajapinta tulisi käydä vähintään kaksi kertaa läpi. Ensimmäisellä läpikäynnillä tulisi saada tuntuma testattavaan tuotteeseen sekä jonkinlainen kuva sen käytöstä ja perustoiminnoista. Toinen kierros taas mahdollistaa keskittymisen tuotteen käytettävyyteen ja sen tiettyihin elementteihin. Näin ollen nämä elementit voidaan nähdä osana isompaa kokonaisuutta. (Nielsen 1993, 158–159.)

3.2 Nielsenin lista

Tunnetussa heuristisessa arvioinnissa, Nielsenin listassa on kymmenen eri kohtaa, joilla arvioidaan tuotetta. Tästä listasta on olemassa paljon myös erilaisia ja eripituisia versioita. (Kuutti 2003, 49) Tämä osio käsittelee yhtä versiota Nielsenin listasta kohta kohdalta.

1. Mahdollisimman yksinkertainen ja luonnollinen dialogi

Käyttöliittymän tulisi olla mahdollisimman yksinkertainen käyttää, jotta käyttäjä oppisi helposti miten ohjelmaa käytettään. Jos käyttöliittymä ei ole tarpeeksi yksinkertainen käyttää, vaarana on, että käyttäjä ymmärtää väärin saamansa informaation. Ihanteellisen yksinkertaisessa käyttöliittymässä esitetään käyttäjälle vain tarvittava informaatio juuri silloin ja siinä paikassa kun sitä tarvitaan. (Nielsen 1993, 115–116.)

2. Puhu käyttäjän kanssa samaa kieltä

Käyttöliittymää suunnitellessa tulisi ottaa huomioon siinä käytettävä terminologia. Osana hyvää käyttäjäkeskeistä suunnittelua huomioidaan, että käytettävä kieli pohjautuu käyttäjien yleiseen kieleen eikä esimerkiksi systeemi- tai tietokonepohjaiseen termistöön. Termien tulisi olla käyttäjille jo tuttuja tai ainakin helposti pääteltävissä. Vuoropuhelu tulisi myös käydä mahdollisimman pitkälle käyttäjän omasta näkökulmasta. (Nielsen 1993, 123; Kuutti 2003, 52; Nummiaho 2003.)

3. Minimoi käyttäjän muistikuormitusta

Tietokoneet ovat erittäin hyviä muistamaan asioita tarkalleen. Siksi onkin käyttäjän näkökulmasta helpottavaa jos tietokoneet on laitettu kantamaan muistamisen taakka käyttäjän harteilta. Tämän takia käyttäjälle tulisikin antaa vaihtoehtoja joista valita kuten esimerkiksi oletusarvoisia vastauksia. (Nielsen 1993, 129; Nummiaho 2003.)

4. Johdonmukaisuus

Yksi käytettävyyden perusperiaatteista on johdonmukaisuus. Koko käyttöliittymän tulisi toimia samalla periaatteella joka kohdassa. Kun käyttäjä tietää saman teon tai komennon johtavan aina samaan lopputulokseen, käyttäjä tuntee olonsa varmemmaksi ja luottavaisemmaksi käyttäessään ohjelmaa. Johdonmukaisuutta on se, että sama informaatio löytyy aina samasta paikasta. (Nielsen 1993, 132)

5. Palautteen anto

Käyttäjää tulisi koko ajan informoida siitä mitä tehdään ja ollaan mahdollisesti tekemässä. Palautteen tulisi olla jatkuvaa, eikä sen saisi antaa odottaa virhetilannetta ilmestyäkseen. Normaalisti palautteen tulee siis ilmestyä nopeasti, mutta se ei saa olla niin nopeaa, että se menee ohi käyttäjältä. Kun prosessi kestää pitkään (yli kymmenen sekuntia), tulee käyttäjälle antaa jonkinasteinen arvio siitä, kuinka kauan prosessi kestää. Tilannepalkki on hyvä tapa havainnollistaa käyttäjää tällaisissa tilanteissa tilanteen kokonaiskestosta ja siitä kuinka kauan prosessi vielä mahdollisesti kestää. Palautteen tulee olla aina positiivista. (Nielsen 1993, 134; Kuutti 2003, 57–58.)

6. Selkeät poistumistiet

Käyttäjä ei pidä tunteesta, että on esimerkiksi tietokoneen vanki. Koska käyttäjän tulisi tuntea itsensä tilanteen herraksi jokaisessa tilanteessa, voidaan tätä helpottaa ja kasvattaa merkitsemällä selkeät peruutus- ja ulospääsytoiminnot. Nämä tulisi löytyä mahdollisimman monesta tilanteesta. Samoin, jos käyttäjällä on tarve päästä poistumaan ohjelmasta, voi hänelle myös syntyä tarve peruuttaa jokin tietty toiminto. Esimerkiksi jos kone ei pysty lopettamaan prosessiaan nopeasti (n. kymmenen sekuntia), niin käyttäjälle täytyy antaa mahdollisuus peruuttaa kyseinen toiminto. (Nielsen 1993, 138–139.)

7. Oikotiet

Käyttöliittymän käyttö tulee olla helppoa niin ensikäyttäjälle kuin kokeneemmallekin käyttäjälle. Uudelle käyttäjälle käyttö tehdään helpoksi esimerkiksi valikoin, joista voi valita haluamansa toiminnon selkeästi. Kokeneempi käyttäjä taas haluaa usein tehdä samoja toimintoja nopeammin aikaa säästäen. Tämä voidaan mahdollistaa erilaisin oikotein.

Suosituimmat ja yleisimmät oikotiet ovat erilaiset näppäinyhdistelmät. Muina oikoteinä voi olla esimerkiksi erilaiset pikanäppäimet tai vaikka hiiren tuplaklikkaus. Oikopolkujen tulee olla myös mahdollisimman yksinkertaisia ja helposti opittavia. (Nielsen 1993, 139)

8. Kunnolliset virheilmoitukset

Virheilmoitusten tulee olla kohteliaita sekä neutraaleja. Ne tulee kirjoittaa selkokielellä ja niiden sisältö tulee ymmärtää sellaisenaan. Virheilmoituksen tulee antaa heti selkeä kuva tapahtuneesta virheestä sekä siitä, miten sen voisi korjata. (Kuutti 2003, 62.)

9. Virheilmoitusten tulisi noudattaa neljää sääntöä:

a. Virheilmoitukset tulisi aina esittää selkeällä ja yksinkertaisella kielellä. Niissä tulisi välttää pitkiä koodeja ja numeropätkiä. Jos välttämättä halutaan näyttää virhekoodi, tulisi se esittää aina viimeisenä. Käyttäjän tulee myös ymmärtää virheilmoitus ilman koodiluetteloja tai ohjekirjoja.

b. Virheilmoitusten tulee olla ennemmin hyvin täsmällisiä kuin hyvin yleisellä tasolla esitettyjä. ”Ei voida avata tiedostoa” vs. ”Ei voida avata ´Kappale 5´, koska kyseistä tiedostoa ei löydy”.

c. Virheilmoitusten täytyy myös auttaa rakentavasti käyttäjää selvittämään kyseistä ongelmaa. Ympäripyöreistä virheilmoituksista ei ole käytännön hyötyä käyttäjälle, jos ne eivät kerro miten ongelmaa voisi mahdollisesti ratkaista.

d. Virheilmoitusten kuten palautteiden tulee olla kohteliaita. Virheilmoitukset eivät saa nolata käyttäjää eivätkä syyttää käyttäjää tapahtuneesta virheestä. (Nielsen 1993, 142–143)

10. Virheiden ennaltaehkäisy

Parempaa kuin saada hyvä virheilmoitus on tietenkin olla saamatta niitä ollenkaan. Aina kun käyttäjää pyydetään kirjoittamaan jotain, on riski kirjoitusvirheestä olemassa. Pienellä suunnittelulla ja muokkauksella voidaan välttää iso joukko virheitä. On esimerkiksi riskialttiimpaa pyytää käyttäjää kirjoittamaan avattavan kansion nimi, kuin laittaa hänet valitsemaan se valmiista listasta. Varsinkin vakavia virheitä voidaan ehkäistä hyvin varmistamalla käyttäjältä tämän haluama toiminto, ennen sen toteuttamista. ”Haluatko varmasti sulkea ohjelman tallentamatta?”. (Nielsen 1993, 145–146.)

11. Apu ja dokumentointi

Vaikka lähtökohtana on tehdä ohjelma, jonka käytössä ei tarvita apua eikä ohjekirjoja, on tällaisen ohjelman tekeminen jos ei mahdotonta niin ainakin hyvin haastavaa. Yleisesti tiedetään, etteivät käyttäjät lue ohjekirjoja. Jos epäilet tätä väitettä, niin Nielsenillä (1993,

149) on oiva keino tämän testaamiseen: Vieraile muutaman käyttäjän luona ja aseta kymmenen dollarin seteleitä jonnekin päin ohjekirjan väliin. Kun seuraavan kerran vierailet kyseisen käyttäjän luona, tarkista montako piilottamaasi seteliä on edelleen ohjekirjan välissä.

Seurausta tästä ilmiöstä on, että käyttäjät tarttuvat ohjekirjaan vasta siinä tilanteessa kun jokin on jo mennyt pieleen ja he yrittävät jonkin asteisessa paniikissa etsiä ratkaisua ongelmaansa. Sen takia hyvästä ohjelmasta onkin jo olemassa kaksi ohjeistusta. Toinen on lyhyt yleisesittely aloitteleville käyttäjille ja toinen on kattavampi hakuteos edistyneille käyttäjille, jotka tietävät mitä hakea ja mistä. On myös hyvä olla olemassa jonkinlainen avustustoiminto, joka esimerkiksi hiiren klikkauksella kertoo, mitä toiminto tekee tai mitä arvoja kenttä vaatii. (Nielsen 1993, 148–149; Kuutti 2003, 65–66.)

3.3 Heuristisen arvioinnin vakavuusluokitus

Osana heuristista arvioita on myös löydettyjen ongelmien vakavuuden luokittelu. Ongelmia luokitellessa tulee ottaa huomioon seuraavat asiat: esiintymistiheys, ongelman vaikutukset käyttäjälle, ongelman toistuvuus sekä ongelman markkinavaikutukset. Kun nämä edellä mainitut asiat on otettu arvioinnissa huomioon, annetaan ongelmalle vakavuusluokitus numeroin 0-5.

0 tarkoittaa asteikolla pienintä mahdollista ongelmaa.

1 tarkoittaa pientä ongelmaa jota voidaan pitää lähinnä kosmeettisena. 1-tason ongelman korjaamisella ei ole kiire.

2 tarkoittaa pientä käytettävyysongelmaa. Se katsotaan käytettävyyttä vaikeuttavaksi ja se on korjattavien asioiden listalla.

3 tarkoitetaan suurta käytettävyyden ongelmaa. Sillä on merkittävä vaikutus ja se on korjattava välittömästi.

4 tarkoittaa katastrofaalista ongelmaa jonka voidaan katsoa tekevän palvelusta lähes käyttökelvottoman. Tällaiset ongelmat viivyttävät arvioitavan tuotteen julkistusta kunnes kyseiset virheet on korjattu. (Mielonen & Hintikka 1998.)

4 Käytettävyystesti

Käytettävyyden testauksella saadaan äärimmäisen tärkeää palautetta tuotteen kehittäjille (Wiio 2004, 61). Käytettävyyttä voidaan testata erilaisilla menetelmillä, joista

käytettävyystestit ovat yleisin tapa. Menetelmänä tämä on joustava ja sillä voidaan testata mitä vain. Sitä voidaan käyttää esimerkiksi www-sivujen, teollisuusrobottien, saavutettavuuden, esteettömyyden ja toimivuuden testaukseen. (Sinkkonen 2002.)

Käytettävyystestien tarkoituksena on tehdä ohjelman käyttölaadusta parempi. Testeissä pyritään löytämään käytettävyysongelmat testaamalla ohjelmaa käyttäjien avulla. Testaus toteutetaan seuraamalla käyttäjän toimintamalleja tilanteessa, joka muistuttaa mahdollisimman paljon oikeaa tilannetta. (Sinkkonen, Kuoppala, Parkkinen & Vastamäki 2006, 275; Etnoteam 2010.)

Käytettävyystestejä voidaan tehdä kahteen eri tarkoitukseen. Ensimmäisenä tarkoituksena käytettävyystestejä tehdään osana tuotteen kehitystyötä. Näissä testeissä etsitään virheitä, jotka sitten testauksen jälkeen raportoidaan eteenpäin korjattaviksi. Toisena tarkoituksena sovellusta testataan ennen levitystä. Siinä katsotaan, onko sovellus valmis eteenpäin levitettäväksi. Sovellus on saavuttanut käytettävyydeltään halutun tason, eikä siitä enää löydy toiminnallisia virheitä. (Sinkkonen ym. 2006, 275.)

Käytettävyystesti voidaan tehdä missä tahansa vaiheessa sovelluksen kehittelyä, riippuen siitä, mitä sillä mitataan. Testi voidaan tehdä alkuvaiheessa prototyypille, josta etsitään virheitä tai valmiille tuotteelle, jonka toimivuus kokonaisuutena halutaan tarkastaa ennen sen eteenpäin viemistä. (Kuutti 2003, 68.)

5 Käytettävyystestin toteutus

Käytettävyystestissä jokainen käyttäjä toimii yksin ja tekee testin/testit yksi kerrallaan, jonka jälkeen ne tallennetaan. Kun kaikki käyttävät ovat tehneet testin, niistä saatu tieto analysoidaan ja ongelmat määritellään ja mietitään mahdollisia korjaustapoja. Testeissä voi testata joko koko tuotteen tai sitten vain osan siitä, esimerkiksi tuotteen keskeisimmät ominaisuudet. Testin pituus voi vaihdella jopa minuutista päivään, mutta on keskimäärin noin tunnin mittainen. Käytettävyystestejä tehdessä täytyy muistaa, ettei kaikkia virheitä tulla löytämään, mutta kuitenkin suurin osa niistä. (Sinkkonen ym. 2006, 277.)

Käytettävyystestit aloitetaan viimeistään siinä vaiheessa, kun ensimmäiset prototyypit ovat valmiina. Testejä voidaan ja kannattaa tehdä koko tuotteen kehitysprosessin ajan. Varsinkin jos on isoja ja laajoja testejä, kannattaa miettiä niiden jakoa useampaan pienempään testiin. (Sinkkonen ym. 2006, 277.)

Käytettävyystesti voidaan jakaa karkeasti kolmeen luokkaan:

1. Testin valmistelu

2. Itse käyttäjätestin suorittaminen
3. Testitulosten analysointi sekä raportin kirjoittaminen

(Kuutti 2003, 70; Sinkkonen ym. 2006, 280–281.)

Nielsen (1993, 165) taas jakaa käytettävyystestin neljään eri tasoon:

1. Testin valmistelu
2. Testin esittely
3. Itse testi
4. Raportti

Kummatkin jaottelut ovat sisällöltään hyvin samanlaiset, erona on vain Nielsenin lisäämä ylimääräinen kohta.

5.1 Testin valmistelu, suunnittelu ja esittely

Testin valmistelu ja suunnittelu ovat hyvin iso osa onnistunutta testiä. Testiä suunnitellessa tulee ottaa huomioon, mihin tarkoitukseen testi on. Siinä on suuri ero, onko testi suunniteltu osaksi tuotteen kehitysprosessia vai tuotteen päättöarviota varten. Suunnitelmasta tulee siis löytyä tarkasti määriteltynä testin tavoitteet, kuten mitä tietoa testillä halutaan kerätä, mitä käytettävyyden tavoitteita on sekä millä käytettävyyttä mitataan. Edellä mainittujen tavoitteiden lisäksi suunnitelman tulisi sisältää testin budjetti. (Nielsen 1993, 171; Kuutti 2003, 70; Sinkkonen ym. 2006, 282.)

Tärkeä osa testin suunnittelua on koehenkilöiden valinta sekä testitehtävien laadinta. Testin painopisteet tulee päättää ja kirjata suunnitelmaan. Koehenkilöitä valittaessa tulee tietää tuotteen mahdollinen käyttäjäryhmä tai –ryhmät. Testiin tulisi valita henkilöitä halutusta käyttäjäryhmästä mahdollisimman laajasti. Käytettäessä useaa koehenkilöä tulisi käyttäjien edustaa mahdollisimman hyvin käyttäjäryhmää siten, että saadaan mahdollisimman monesta kategoriasta koehenkilö. (Nielsen 1993, 175.)

Testitehtäviä laadittaessa tulee ottaa huomioon haluttu testin laajuus, niiden helppous sekä tuotteen keskeisimmät toiminnot. Hyvät testitehtävät ovat helppoja, haastavia, helposti ymmärrettäviä ja monipuolisia. Hyvät testitehtävät myös testaavat tuotteen keskeisten toimintojen käytettävyyttä. (Sinkkonen ym. 2006, 285.)

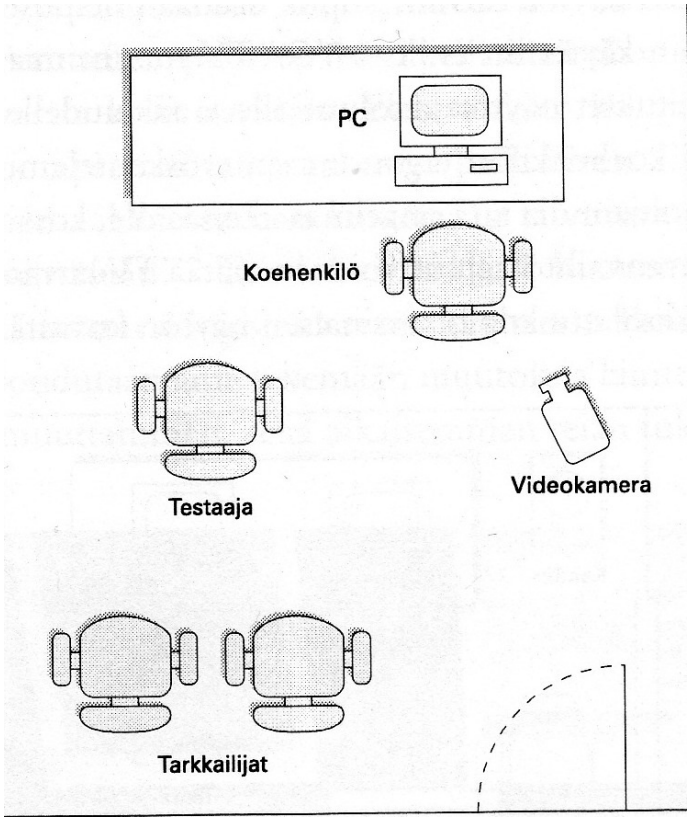
Kun testin suunnittelu ja valmistelut on saatu valmiiksi, tulee tehdä vähintään yksi pilottitesti. Yhtään käytettävyystestiä ei tulisi suorittaa ennen pilottitestiä. Pilottitestissä testataan, että kaikki testissä tarvittavat asiat toimivat ongelmitta, kuten laitteet ja testissä

käytettävät ohjelmistot. Usein riittää yksi tai kaksi testipilottia, mutta laajoissa testeissä saatetaan tarvita useampia. (Nielsen 1993, 174.)

Testin esittelyssä testin vetäjä toivottaa käyttäjän tervetulleeksi testiin ja antaa lyhyen yhteenvedon testistä ja siitä mitä se tulee pitämään sisällään. Käyttäjälle esitellään testimenetelmät sekä kerrotaan mitä testataan ja miksi. Esittelyosuuden jälkeen käyttäjä saa kaikki testissä tarvittavat kirjoitetut ohjeet testin tehtävät mukaan lukien. (Nielsen 1993, 188–190.)

5.2 Käytettävyystestin suorittaminen

Itse testitilanteen tulisi olla mahdollisimman luonnollinen ja testaajan tulisi tuntea olonsa mahdollisimman mukavaksi testitilanteessa. Laboratoriossa toteutettuja käytettävyystestejä onkin jonkin verran moitittu ja väheksytty siitä, että laboratorioympäristö on paljon häiriöttömämpi eikä ollenkaan niin aito kuin oikea käyttöympäristö. Tästä huolimatta havaittiin virhe kummassa tahansa ympäristössä, se on silti aito ja korjausta vaativa. (Nielsen 1993, 190; Sinkkonen 2002.)



Kuva 1: Esimerkki yksinkertaisesta käytettävyyslaboratoriosta (Kuutti 2003, 81).

Testin esittelyn jälkeen siirrytään itse testiin. Testin suorittaminen tapahtuu siten, että käyttäjä tekee hänelle annettuja tehtäviä omassa rauhassa itsenäisesti ohjaajan istuessa vieressä. Ohjaajan tulee olla mahdollisimman huomaamaton ja varoa häiritsemästä käyttäjää. Ohjaaja ei myöskään saa missään tilanteessa tuoda esiin omaa mielipidettään käyttäjän suoriutumisesta, onnistui tämä sitten hyvin tai huonosti ohjaajan mielestä. (Nielsen 1993, 190.)

Usein testitilanteissa on myös muita tarkkailijoita jotka laativat omia muistiinpanoja testiin liittyen. Koko testin ajan kaikkien muiden tarkkailijoiden tulisi olla aivan hiljaa ja mielellään jos mahdollista niin eri huoneessa. Ainoastaan testin ohjaaja saa puhua ja neuvoa testin aikana ja sekin vain silloin, kun käyttäjä kohtaa jonkun ylitsepääsemättömän ongelman. (Nielsen 1993, 190; Kuutti 2003, 74–75.)

5.3 Testiraportti

Testin jälkeen käyttäjä vielä haastatellaan ja hänelle saatetaan antaa jonkinlainen kyselylomake täytettäväksi. Käyttäjän lähdettyä ohjaaja tekee lyhyen raportin tehdystä testistä vielä, kun tapahtumat ovat hänen tuoreessa muistissaan. (Nielsen 1993, 191.)

Viimeisenä vaiheena testissä on testiraportin laatiminen. Kaikki testitulokset, mitä testien aikana saadaan, tulee pitää luottamuksellisena. Testiraportti tuleekin laatia siten, ettei siitä kukaan voi tunnistaa yksittäistä käyttäjää. (Nielsen 1993, 183.)

Kaikki testeissä kerätty informaatio käydään läpi, analysoidaan ja lajitellaan. Kaiken informaation jäsentelyn jälkeen se tulisi laittaa helposti käsiteltävään muotoon, esimerkiksi sähköiseen. Kun tuloksista havaitaan epäkohtia ja ongelmia, tulisi aina selvittää ongelman alkuperä, jotta siihen voidaan löytää looginen ratkaisu. (Kuutti 2003, 78–79.)

6 Ohjelmistotestaus

Ohjelmointitestaus on mitä tahansa sellaista tekemistä, jonka avulla pystytään arvioimaan ohjelman toimivuutta, ominaisuuksia sekä sitä kohtaako se asetetut tavoitteet. Testauksen avulla ohjelmasta tulisi löytää virheitä. Suurin osa ohjelmistojen virheistä on suunnitteluvirheitä. On hyvä muistaa, että ilman ohjelmaan tulevia päivityksiä harvoin löydetään uusia virheitä. (Pan 1999.)

Testaus ei ole pelkästään virheiden etsimistä. Sen tarkoituksena voi olla myös laadun varmistaminen, tarkastaminen tai validoiminen. Luotettavuuden testaus on myös osa

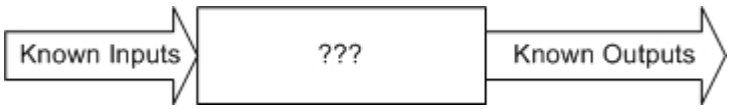
testausprosessia. Loppupeleissähän ohjelman luotettavuus, laatu ja turvallisuus ovat kaikki tiukasti yhteydessä toisiinsa. (Pan 1999.)

6.1 Testausmenetelmät

Ohjelmistoestaukselle on monenlaisia eri tekniikoita, jotka ovat tarkoituksensa mukaan jaoteltu eri ryhmiin. Oikeellisuuden testaus, luotettavuuden testaus, suorituskykytestaus ja turvallisuuden testaus ovat testaustekniikoita, jotka ovat jaoteltu tarkoituksen mukaan. Testaus voidaan myös jakaa sen elinkaaren mukaan. Vaatimusten testaus, suunnittelun testaus, ohjelman testaus, testaustulosten arviointi, asennuksen testaus, hyväksymisen testaus ja ylläpidon testaus ovat elinkaaren mukaan jaettu. Ohjelmistotestaus voidaan taas jakaa järjestelmätestaukseen, joka sisältää integraatio- ja julkaisutestauksen, ja komponenttitestaukseen. Ylläpidon testaukseen kuuluu regressiotestaus, joka tarkoittaa testausta, jolla varmistetaan, että bugit on korjattu. (Kelly 1997; Pan 1999; Blokdijk & Menken 2008, 50, 55-59.)

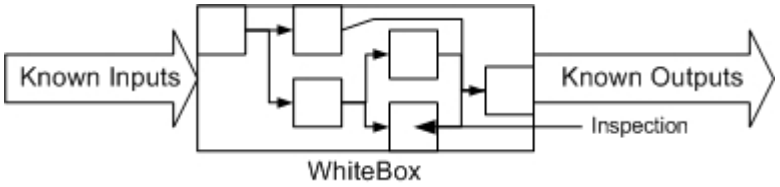
Oikeellisuuden testaus tarkastaa, että ohjelmalle asetetut minimivaatimukset täytetään. Se tarkoittaa sitä, että ohjelma toimii niin kuin sen pitäisikin. Oikeellisuuden testaus tarvitsee niin sanottuja oraakkeleita erottamaan oikean käytöksen väärästä lopputuloksesta. Esimerkkeinä oikeellisuuden testaukseen ovat black box- ja white box –testaus. Nämä eivät kuitenkaan rajaudu vain oikeellisuuden testaukseen. (Pan 1999.)

Black box –testaus on testausmetodi, jossa data on johdettu tietyistä toiminnallisista vaatimuksista. Sitä on kuvattu myös vaatimuspohjaiseksi testaukseksi. Siinä syötetään arvoja ja katsotaan tulokset, ilman, että nähdään, mitä ohjelman sisällä tapahtuu. (Pan 1999; Clifton 2003.)



Kuvio 2: Black box – testaus (Clifton 2003).

White box –testaus taas on testausmetodi, jota käytetään, kun ohjelman struktuurit ovat testaajalle näkyvissä. Tämä testaus toteutetaan ohjelmistototeutuksen tietojen mukaan, kuten ohjelmointikielen, logiikan ja tyylien. White box –testauksen testitapaukset johdetaan ohjelman struktuurista. White box –testauksessa nähdään oikeat syötteet ja väärät syötteet, jotta pystytään tarkistamaan virheidenhallinnan toimivuus. (Pan 1999; Clifton 2003.)



Kuvio 3: White box – testaus (Clifton 2003).

Järjestelmätestaus suoritetaan erillisen testaustiimin avulla, joka on keskittynyt testaukseen ja sen suunnitteluun. Nämä testit perustuvat järjestelmän määrittelyihin eli siihen miten järjestelmän tulisi toimia. (Blokdijk & Menken 2008, 50.)

Integraatiotestauksessa testataan eri luokkien yhteensopivuutta. Siihen voidaan käyttää joko bottom-up –menetelmää tai top-down –menetelmää. Bottom-up –menetelmä yhdistää ensin infrastruktuurikomponentit ja sitten vasta toiminnalliset komponentit. Top-down –menetelmä taas yhdistää ensin järjestelmän perustan ja sitten lisää sinne komponentteja. (Blokdijk & Menken 2008, 55.)

Julkaisutestauksen avulla pyritään saamaan hankkijan luottamus järjestelmän toiminnallisuuteen ja vaatimuksiin. Julkaisutestaus on yleisimmin toiminnallisuuden testausta tai Black box –testausta ja perustuu järjestelmän määrityksiin. Julkaisutestausta on aika lähellä luotettavuustestausta, jonka avulla ennustetaan ohjelman luotettavuutta tulevaisuudessa esimerkiksi niin että tutkitaan black box –testauksessa saatua dataa. Datan avulla pystytään määrittelemään, milloin ohjelma kannattaa julkaista. Jos ohjelma sisältää vielä paljon virheitä, tulee julkaisua siirtää. (Pan 1999; Blokdijk & Menken, 2008, 55.)

Komponenttitestaus on yleensä komponenttien kehittäjien vastuulla. He pystyvät luomaan testejä käyttäen hyväksi omaa kehittäjän kokemusta ja tietotaitoa. Tämä testitapa testaa vain yksittäisiä komponentteja. Komponenttitestejä yhdistämällä voidaan rakentaa pohjaa integraatiotestaukselle. (Blokdijk & Menken 2008, 55)

Suoritustasotestaus sekä stressitestaus ovat tapahtuvat käyttöliittymän takana itse prosesseissa. Esimerkkinä tästä ovat testit, jotka mittaavat kuinka paljon samanaikaisia latauksia tai tilauksia järjestelmä kestää, milloin suoritustaso laskee ei hyväksyttävälle tasolle sekä kuinka paljon samanaikaista käyttöä se kestää. Jokaiselle ohjelmalle on määritelty tietty aika, jonka aikana sen tulisi selviytyä tietyistä tehtävistä. Esimerkiksi kuinka kauan kestää sivun latautuminen. Suorituskyvyn arviointi sisältää resurssien käytön, eli kuinka paljon resursseja tietty toiminto vie ohjelmalta, suoritustehon, eli kuinka monta samanaikaista tehtävää se pystyy hoitamaan, sivuston vastausajat, esimerkiksi sivun latautuminen sekä sen kuinka monta tehtävää on jonossa. (Pan 1999; Blokdijk & Menken 2008, 58-59.)

Turvallisuuden testaus edesauttaa ohjelmaa selviytymään esimerkiksi hakkerien hyökkäyksistä. Turvallisuustestauksella yritetään poistaa tietoturva-aukkoja ja muita ohjelman ongelmia, joita hakkerit voisivat käyttää hyväkseen. Paras keino tämän testaukseen on tehdä simuloituja turvahyökkäyksiä. (Pan 1999.)

6.2 Testauksen tarkoitus

Testauksen tarkoituksena on löytää mahdollisia ongelmatilanteita ja vikoja. Näitä voi ilmetä aina uuden päivityksen myötä. Jokaisen päivityksen jälkeen olisi hyvä tehdä samat testit kuin aiemmin ja katsoa saadaanko samat lopputulokset. Toisto ja väärien tuloksien saaminen ovat oleellisia testauksen kannalta. Mitä enemmän virheitä, sitä paremmin testaus on tehnyt tehtävänsä. Jos testaus ei löydä virheitä, se ei tarkoita etteikö niitä olisi. (Pan 1999; Blokdijk & Menken 2008, 51, 57.)

Panin mukaan (1999) testausta voisi pitää omana taiteenmuotonaan. Vielä nykyäänkin käytetään samoja testausmenetelmiä, jotka keksittiin vuosikymmeniä sitten. Jotkin näistä testausmenetelmistä ovat lähinnä heuristisia eivätkä juurikaan teknisiä. Testaus saattaa olla myös hyvinkin kallista, mutta yleensä vielä kalliimpaa on testaamatta jättäminen. Paljon helpompaa on selvittää ongelmat juuri heti niiden ilmettyä kuin monien viikkojen jälkeen, kun ongelmat tulevat esiin käytössä.

7 Testiautomaatio

Testiautomaatio kuuluu osaksi testausta ja on koneella suoritettavaa. Koska kone pystyy suoriutumaan samoista testeistä nopeammin kuin ihminen, kannattaa osa testeistä automatisoida. Testiautomaatio kannattaa ottaa manuaalitestauksen lisäksi testaukseen nopeuttamaan testausta ja helpottamaan manuaalitestaajien työsarkaa. Kone myös toistaa testit muuttumattomina paremmin kuin ihminen.

7.1 Testiautomaation suunnittelu

Testiautomaatio tulisi suunnitella ennen sen käyttöön ottoa. Suunnittelussa tulisi ottaa esille mitä välineitä käytetään, onko testiautomaatio kannattavaa ja oleellista ohjelman toimivuuden testauksessa, kuinka testiautomaatio aiotaan hoitaa sekä kuinka paljon resursseja testiautomaatio vie. Testimanagerit tulisi kouluttaa erottamaan faktat fiktiosta ja tuntemaan testiautomaation hyödyt. Erilaisia testaustyökaluja käsitellään luvussa 7.2. (Beall 2008, 6; Alam, 3.)

Testityökalut, testiscriptit, testitapausten implementointi, tulosten monitorointi ja tulkkaus kuuluvat testwaren suunnitteluun. Testiautomaatio tuottaa paljon enemmän dataa kuin manuaalitestaus jo senkin takia, että testaukseen kuluu vähemmän aikaa, eli samassa ajassa pystytään ajamaan enemmän testejä. Tämän datan tulkitsemiseen tarvitaan ihminen erottelemaan oikeat bugit testien muista ongelmista, kuten timeout-ongelmista. (Hoffman 1999, 1.)

Testityökalujen tutkiminen ennen sen käyttöön ottoa on todella tärkeää jo senkin kannalta, että työkalujen lisenssit saattavat maksaa ja jos työkalu ei olekaan sopiva testattavan ohjelman kanssa, on turhaan käytetty rahaa hukkaan. Alam kertoo, että työkalut kannattaisi valita niiden käyttäjien kanssa, koska yleisimmin he tuntevat testattavan ohjelman parhaiten ja tietävät millaisen työkalun he tarvitsevat. Bach (1998, 3, 5) on tehnyt listan siitä, mitä tulisi ottaa huomioon, kun valitaan testiautomaatiotyökalua. Se sisältää seuraavat kysymykset:

- Onko työkalulla tarvittavat ominaisuudet ohjelman testaukseen?
- Toimiiko työkalu pitkään ilman ongelmia, löytyykö siitä ongelmia?
- Pystyykö työkalulla ajamaan useita testejä jopa useita päiviä putkeen?
- Onko työkalu helppokäyttöinen ja oppiiko sen nopeasti?
- Onko työkalun ominaisuudet hankalia?
- Säästääkö työkalu enemmän aikaa testikehityksessä kuin manuaalitestaus tiettyjen testien osalta?
- Toimiiko työkalu testattavan ohjelman kanssa?
- Kuinka hyvin se simuloi oikeata käyttäjää?
- Onko käytös samanlaista automaation kanssa tai ilman?
- Työkalussa tulisi olla raportit epäonnistuneille ja läpäisseille testeille.

Ennen testityökalun ostamista kannattaa miettiä automaation kuluja, eli kuinka paljon maksaa automaation kehittäminen, automaatiotestien operointi ja automaation ylläpito. Yleensä testiautomaatio koetaan kalliiksi tavaksi tehdä testejä, mutta jos se osataan suunnitella oikein, tulevat kustannuksetkin olemaan halvemmat kuin ilman testiautomaatiota. (Bach 1998, 4.)

Kun testityökalu on valittu, tulisi miettiä mitkä kaikki testeistä kannattaa automatisoida. Automaatiotestauksessa voidaan tehdä myös sellaisia testejä, joita ihminen ei pysty suorittamaan. Esimerkiksi suorituskyvyn testaus on paljon helpompaa ja halvempaa suorittaa koneella kuin pyytää 300 henkilöä käyttämään samanaikaisesti ohjelmaa ja sen ominaisuuksia. (Marick 1998, 2.)

Testitapauksia suunniteltaessa tulisi ottaa huomioon:

- Ovatko testitapaukset helposti ylläpidettäviä?
- Ovatko testit kustannustehokkaita?
- Ovatko testit luotettavia ja käytettäviä?
- Selviävätkö ne odottamattomista tapauksista?
- Voidaanko niitä ajaa eri ympäristöissä?
- Onko testidata joustavaa muutostilanteissa?
- Onko testien ajoaika tehokas?
- Ovatko testit itsenäisiä vai riippuvaisia toisistaan?
- Ovatko testit yksinkertaisia?
- Onko ne dokumentoitu?

(Fewster & Graham 1999, 202, 227.)

Testitapausten tekeminen pitää organisoida. Manuaalitestaajien tulisi tehdä testitapaukset, jotka testien automatisoija ohjeiden mukaan automatisoi. Testitapauksia tehdessä tulee myös huomioida, millaisia testejä kannattaa tehdä. Esimerkiksi samasta testistä kannattaa tehdä useita eri variaatioita, joissa testataan eri asioita. Testejä saattaisivat olla esimerkiksi kaikkien kenttien täyttäminen ja vain pakollisten kenttien täyttäminen. Näin saadaan kahden eri tapauksen erot selville. (Marick 1998, 13; Hoffman 1999, 1.)

Testaukselle on viisi kehitysastetta (kuvio 4). (1) Ensin tulee tunnistaa testiolot, jonka jälkeen testi voidaan (2) suunnitella. (3) Suunnittelun jälkeen testi voidaan rakentaa. (4) Rakentamisen jälkeen se voidaan ajaa ja (5) testitulokset voidaan tutkia testien ajon jälkeen. Testiolojen tunnistamisessa mietitään mitä voidaan testata ja mitä ei. Suunnittelussa mietitään miten mitäkin testataan. Rakentaminen käsittää testitapausten luomisen valitulla testiautomaatiotyökalilla ja tämän jälkeen ne voidaan ajaa tietyillä komennoilla. Kun testit on ajettu tutkitaan tuloksia ja katsotaan, vastaavatko ne ennalta suunniteltuja odotettuja lopputuloksia. (Fewster & Graham 1999, 15-17.)

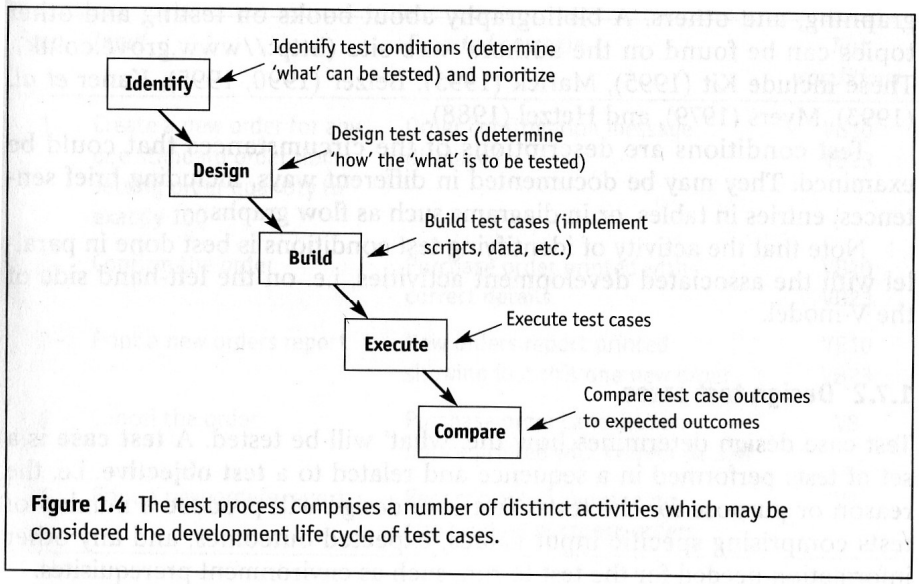


Figure 1.4 The test process comprises a number of distinct activities which may be considered the development life cycle of test cases.

Kuvio 4: Testauksen viisi kehitysastetta (Fewster & Graham 1999, 13).

Kuten edellä jo mainittiin testituloksia pitäisi voida vertailla jo ennalta määrättyihin lopputuloksiin. Nämä ennalta määrätyt lopputulokset päätetään joko ennen testien ajoa tai sitten tutkitaan ensimmäistä tulosta ja katsotaan voisiko se olla sellainen. Ennalta määrätyt lopputulokset valitaan yleensä testausoraakkelien avulla. Manuaalitestauksessa oraakkeli on yleensä ihminen, joka testaa testit. Testiautomaatiossa kuitenkin yleensä käytetään konetta, koska ihmisoraakkelit eivät ole täydellisiä ja koneen avulla voidaan koko testausoperaatio automatisoida. Ihminen väsyy helposti ja saattaa tämän takia ohittaa virheitä tai nähdä niitä siellä, missä niitä ei olisi. Koneoraakkelien käyttö on nopeampaa, koska kone pystyy nopeammin väläyttelemään eri ruutuja kuin ihminen ottaa kuvakaappauksia. (Hoffman 1999, 2, 5.)

Jos testi on sellainen, että se muuttaa dataa, tulee tiedonsyöttämisen jälkeen tarkastaa, että tieto on varmasti mennyt ohjelmaan tarkastamalla ohjelmiston tila. Esimerkiksi ohjelmassa saatetaan luoda uusi asiakas ja syötekenttiin tulee syöttää nimi, y-tunnus ja muita tarpeellisia tietoja. Kun on painettu tilaa-nappia, tulee sivulle tulla tiedot luodusta asiakkuudesta. Jos testin taas ei ole tarkoitus muuttaa dataa, on hyvä tarkastaa, että data ei ole muuttunut. (Hoffman 1999, 4.)

Testauksen osatarkoituksena on varmistaa jo korjattujen bugien katoaminen. Kun testi on löytänyt bugin ja ohjelmoija on sanonut korjanneensa sen, on hyvä ajaa testi uudestaan, jotta nähdään onko bugi korjaantunut. Automaation löytämiä bugeja kannattaa käsin kokeilla pystytäänkö toistamaan sama tilanne myös käsin. Samojen bugien löytyminen uudestaan on helppoa testiautomaation avulla, koska testitapaukset tekevät kaiken joka ajokerralla samalla

tavalla. Testitapaukset on helpompi ajaa päivittäin. Päivittäisen ajon jälkeen löydetty bugi on helpompi korjata, koska ohjelmoijan ei tarvitse debugata koko koodia läpi vaan ainoastaan päivän ajalta tulleet muutokset. Tämä helpottaa ohjelmoijien työtä. (Marick 1998, 1.)

7.2 Erilaiset testiautomaatiotyökalut

Erilaisia testiautomaatiotyökaluja löytyy yhtä paljon kuin erilaisia käyttäjiäkin. Tässä esitellään muutamat ensin sukupolvien mukaan ja sitten vielä muutamat, joita ei ole lajiteltu mihinkään kohtaan testiautomaation kehityshistoriassa.

Hinz ja Gijsen kertovat, että testiautomaatiota on kehitetty vuosien aikana ja nyt ollaan menossa viidennessä sukupolvessa. Taulukossa 1 on lueteltu nämä viisi eri sukupolvea. Ensimmäinen sukupolvi käsitti äänitys ja soitto –testiautomaation, jossa testitapaukset äänitettiin. Tämä ei ole kannattava tapa enää suorittaa testausta, koska tämän tavan testiscriptit sisältävät kovakoodattuja arvoja, joiden tulee muuttua, jos mikä tahansa muuttuu sovelluksessa. Testitiimin pitää päivittää skriptit aina, kun data muuttuu, mikä vie paljon aikaa. Nämä skriptit sisältävät hyvin vähän virheiden hallintaa eivätkä siksi ole luotettavia. Jos sovellus muuttuu, tulee testit äänittää uudestaan. (Godase 2003.)

Toisen sukupolven testiautomaatio käytti uudestaan ja uudestaan samoja funktioita testiskripteissä. Kolmas sukupolvi käsitti taas datakeskeiset skriptit ja funktiot. Datakeskeinen testaus on sitä, että testitapausten syöttöjen ja tulosten arvot ovat luettavissa datatiedostoista ja ovat ladattavissa muuttujiksi manuaalisissa koodausskripteissä. Tämä framework tarvitsee muuttujia sekä syöttöihin että tulosteisiin. (Alliance Global Services 2009, 4; Hinz & Gijsen, 6.)

Neljäs sukupolvi paneutui avainsanaskripteihin. Tämän testiautomaatio frameworkit ovat Hans Buwaldan kehittämiä, kuten TestFrame. Yleensä avainsanat päätyvät kolmannen osapuolen testiautomaatiotyökalujen pohjaksi. Avainsanakeskeinen framework tarvitsee avainsanojen kehitystä. (Alliance Global Services 2009, 4; Hinz & Gijsen, 6.)

Viides automaationsukupolvi on taas kokonaan skriptitön. Se perustuu toimintoihin, joissa avainsanat ovat ensisijaisesti uudelleen käytettäviä testitapauksesta toiseen. Tästä syystä sen ylläpito hoidetaan avainsana-tasolla. Mikä tahansa muutos avainsanassa heijastaa kaikkiin testitapauksiin, joissa sitä käytetään. (Hinz & Gijsen, 6.)

1st Generation	Record and Playback
2nd Generation	Use/reuse of functions in test scripts
3rd Generation	Data Driven scripts/functions
4th Generation	Action word (keyword) scripts/functions
5th Generation	Scriptless Automation

Taulukko 1: Testiautomaation sukupolvet (Hinz & Gijsen, 4).

Toiminnallisuuskeskeisessä metodissa erotellaan data funktioista. Tässä testausmetodissa voidaan skriptit luoda ennen kuin sovellus on edes valmistunut. Skriptien uudelleenkäyttö on laajaa ja odotettujen tulosten ylläpito on helppoa. Tähän kuitenkin tarvitaan tekninen henkilö ylläpitämään testitapauksia. (Godase 2003.)

Avainsanakeskeinen metodi on helpommin luotavissa ja ylläpidettävissä kuin toiminnallisuuskeskeinen metodi. Tämän oppiminen on helppoa ja jopa ei-tekninen henkilö kokee sen helpoksi käyttää. Myös ylläpidettävyys on helppoa. Kuitenkin, jos luodaan skriptejä monimutkaisiin skenaarioihin, niiden tekeminen voi viedä aikaa. (Godase 2003.)

Moduulitestausmetodissa luodaan pieniä itsenäisiä skriptejä, jotka edustavat testauksen kohteena olevan sovelluksen moduuleja, sektioita ja funktioita. Näiden avulla luodaan isompia testejä. Tämä tapa on sopiva isojen ja vakaiden sovellusten automaatiotestaukseen. (Alliance Global Services 2009, 4.)

Testikirjastotestausmetodissa jaetaan testauksen alla oleva sovellus pieniin proseduureihin ja funktioihin eikä skripteihin. Tämä sopii pienten ja keskikokoisten vakaiden sovellusten automaatiotestaukseen. (Alliance Global Services 2009, 4.)

Yleisimmin käytetään kuitenkin metodia, joka on sekoitus kaikista eri metodeista. Hybridissä pystytään yhdistämään kaikkien teknologioiden parhaat puolet. Parhaiten yhdistetty framework on datakeskeisen ja avainsanakeskeisen välinen metodi. (Alliance Global Services 2009, 4.)

7.3 Testien automatisointi ja ylläpito

Fewster ja Graham (1999, 4) tuovat esiin, että testiautomaatio on erilaista kuin manuaalitestaus. Testiautomaatiotestit tuleekin sen takia valita huolellisesti. Yleisesti parhaimmat hyödyt saadaan testeillä, joita voidaan toistaa.

Testiautomaatiolla on yleensä omat tekijänsä, joita kutsutaan testiautomoijiksi tai automaatiotestaajiksi. Nämä testiautomoijat tulisi pitää vain yhden projektin parissa kerrallaan, jotta paras mahdollinen lopputulos saataisiin aikaan ja myöskin siksi, että aikaa ei menisi hukkaan vaihdettaessa kesken kaiken projektista toiseen ja taas takaisin. (Fewster & Graham 1999, 5; Beall 2008, 5.)

Testaukseen on aina määritelty joku aika, jonka aikana se tulisi suorittaa. Tämän ajan mukaan pystyy päättelemään, mitkä testit kannattaa automatisoida, jotta testaus onnistutaan suorittamaan annetun ajan puitteissa. Testiautomaatiotestejä valittaessa tulisi ottaa huomioon mitkä mahdollisuudet niillä on löytää bugeja myös tulevaisuudessa, jos ne löytävät niitä nyt. (Marick 1998, 2.)

Hyvä testaus on interaktiivista ja kognitiivista. Automaatiotestaus toimii parhaiten pienessä osassa testausta, koska kaikkia testejä ei kannata automatisoida. Jos kaikki testit automatisoidaan, käytetään rahaa turhaan, kun testejä pitää muutella koodin muuttuessa. Näin saadaan suhteellisen heikkoja testejä, jotka eivät löydä bugeja. Jos testi ei ensimmäisellä ajokerralla löydä bugeja, pienenee sen mahdollisuudet löytää niitä koskaan elleivät ne johdu koodimuutoksista. (Bach 1999, 2.)

V-mallin (kuva 6) tarkoituksena on, että jokaista koodivaihetta vastaa testausvaihe. Se on hyvä tapa näyttää jokaiselle koodausvaiheelle oma testivaihe. V-malli aiheuttaa sen, että bugit huomataan ajoissa, jolloin niiden korjaaminen on halvempaa. (Fewster & Graham 1999, 5.)

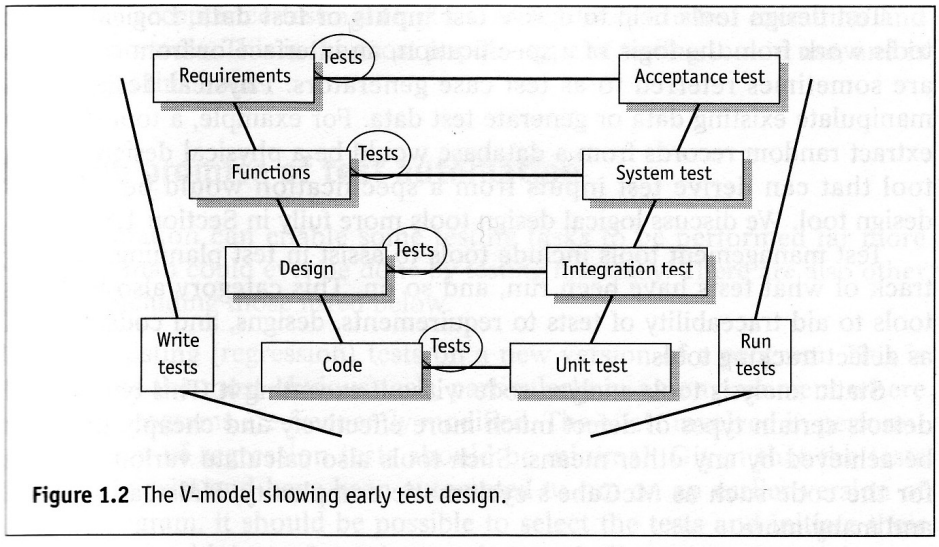


Figure 1.2 The V-model showing early test design.

Kuvio 5: V-malli (Fewster & Graham 1999, 7).

Smoke-testit kannattaa automatisoida, koska niitä ajetaan päivittäin. Smoke-testit ovat testejä, jotka testaavat perustoimintoja ja niiden avulla pitäisi löytää suurimmat ongelmat. Jos smoke-testit menevät läpi, on koodi tarpeeksi vakaata muuten testattavaksi. (Marick 1998, 14.)

Marick (1998, 4) kertoo, että testiautomaatiota mietittäessä tulee myös arvioida kuinka monta koodimuutosta testit kestävät. Jos koodi muuttuu niin paljon, että testit hajoavat, tulee miettiä kannattaako testiä muuttaa vai onko se enää edes tarpeellinen. Jos testi koetaan hyödyttömäksi, tulee se poistaa, jotta testisarjaan ei kasaannu hyödyttömiä testejä. Kuitenkin paras ratkaisu on koodimuutosten aiheuttamien ongelmien minimointi. (Hoffman 1999, 2.)

Kokonaan automatisoitu testi tarkoittaa sitä, että ajetaan kahta tai useampaa testitapausta samanaikaisesti. Testiautomaatiossa ei tarvitse puuttua testitapausten ajoon, kun testi on käynnistetty. Koneen tulisi itse automaattisesti asentaa ja tallentaa testiympäristön parametrit, ajaa testitapaukset, tallentaa tarpeelliset tulokset, vertailla todellisia tuloksia odotettuihin tuloksiin ja huomata erot sekä analysoida ja raportoida läpäisseet ja epäonnistuneet testit. (Hoffman 1999, 2.)

Automaatiossa tulisi ottaa huomioon, että on henkilökuntaa, joka voi ylläpitää testejä eikä vain tehdä uusia. Yksi testiautomaation suurimmista haasteista on juuri näiden testien ylläpito. Nykyään ohjelmistosuunnitelmat yleensä toteutetaan kahden viikon jaksoissa (scrum tekniikka), jolloin myös koodi muuttuu ja julkaistaan nopeammin, mikä saattaa aiheuttaa testitapausten hajoamisen. Osa testiautomoijan työajasta menee juuri näiden hajonneiden testitapausten korjaamiseen. (Beall 2008, 6; Hinz & Gijsen, 5.)

Jokaista uutta toimintoa kohden tulisi tehdä uusi testitapaus, jotta saadaan testattua ja selvitettyä mahdolliset ongelmat siinä ja aiheuttaako se ongelmia vanhaan koodiin yhdistettynä. Testitapaukset tulisi tehdä sellaisiksi, että ne imitoisivat oikeata käyttäjää. Tämä saadaan aikaiseksi käyttötapausten avulla. Näiden avulla löydetään samoja bugeja, joita käyttäjät löytäisivät. (Marick 1998, 15-16.)

Jokaisen ajon jälkeen tulisi automaatiotestauksessa käydä läpi menivätkö testit läpi vai epäonnistuivatko ne ja jos epäonnistuivat, mihin ne kaatuivat. Tämä tarkoittaa sitä, että kaikki automaatiotestit tarvitsevat kuitenkin ihmisen apua. Ihmisen tulisi vertailla saadut tulokset odotettuihin tuloksiin. (Bach 1999, 3; Hinz & Gijsen, 4.)

7.4 Myyttejä ja ongelmia testiautomaatiossa

Testimanagereilla on jonkinlainen käsitys testiautomaatiosta ennen sen käyttöönottoa. Yleensä käsitys on kuitenkin väärä ja ollaankin myyttien puolella. Tämän takia testimanagerit tulisi kouluttaa tunnistamaan testiautomaation hyödyt ja rajoitteet. Tämä auttaa tekemään testiautomaatiosta sujuvampaa ja täyttämään tarpeet. (Bach 1999, 1, 3.)

Testiautomaatiosta oletetaan, että se löytää enemmän bugeja kuin manuaalitestaus. Asia ei kuitenkaan ole näin. Automaatiotestit eivät ole niitä, jotka bugeja löytävät, vaan niiden pohjalla olevat kirjoitetut testitapaukset. Testitapausten kirjoittaminen on yleensä testaajan vastuulla ja testaajan kirjoittamien testitapausten perusteella testit automatisoidaan. Tulee ottaa huomioon se, että jos testit eivät ole löytäneet ensimmäisellä kerralla ongelmia, ei seuraavallakaan ajokerralla löydetä mitään, ellei koodi ole muuttunut. (Alam, 1; Fewster & Graham 1999, 11.)

Alam (2) tuo lisäksi esille toisen myytin. Testimanagerit olettavat, että automaatiotestaus eliminoi tai vähentää manuaalitestaajien tarvetta. Tämäkään ei ole ihan näin. Manuaalitestaajat tuntevat ohjelman kuin omat taskunsa. He ovat juuri niitä, jotka tekevät testitapaukset, joiden pohjalta tehdään automaatio. Vähennys ei näin ollen ole tarpeen.

Testimanagereiden epätodelliset odotukset ovat yksi suuri ongelma. He saattavat luulla testiautomaation ratkaisevan kaikki ongelmat. He saattavat valita työkalut demojen ja myyntipuheiden perusteella, joissa näytetään kuinka helppoa testiautomaatio on. Demojen ja myyntipuheiden tarkoitushan on myydä, eikä tuoda esille epäkohtia tai ongelmia. Nämä saattavat viedä testityökaluista päättäviä ihmisiä harhaan. Testityökalut tulisi valita testaajien kanssa, koska he tuntevat ohjelman ja tietävät millaisen testityökalun he tarvitsevat. (Fewster & Graham 1999, 10; Alam, 2.)

Jos testit ovat alunperinkin suunniteltu huonosti, on niitä turha lähteä automatisoimaan. Ensin tulisi korjata testit paremmiksi ja vasta sitten automatisoida ne. Testiautomaatio vaatii myös ylläpitoa. Jos testit hajoavat, tarvitaan joku korjaamaan ne. Lisäksi itse testiautomaatio-ohjelmat saattavat sisältää bugeja ja näin hankaloittaa testausta. (Fewster & Graham 1999, 11.)

Testimanagereilla saattaa olla vääränlainen turvallisuuden tunne, jos testien ajon jälkeen ei löydykään yhtään bugeja. Se, että testi ei löydä bugeja, ei tarkoita sitä, että niitä ei olisi. Tämä tarkoittaa vain sitä, että testi ei löytänyt yhtään. (Fewster & Graham 1999, 11.)

Testiautomaatiolla on kuitenkin monia hyötyjä verrattuna manuaalitestaukseen. Esimerkiksi regressiotestien ajaminen uudessa ohjelmaversiossa on nopeampaa automaatiolla kuin manuaalisesti. Useiden testien ajaminen samanaikaisesti nopeuttaa testiprosessia.

Automaatiotesteillä pystytään myös ajamaan sellaisia testejä, mitä ei manuaalisesti voida ajaa, kuten stressitestit. (Fewster & Graham 1999, 9-10.)

Fewster ja Graham (1999, 10) tuovat esille sen, että automaatiotestaus auttaa resurssien hyödyntämisessä. Tylsät tehtävät, kuten samojen syötteiden syöttäminen, pystytään automatisoimaan helposti ja saamaan näin tarkkuutta testeihin koneen syöttäessä aina täysin samat arvot, mihin ihminen ei välttämättä pysty. Testejä voidaan käyttää uudelleen ja toistaa useammin kuin manuaalisesti. Automaatiotestauksen vähentäessä testaukseen menevää aikaa saadaan testattava tuote aikaisemmin markkinoille.

7.5 Automaatio- vai manuaalitestaus

Manuaalitestaus ja automaatiotestaus eivät poissulje toisiaan. Kuten aiemmin on jo todettu manuaali- ja automaatiotestaus tukevat toisiaan. Seuraavaksi käydään läpi, miten valita manuaalitestauksen ja automaatiotestauksen väliltä.

Testiautomaatiota voidaan käyttää testauksen tukena esimerkiksi jo edellä mainittuihin stressitesteihin, jotka olisi mahdotonta tai vaikeaa järjestää manuaalisesti. Kuitenkin testit, joita ajetaan vain kerran eivät ole automaation arvoisia. Testejä, joiden tekemiseen menee moninkertainen aika verrattuna käsin testaamiseen, on turha automatisoida. (Hoffman 1999, 3; Godase 2003; Beall 2008, 4.)

Toisaalta taas jotkut tehtävät ovat hankalia koneille, mutta helppoja ihmisille. Esimerkiksi tulosten tulkinta on ihmiselle paljon helpompaa kuin koneelle. Innovatiivisessa ohjelmistokehityksessä on paljon muutosta uusien ominaisuuksien myötä ja se lisää automaation epävarmuutta. (Bach 1999, 3.)

Marickin (1998, 17) mukaan ihmiset saattavat löytää bugeja, joita koneet eivät löydä, ja koneet bugeja, joita ihmiset eivät löydä. Esimerkiksi muutokset käyttöliittymässä eivät välttämättä vaikuta koneen testitapauksen ajoon. Värien muutos sivulla voi jäädä koneelta huomaamatta. Toisaalta koneet pystyvät näkemään datastruktuurit käyttöliittymän takana, mihin ihminen ei pysty. Ihmiset tekevät testit aina vähän eri tavalla. He saattavat kirjoittaa jotain väärin ja tämän jälkeen korjata kirjoittamansa. He saattavat myös tehdä virheitä tai kokeilla syöttää tekstiä uudestaan. Näiden avulla voi löytyä bugeja, joita koneet eivät löydä.

Manuaalitestaus sopeutuu helposti erilaisiin muutoksiin ja pystyy selviytymään niistä. Ihmiset pystyvät löytämään satoja ongelmia toimintamalleista. Tämä vaihtelevuus ja muutoksista selviytyminen on manuaalitestauksen vahvuus verrattuna automaatioon. Manuaalitestauksessa

voi paremmin keskittyä uusiin ominaisuuksiin ja kohtiin, joista saattaisi löytyä uusia bugeja. (Bach 1999, 2.)

Testiautomaatio voi huomattavasti vähentää panostusta riittävään testaukseen tai lisätä testausta, joka pitää suorittaa tietyssä ajassa. Testiautomaatiolla helpot tehtävät ovat paljon nopeampia tehdä kuin manuaalisesti. Testiautomaatio käyttää samoja syötteitä aina. Tätä ei saada aikaiseksi manuaalitestauksella. (Fewster & Graham 1999, 3.)

Manuaalitestaus ja automaatiotestaus ovat lopulta kuitenkin kaksi eri prosessia ja niitä on tämän takia turha vertailla. Niiden dynamiikat ovat erilaisia ja niiden löytämät ongelmat ovat erilaisia. Kuitenkin on hyvä tietää vähän niiden eroista. (Bach 1999, 4.)

8 Robot Framework

Robot Framework on avainsanakeskeinen testiautomaatiotyökalu, jonka on kehittänyt Pekka Klärck Nokia Siemens Networksille vuonna 2005. Vuonna 2008 ohjelmasta julkaistiin open source –versio. (Larman & Vodde 2010, 4.)

Robot Frameworkissa automaatiotestit kirjoitetaan taulukkoihin käyttäen HTML:ää, TSV:tä, rEST:iä tai tavallista tekstiä. TSV-muotoiset (tab separated values) taulukot voi tuoda Exceliin tai vastaavaan taulukko-ohjelmaan. rEST-tiedostot on luotu marginaalikielellä, jota usein käytetään dokumentointiin Python-projekteissa. HTML ja tavallinen teksti ovat käytetyimpiä tapoja. HTML:ssä Robot Framework käyttää vain taulukoita eikä huomioi mitään ylimääräistä tekstiä, jota voi käyttää dokumentointiin. Työkalussa on neljä erilaista taulukkotyyppiä:

- Testitapaustaulut
- Avainsanataulut
- Asetustaulut
- Muuttujataulut.

(Larman & Vodde 2010, 6.)

Testitapaustaulut sisältävät nimensä mukaan testitapaukset. Ensimmäisen solun tulisi sisältää “Test Case” tai “Test Cases”, jotta taulukko tietää, mistä taulukosta on kyse. Avainsanatauluissa on alemman tason keywordit, joiden avulla rakennetaan testitapaukset. Ensimmäiseen soluun tässä taulussa tulee “Keyword” tai “Keywords”. Asetukset-taulu sisältää metadataa ja sinne voidaan tuoda myös ulkopuolisia tiedostoja, kuten esimerkiksi SeleniumLibraryn, jota käytetään selaintestaukseen. Asetustaulun ensimmäisen solun tulee sisältää “Setting tai “Settings”. Muuttujataulu kertoo käytettävät muuttujat, jotka sisältävät

globaalia dataa, jota voidaan käyttää missä tahansa. Tämän taulun ensimmäinen solu sisältää “Variable” tai “Variables” –sanan. (Larman & Vodde 2010, 6.)

Robot Frameworkissa testit jaotellaan tagien avulla. Tämä helpottaa esimerkiksi testien ajoa, kun halutaan ajaa vain tietty ryhmä, eikä kaikkia testitapauksia. Robot Frameworkilla pystyy tekemään myös omia avainsanoja, niin sanottuja ylemmän tason avainsanoja, joilla saadaan testeistä luettavampia. Tutkimuksessamme (luku 9) on esimerkki ylemmän tason avainsanasta (Kuva 7). Ylemmän tason avainsanoja luodaan samalla tavalla kuin testitapauksia. (Larman & Vodde 2010, 14; Robot Framework 2011.)

Robot Frameworkiin on useita eri kirjastoja, kuten Swingille (Java-pohjainen) ja Seleniumille (selaintestaukseen tarkoitettu) omat. Robotiin pystyy myös itse tekemään kirjastoja Pythonin tai Javan avulla ja tuomaan ne testitapauksiin. Tutkimuksessamme (luku 9) on käytetty selaintestaukseen käytettävää Selenium-kirjastoa. (Robot Framework 2011; Larman & Vodde 2010, 14.)

Testiajojen jälkeen Robot tulostaa raportit ja logit HTML-muotoon. Näistä näkee selvästi ovatko testit läpäisseet vai eivät. Robot sisältää myös oman integroidun kehitysympäristön (Robot Framework Integrated Development Environment eli RIDE). Sen avulla pystyy helposti muokkaamaan ja luomaan testidataa Robot Frameworkiin. (Larman & Vodde 2010, 14; robotframework-ride 2011.)

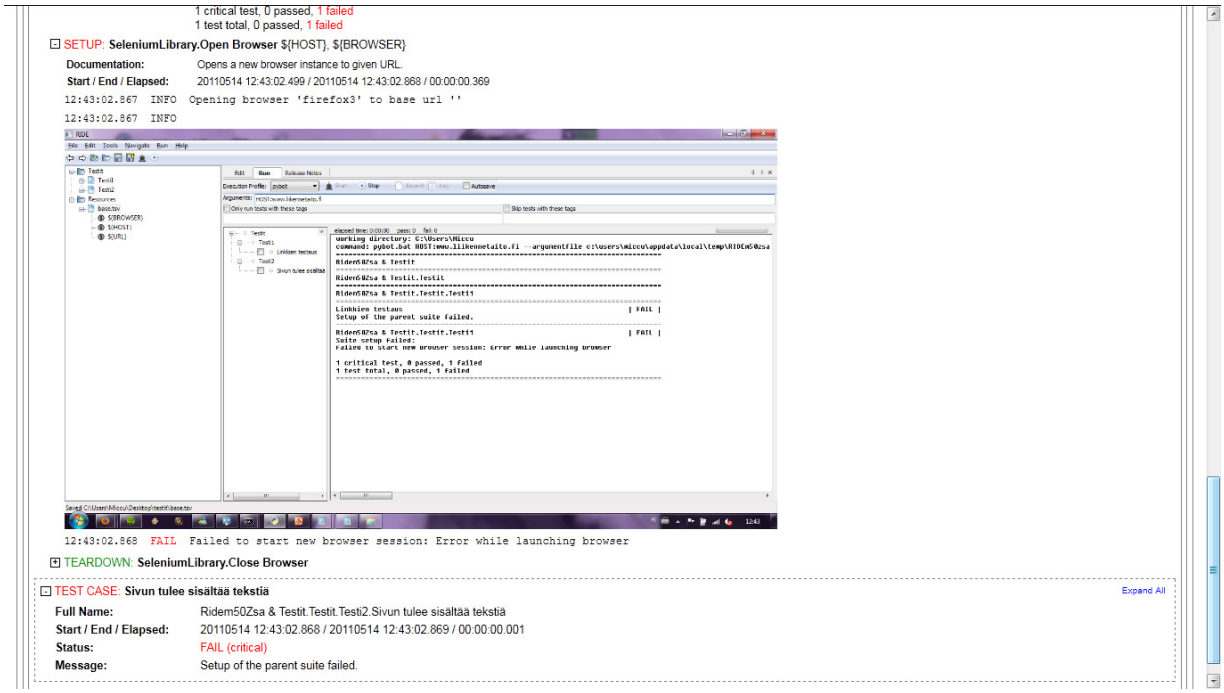
Robot Framework on rakennettu Pythonilla, joten tarvitaan Python sen asennukseen. Jos haluaa lisätä testikirjastoja Javaa käyttäen, myös Jython on pakollinen. Tutkimukseemme (luku 9) tarvitsimme Robot Frameworkin lisäksi SeleniumLibraryn, joka sisältää selaintestaukseen tarvittavat avainsanat, sekä itse RIDE-työkalun. RIDE tarvitsee Pythonin lisäksi pohjalle myös wxPython asennuksen. Näin ollen, jotta saa Robot Frameworkin toimimaan samoilla olosuhteilla kuin tutkimuskin on tehty, tarvitsee viisi eri asennusta. (Klarck 2011; InstallationInstructions 2011.)

9 Robot Frameworkin ja RIDE-työkalun heuristinen arviointi

Heuristiseen arviointiin käytimme Nielsenin listaa, joka sisältää kymmenen eri kohtaa. Ohjelma on arvioitu näiden kohtien kautta ja arvioinnin tulokset on koottu yhteen. Arviointi suoritettiin erikseen ja tämä on yhteenveto molempien arvioijien pohdinnoista.

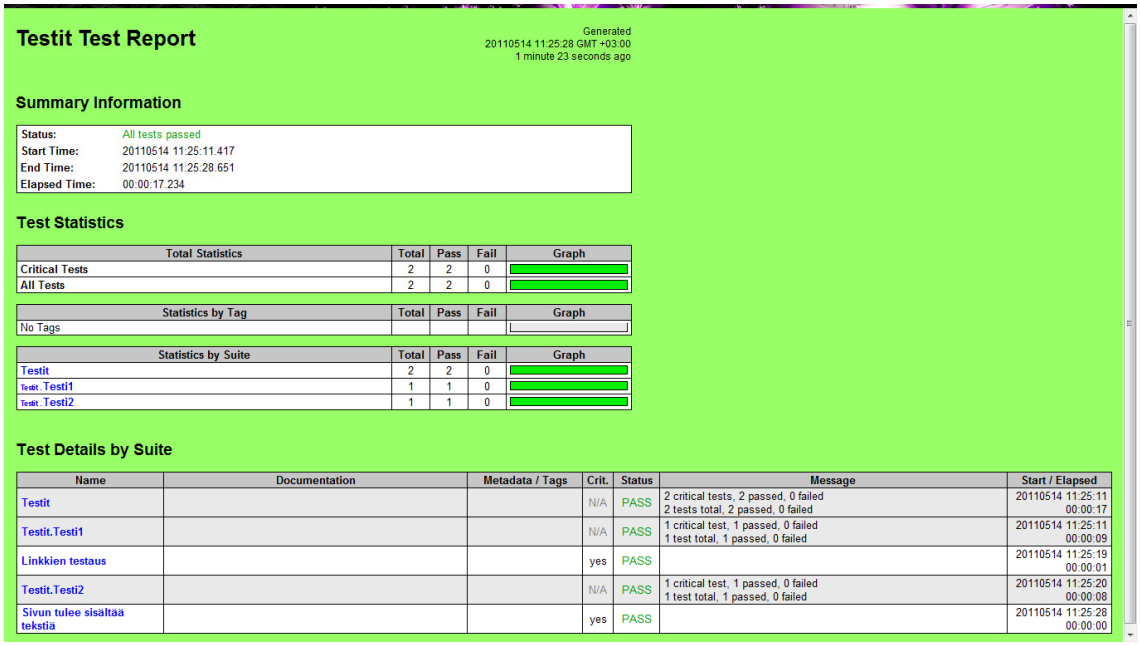
9.1 Yksinkertaisten ja luonnollisten dialogien käyttäminen

Jokaisen testiajon jälkeen tulee välitön palaute (pass läpäisylle ja fail epäonnistumiselle). Jos testi ei mene läpi, ohjelma myös kertoo mihin kohtaan testiä se on kaatunut. Jokaisesta testistä löytyy testinajon jälkeen raportit (kuva 2), joissa tarkemmin näytetään, mihin testi on kaatunut ja kuvakaappaus siitä, mihin kohtaan selain on jäänyt testin kaatuessa.



Kuva 2: Robotin kuvakaappaus.

Testeistä näkyy myös logit, joista näkyy kuinka, monta testiä kaikista testeistä on mennyt läpi ja kuinka monta on kaatunut. Kuva 3 havainnollistaa, miltä läpimenneen testiajon logi näyttää. Kuvassa 4 nähdään epäonnistunut testisarja, jonka toinen testitapaus on mennyt läpi ja toinen on kaatunut kesken ajon.



Testit Test Report

Generated
20110514 11:25:28 GMT +03:00
1 minute 23 seconds ago

Summary Information

Status:	All tests passed
Start Time:	20110514 11:25:11.417
End Time:	20110514 11:25:28.651
Elapsed Time:	00:00:17.234

Test Statistics

Total Statistics	Total	Pass	Fail	Graph
Critical Tests	2	2	0	
All Tests	2	2	0	

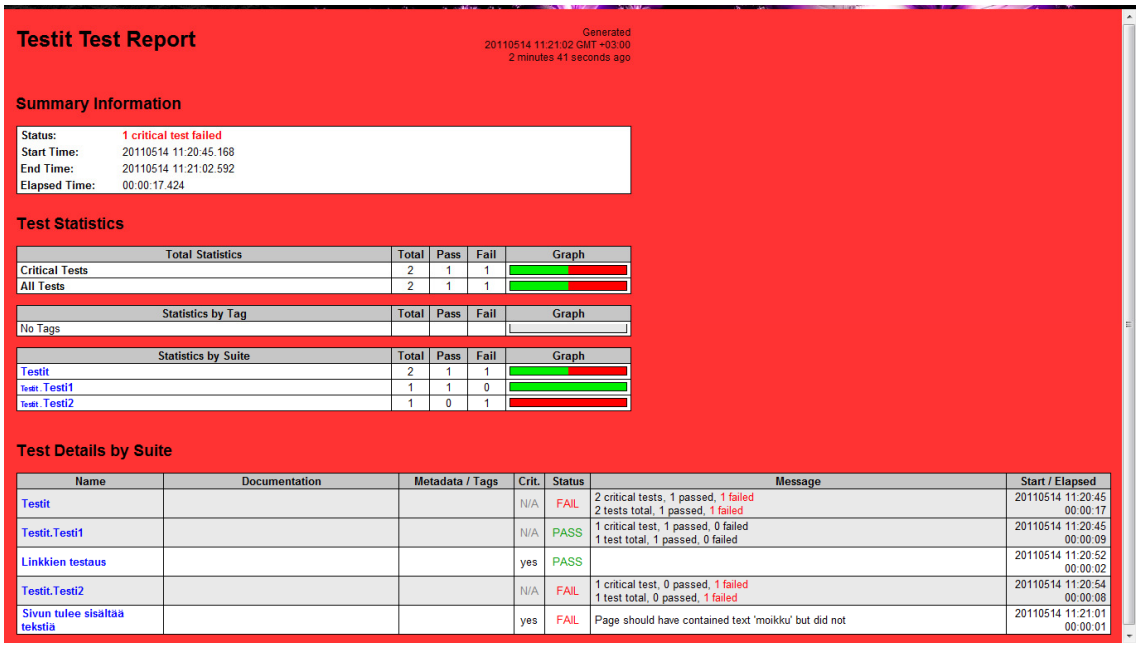
Statistics by Tag	Total	Pass	Fail	Graph
No Tags				

Statistics by Suite	Total	Pass	Fail	Graph
Testit	2	2	0	
Testit.Testi1	1	1	0	
Testit.Testi2	1	1	0	

Test Details by Suite

Name	Documentation	Metadata / Tags	Crit.	Status	Message	Start / Elapsed
Testit			N/A	PASS	2 critical tests, 2 passed, 0 failed 2 tests total, 2 passed, 0 failed	20110514 11:25:11 00:00:17
Testit.Testi1			N/A	PASS	1 critical test, 1 passed, 0 failed 1 test total, 1 passed, 0 failed	20110514 11:25:11 00:00:09
Linkkien testaus			yes	PASS		20110514 11:25:19 00:00:01
Testit.Testi2			N/A	PASS	1 critical test, 1 passed, 0 failed 1 test total, 1 passed, 0 failed	20110514 11:25:20 00:00:08
Sivun tulee sisältää tekstiä			yes	PASS		20110514 11:25:28 00:00:00

Kuva 3: Testiajo on mennyt läpi.



Testit Test Report

Generated
20110514 11:21:02 GMT +03:00
2 minutes 41 seconds ago

Summary Information

Status:	1 critical test failed
Start Time:	20110514 11:20:45.168
End Time:	20110514 11:21:02.592
Elapsed Time:	00:00:17.424

Test Statistics

Total Statistics	Total	Pass	Fail	Graph
Critical Tests	2	1	1	
All Tests	2	1	1	

Statistics by Tag	Total	Pass	Fail	Graph
No Tags				

Statistics by Suite	Total	Pass	Fail	Graph
Testit	2	1	1	
Testit.Testi1	1	1	0	
Testit.Testi2	1	0	1	

Test Details by Suite

Name	Documentation	Metadata / Tags	Crit.	Status	Message	Start / Elapsed
Testit			N/A	FAIL	2 critical tests, 1 passed, 1 failed 2 tests total, 1 passed, 1 failed	20110514 11:20:45 00:00:17
Testit.Testi1			N/A	PASS	1 critical test, 1 passed, 0 failed 1 test total, 1 passed, 0 failed	20110514 11:20:45 00:00:09
Linkkien testaus			yes	PASS		20110514 11:20:52 00:00:02
Testit.Testi2			N/A	FAIL	1 critical test, 0 passed, 1 failed 1 test total, 0 passed, 1 failed	20110514 11:20:54 00:00:08
Sivun tulee sisältää tekstiä			yes	FAIL	Page should have contained text 'moikku' but did not	20110514 11:21:01 00:00:01

Kuva 4: Testiajossa toinen testitapauksista ei ole mennyt läpi ja toinen on.

9.2 Käyttäjien oman kielen käyttäminen

Hyvin suunnitellussa testiautomaatioprosessissa käyttäjien tulisi tietää testauksen perustermit. Jos testiautomaatioprosessia ei ole suunniteltu etukäteen juuri ollenkaan, niin

käyttäjät eivät välttämättä tunne termejä entuudestaan. Termit ovat kuitenkin helposti selvitettävissä sekä suhteellisen yksinkertaisia.

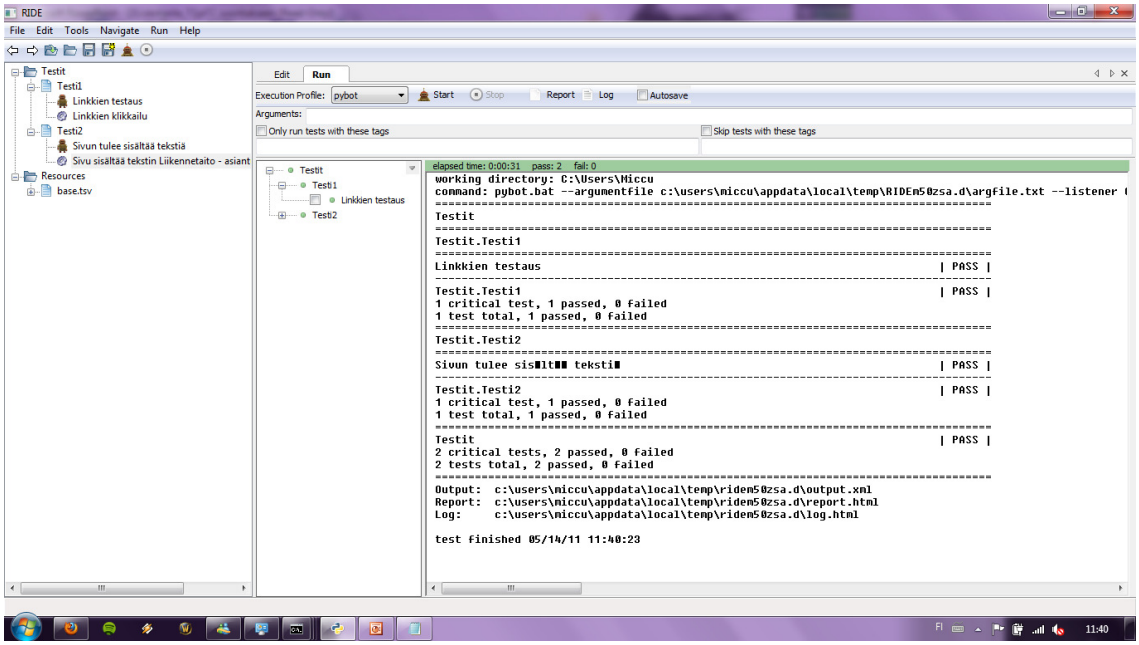
Robot Frameworkissä yleisimmin käytettäviä termejä ovat testisarja (test suite), testitapaus (test case) ja avainsana (keyword). Kyseiset termit ovat helposti ymmärrettäviä ja pääteltävissä, missä niitä käytetään.

9.3 Käyttäjien muistikuorman minimoiminen

Robot Framework kirjoittaa jokaisen login ja raportin oletusarvoisesti log.html ja reporti.html muotoon. Näin ollen uudet logit ja raportit tallentuvat vanhojen päälle. Kuitenkin ohjekirjaa tutkittaessa löytyy ratkaisu siihen, että jokainen logi ja raportti jäävät tallennetuiksi. Testiajon yhteydessä on mahdollista määritellä joko raporteille tai logeille omat nimet tai käyttää aikaleima-komentoa, joka lisää jokaisen raportin perään päivämäärän sekä kellonajan, jolloin testit on ajettu.

”Timestamping output files
All output files listed in this section can be automatically timestamped with the option --timestampoutputs (-T), which is one of the rare options taking no value. When this option is used, a timestamp in the format YYYYMMDD-hhmmss is placed between the extension and the basename of each file. The example below would, for example, create such output files as output-20080604-163225.xml and mylog-20080604-163225.html.
pybot --timestampoutputs --log mylog.html --report NONE tests.html”
(Robot Framework User Guide 2011.)

Testit on helppo ajaa uudestaan joko itse RIDE-ohjelmasta (kuva 5) tai komentoriviltä pybot-komennolla (kuva 6). Tähänkin löytyvät ohjeet Robot Frameworkin käyttöoppaasta.



Kuva 5: RIDE-ohjelmassa ajettu testiajo.

Kuva 6: Komentorivillä ajettu testiajo.

Jokaisen testitapauksen päätteeksi näytetään testiajon tulos, jonka avulla tiedetään, mikä testiajoista on mennyt läpi ja mikä ei. Tämän avulla on helpompaa ajaa vain kaatuneet testit uudelleen. Tämä havainnollistetaan kuvissa 5 ja 6.

Kun luot uuden testitapauksen tai resurssitiedoston etkä kirjoita siihen mitään tai liitä resurssitiedostoa mihinkään testitapaukseen sekä avaat uudestaan testit RIDE:n, niin tiedosto katoaa eikä sitä olekaan luotu mihinkään. Tämän voi kokea toisaalta hyvänä ja toisaalta

huonona asiana. Hyvää on se, että jos kirjoitat väärin testitapauksen nimen, voit vain luoda uuden testitapauksen ja aloittaa näin alusta ilman hankalaa uudelleen nimeämistä. Toisaalta, jos vahingossa ohjelma suljetaan ja olisikin haluttu säilyttää jo luotu tyhjä testitapaus, ei sitä löydykään enää mistään.

Avainsanoista on olemassa listaus Robot Frameworkin sivuilla, josta löytyy kaikkiin olemassa oleviin avainsanoihin tarvittavat attribuutit sekä niiden selitykset ja käyttötarkoitukset. Avainsanat ovat käyttötarkoituksiaan kuvaavia ja avainsanan tarkoitus on helposti pääteltävissä nimestä, kuten esimerkiksi Click Link, jonka avulla saadaan testitapaus klikkaamaan linkkiä. (SeleniumLibrary 2011.)

9.4 Käyttöliittymän yhdenmukaisuus

Testien ajaminen komentoriviltä on jokaisella koneella samanlaista käyttöliittymästä riippumatta komentorivin ollessa suhteellisen samanlainen käyttöliittymästä riippumatta. Jokaisella käyttöliittymällä toimii sama pybot-komento, joka ajaa testit. Testitapausten ajaminen on yhdenmukaista, ajettiin se sitten komentoriviltä tai RIDE:a käyttäen.

RIDE-työkalu on yhdenmukainen toimintojensa puolesta. Esimerkiksi jokainen testitapaus sekä resurssitiedosto luodaan samalla tavalla. Jokaista avainsanaa käytetään samalla tavalla. Ensin tulee itse avainsana, jonka jälkeen kerrotaan elementin sijainti sivustolla, esimerkiksi koodista löytyvän nimen tai arvon avulla. Tämän jälkeen yleensä laitetaan avainsanasta riippuen joko kenttään syötettävä arvo tai aika, jonka avulla pystytään määrittelemään, minkä ajan jälkeen testi kaatuu, jos tarvittavaa elementtiä ei löydy. Tämä havainnollistetaan taulukossa 1.

Input Text	userName	Matti
Wait Until Page Contains	Tervetuloa	30

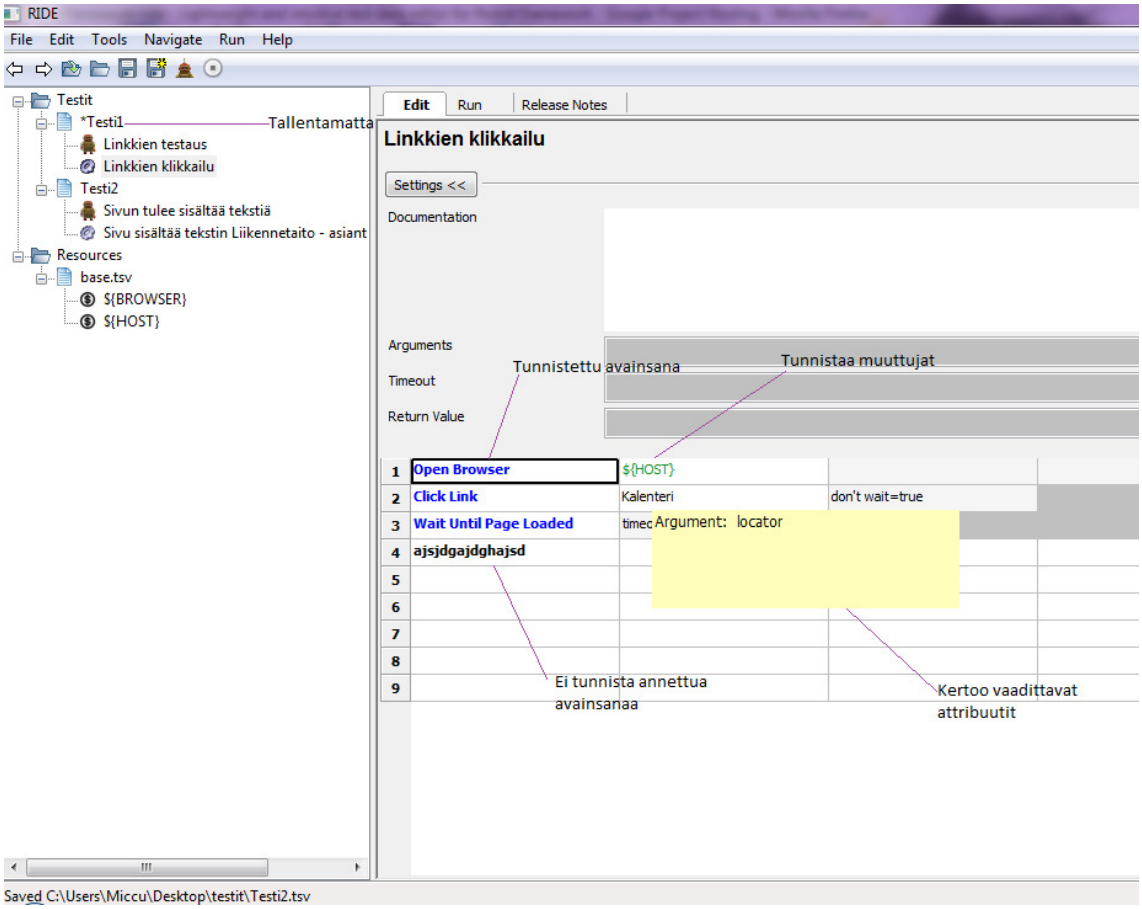
Taulukko 2: Esimerkki avainsanoista.

Robot Frameworkissa on myös mahdollista käyttää ylemmän tason avainsanoja, jotka helpottavat testitapausten lukemista. Esimerkiksi kuvassa 7 on ylemmän tason avainsana “Klikkaa linkkiä, odota sivun latautumista ja tekstin ilmestymistä”, joka on jaoteltu kuvan alareunassa alempiin avainsanoihin.

Kuva 7: Ylemmän tason avainsana.

9.5 Palautteen anto käyttäjälle

RIDE-ohjelma kertoo väreillä käyttäjälle tunnistetuista avainsanoista (kuva 8). Musta väri kertoo, että RIDE-ohjelma ei tunnista annettua avainsanaa. Sininen väri taas on merkki avainsanan tunnistuksesta. Vihreällä merkitty ${HOST} kertoo, että RIDE-ohjelma tunnistaa sen kirjoitusmuodon, mutta ei välttämättä tiedä mihin se viittaa. Jos resurssitiedostoa, jossa ${HOST}:n määritelmä sijaitsee, ei ole liitetty testitapaukseen, testitapausta ajettaessa testi kaatuu tunnistamattomuuteen.



Kuva 8: RIDE:n toimintoja.

Tähti testitapauksen edessä kertoo, että testiä ei ole tallennettu. Hiiren vieminen eri solujen päälle antaa tietoja soluista ja niihin tarvittavista attribuuteista. Tämä helpottaa käyttäjää muistamaan, mitä mihinkin kenttään tulee syöttää.

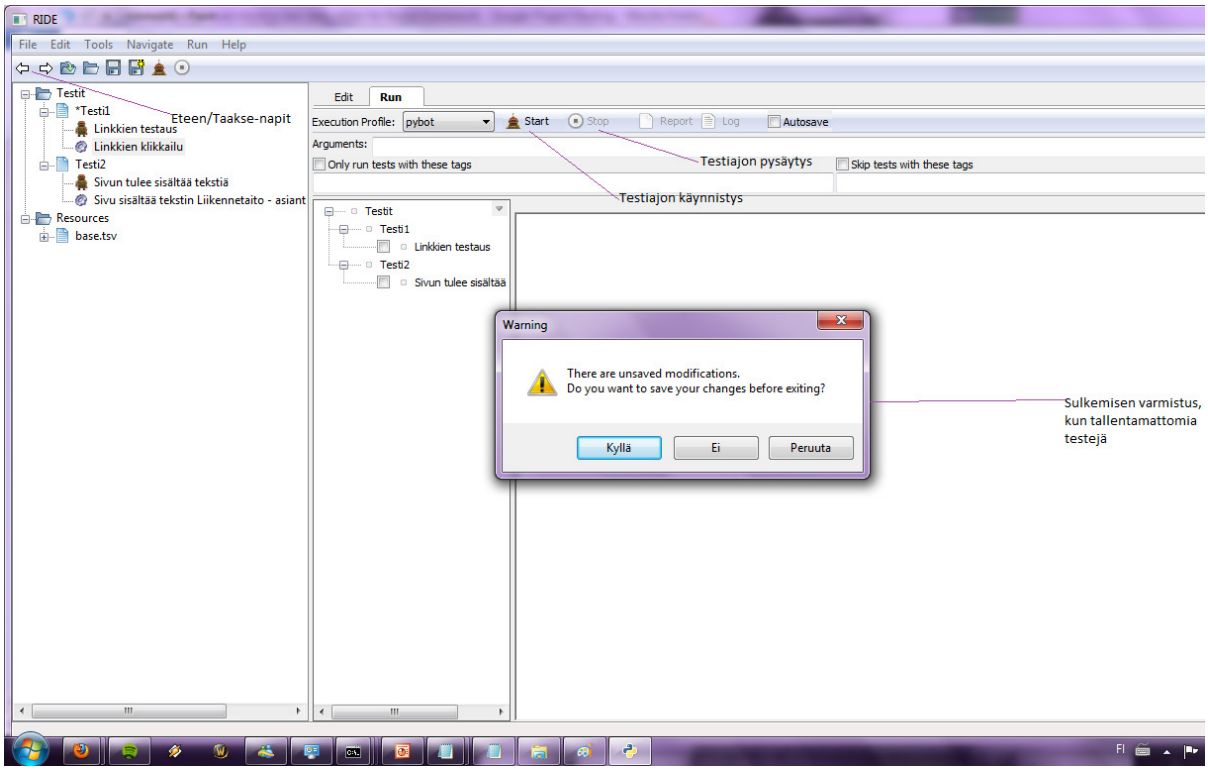
Testin ajon jälkeen ohjelma tai komentorivi, riippuen kummalla testit on ajettu, kertoo testin onnistumisesta sekä sen kuinka monta testiä ajettiin, kuinka monta ajetuista meni läpi ja kuinka moni epäonnistui. Jos testit epäonnistuvat, ohjelma antaa palautetta mihin ne kaatuivat. Kuvat 4 ja 5 näyttävät hyvin testiajojen lopputulokset.

Kun ajetaan koko testikansio, jokaisen testitapauksen ajon jälkeen ohjelma kertoo menikö testitapaus läpi vai ei. Kun kaikki kansiossa olevat testit on ajettu, kertoo ohjelma vielä loppuyhteenvedon onnistuneista ja epäonnistuneista testeistä. Testikansion ajon jälkeen on myös nähtävillä logit ja raportit, joista voi katsoa tarkemmat testiajojen tiedot.

9.6 Selkeät poistumistavat eri tiloista ja toiminnoista

Eteen- ja taaksepäin-napit (kuva 9) mahdollistavat RIDE:ssä liikkumisen testitapauksesta toiseen sen mukaan missä on viimeksi käynyt tai mihin on seuraavaksi mennyt. Tämä on kätevää silloin, kun testitapauksia on paljon ja vahingossa painaa jotain testitapausta, vaikka ei olisi pitänyt.

Testiajolle on Ridessä omat käynnistys- ja pysäytysnapit (kuva 9). Käynnistämisen jälkeen voi testitapausten ajon lopettaa missä vaiheessa tahansa pysäytysnapista. Komentorivissä toimii komento CTRL+C, joka on komentorivin oma keskeytystoiminto.

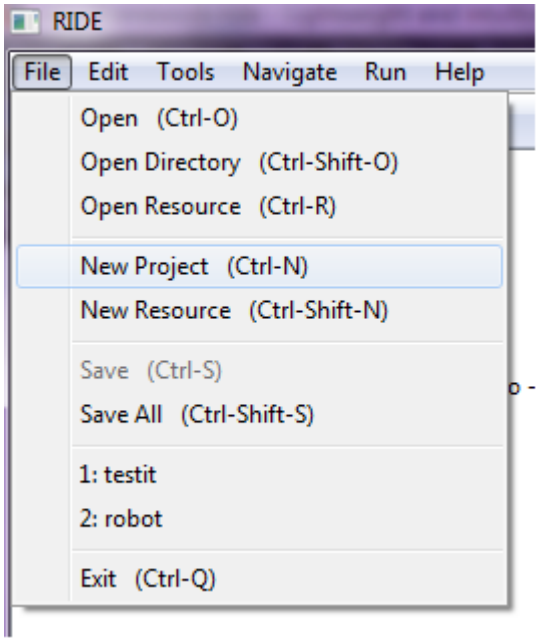


Kuva 9: Kuvassa näkyy RIDE:n eri toiminnallisuuksia.

Jos yrittää tallentamatta sulkea RIDE:a, ohjelma varmistaa haluatko varmasti sulkea ohjelman tallentamatta muutoksia. Ohjelmasta pääsee poistumaan joko painamalla ruksia oikeasta yläkulmasta, valitsemall File-valikosta Exit tai pikanäppäinyhdistelmällä CTRL+Q.

9.7 Oikopolut

RIDE:n lähes jokaiselle toiminnolle löytyy oma näppäinydistelmänsä, joka löytyy valikosta valittaessa toiminnon nimen yhteydestä, esimerkiksi New Project CTRL+N. Tämä havainnollistetaan kuvassa 10.

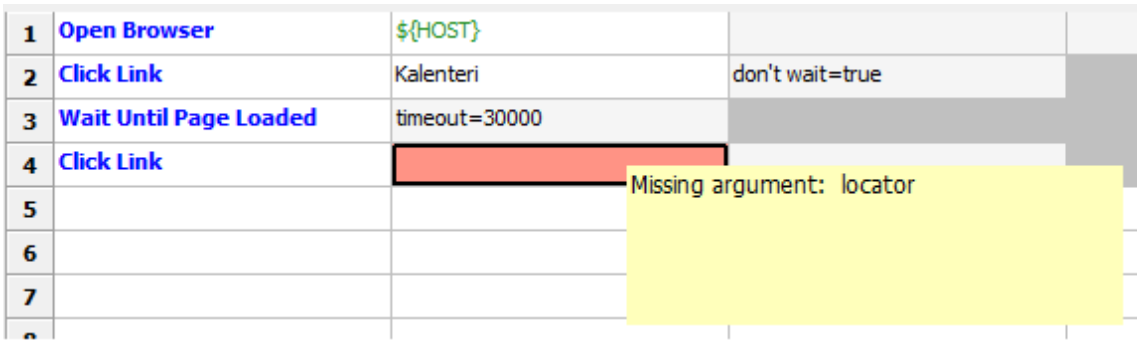


Kuva 10: Pikanäppäinyhdistelmiä

9.8 Virhetilanteiden selkeät virheilmoitukset ja niiden välttäminen

Virheilmoitukset ovat englanniksi. Jos ohjelma ei suostu avautumaan jonkun lisäosan puuttumisen takia, kerrotaan tämä ohjelmaa avattaessa komentoriviltä. Jos testitapauksista puuttuu jotain oleellista, testiä ajettaessa ohjelma kertoo testin kaatuneen sekä syyn kaatumiseen.

Avainsanoja kirjoitettaessa RIDE:n se näyttää solun punaisella (kuva 11), jos vaadittava argumentti puuttuu. Vietäessä hiiri solun päälle ohjelma kertoo, että solusta puuttuu argumentti ja kertoo, mikä argumentti siihen kuuluu.



Kuva 11: Puuttuva argumentti.

Kokemusten perusteella ei ole tullut ohjelmaan liittyviä virhetilanteita. Sivuilta (Release Notes 0.34 2011) myös näkyy, että bugikorjauksia tehdään kun niitä ilmenee.

9.9 Riittävä ja selkeä apu sekä dokumentaatio

Robot Frameworkin käytostä löytyy käyttöopas (Robot Framework User Guide 2011) internetistä. Käyttöopas sisältää tarkat ohjeet testitapausten luomisesta, testien ajamisesta sekä laajempia käyttöohjeita, kuten kuinka luoda omia testikirjastoja ja käyttää Robot Frameworkia Javasta.

Lisäksi itse testieditointiohjelmasta, eli RIDE:sta, löytyy oma käyttöopas (HowTo 2011). RIDE:n käyttöopas on ehkä turhan suppea ainakin RIDE:n oman ajotoiminnon suhteen. Kummassakin käyttöoppaassa kerrotaan kuinka luoda testitapauksia ja kuinka niitä voidaan editoida. Robot Framework käyttöoppaasta löytyvät ohjeet kuinka testitapauksia voidaan ajaa.

10 Havaitut ongelmat

Robot Framework on suurimmilta osin hyvin tehty eikä ongelmia havaittu kuin vain muutamista asioista. Ongelmat parannusehdotuksineen on listattu alle.

Ongelma 1: Muuttujia voi uudelleen nimetä erillisiin resurssitiedostoihin testitapausten luettavuuden parantamiseksi. Nämä muuttujat kirjoitetaan dollarin ja hakasulkujen väliin, esimerkiksi ${HOST}. Lisättäessä muuttujaa testitapaukseen, muuttuja muuttuu vihreäksi. Tämä tapahtuu myös silloin, kun testin resurssitiedostoa ei ole liitetty testitapaukseen. Tämä siis tarkoittaa sitä, että testitapaus ei kuitenkaan tunnista muuttujaa, mutta silti se on vihreällä. Tämä saattaa olla harhaanjohtavaa, koska yleensä vihreä väri tarkoittaa onnistumista.

Ongelma esiintyy usein, jos käyttäjä toistuvasti unohtaa liittää resurssitiedostot testitapauksiin. Ongelmatilanteesta on kuitenkin helppo selviytyä, koska Robot Framework kertoo aina miksi testit ovat ajon aikana kaatuneet. Ongelma on helposti ohitettavissa, jos käyttäjä muistaa aina lisätä resurssitiedostot testitapauksiin. Ongelma ei tee ohjelmasta markkinoinnin kannalta huonompaa. Ongelman vakavuusluokka on 1. Vakavuusluokat on esitelty kappaleessa 3.3 “Heuristisen arvioinnin vakavuusluokitus”.

Ongelma 2: Jos testin ajaa RIDE:n omalla ajotoiminnolla ja testin ajon lopettaa kesken, saattaa se jostain syystä jäädä “päälle” ja avata koko ajan uusia selainikkunoita. Tämä viittaisi siihen, että testiajo jää jotenkin jumiin, kun sen lopettaa kesken.

Ongelma esiintyi suoritettujen testiajojen aikana joka kerta kun testiajon pysäytti kesken. Ongelmatilanteesta on periaatteessa helppo selviytyä, koska ohjelma myös itse sulkee avaamansa selainikkunat, mutta koskaan ei voi tietää kuinka kauan tai kuinka monta ikkunaa se avaa. Ongelman voi ohittaa niin, että ei keskeytä testiajoa, mutta testiajon keskeytystilanteessa ongelmaa ei voida ohittaa. Ongelmalla ei ole suurempia vaikutuksia markkinoinnin kannalta. Kuitenkin testejä pystyy ajamaan myös komentoriviltä, jolloin tätä ongelmaa ei esiinny. Ongelman vakavuusluokka on 2.

11 Parannusehdotukset

Heuristisessa arvioinnissa ei yleensä oteta kantaa ongelmien korjaamiseen, mutta tässä tapauksessa esittelemme muutamia parannusehdotuksia. Parannusehdotukset ovat suuntaa antavia ja käytettävyyttä parantavia.

Parannusehdotus 1: Jos resurssitiedostoa ei ole liitetty testitapaukseen, voisi muuttujien värit, joita se ei tunnista olla esimerkiksi harmaina, jotta käyttäjä huomaa heti jonkun olevan vialla.

Parannusehdotus 2: RIDE:n ohjeisiin voisi lisätä ohjeistuksen siitä, kuinka RIDE:llä ajetaan testitapaukset ja mitä eri vaihtoehtoja ajamiselle on. RIDE:ssä vaihtoehtoina ovat jybot (Jython-pohjainen ajaminen), pybot (Python-pohjainen ajaminen) sekä custom script, johon voi luoda oman skriptin.

12 Tulokset

Heuristisen arvioinnin tuloksena löytyi muutama ongelma. Ongelma 1 oli resurssitiedoston liittämisen unohtamiseen liittyvä ongelma, jossa muuttujan väri oli harhaanjohtava. Ongelma 2 taas liittyi RIDE:n ajo-ominaisuuteen, jonka keskeytys aiheutti selainikkunan toistuvaa avautumista.

Ongelmaan numero yksi löytyi ratkaisu väriä muuttamalla, kun resurssitiedostoa ei ole liitetty testitapaukseen. Ongelmaan numero kaksi ei löytynyt selvää ratkaisua ja tämä tulisi ratkaista ohjelman koodia tutkimalla.

Tutkittu ohjelma on kuitenkin käytettävyydeltään hyvä sekä helppo oppia. Ohjelman ollessa open source voi kuka tahansa saada sen ilmaiseksi käyttöönsä sekä myös lähettää omia parannusehdotuksiaan tekijöille.

13 Johtopäätökset

Heuristinen arviointi on helppo ja yksinkertainen tapa toteuttaa käytettävyyden arviointi, koska sen voi tehdä kuka tahansa milloin tahansa. Nielsenin versio on tunnetuin ja käytetyin heuristisesta arvioinnista, koska siinä on hyvin ja ytimekkäästi tiivistetty tärkeimmät asiat käytettävyyden arvioinnissa kymmeneen eri kohtaan.

Vaikka heuristisen arvioinnin voi suorittaa kuka tahansa, parhaimmat tulokset saadaan, kun käytetään 3-5 arvioijaa, jotka ovat käytettävyyden ammattilaisia tai joilla on aikaisempaa kokemusta käytettävyyden arvioinnista. Heuristinen arviointi on käytettävyystestauksen lisäksi yleisin käytettävyyden tutkimistapa, koska se on nopea ja halpa suorittaa. Normaalisti arviointi kestää tunnista pariin tuntiin ja toteutetaan yksilötyönä. Arvioinnin jälkeen keskustellaan muiden arvioijien kanssa tuloksista ja kasataan loppuraportti.

Käytettävyyden testauksella saadaan tärkeää palautetta tuotteen kehittäjälle. Käytettävyystestien tarkoitus on tehdä ohjelman käyttölaadusta parempi. Käytettävyystestit ovat monipuolisia ja niitä voidaan käyttää minkä tahansa testaukseen. Käytettävyystestejä tehdään kahteen eri tarkoitukseen riippuen siitä missä vaiheessa sovelluskehitystä ollaan. Käytettävyystesteillä etsitään muun muassa bugeja, jolloin käytettävyystestit tehdään sovelluskehityksen aikana, tai niillä voidaan varmistaa tuotteen kokonaistoimivuus ennen tuotteen julkaisemista.

Testausta on monenlaista. Se jakautuu testimetodien ja testitapojen perusteella eri alaluokkiin. Testaus voidaan jakaa myös manuaalitestaukseen, joka suoritetaan käsin, ja automaatiotestaukseen, jonka kone suorittaa. Manuaalitestaus ja automaatiotestaus tukevat toisiaan testausprosessissa. Automaatio ei vähennä manuaalitestauksen tarvetta vaan helpottaa testaajia keskittymään hankalampiin ja vain yhden kerran toistettaviin testitapauksiin, kun automaatiotestaus hoitaa usein toistettavat testitapaukset.

Automaatiotestauksessa tarvitaan myös ihmisten työtä. Testitapausten tulokset tulee arvioida ja verrata odotettuihin testituloksiin. Testauksen suunnittelu onkin ensiarvoisen tärkeää, jotta voidaan määritellä, mihin testausta käytetään ja mitkä testit kannattaa automatisoida.

Automaatiotestaukseen liittyy paljon myyttejä, jotka tulisi erotella faktoista ennen testiautomaation mukaan ottamista. Työkalujen valinta on ensiarvoisen tärkeää, jotta ei osteta monilla tuhansilla euroilla lisenssejä ja sitten huomata, että työkalu ei olekaan sopiva juuri oman ohjelman testaukseen. Automaatiotestaus on tärkeä osa testausta, jos se osataan toteuttaa oikein. Se tuo lisäarvoa sillä, että pystytään suorittamaan testejä, joita ei käsin voida suorittaa.

Robot Framework on helppokäyttöinen ja helppo oppia. Se on selaintestaukseen hyvä valinta, koska sen toimivuus on hyvä, eikä bugeja juurikaan löydy. Bugit korjataan ja niistä kirjoitetaan julkaisutietoihin. Ohjelma on open source, eikä näin ollen maksa mitään.

Lähteet

Alam, M.N. Software Test Automation Myths and Facts. Viitattu 22.5.2011. http://www.benchmarkqa.com/pdf/papers_automation_myths.pdf

Alliance Global Services. Guidelines to create a robust test automation framework. Viitattu 22.5.2011. http://info.allianceglobalservices.com/Portals/30827/docs/test%20automation%20framework%20and%20guidelines.pdf

Bach, J. 1999. Test Automation Snake Oil. Viitattu 22.5.2011. http://www.satisfice.com/articles/test_automation_snake_oil.pdf

Beall, A. 2008. Why Automation Projects Fail (and what you can do to prevent failure). Viitattu 22.5.2011. http://martproservice.com/Why_Software_Projects_Fail.pdf

Blokdijk, G & Menken, I. 2008. Software Testing and Quality Assurance with It Change Management Transition Planning, Support, Service Validation, Testing and Evaluation Handbook. Change without Risk. Viitattu 22.5.2011. http://books.google.com/books?id=uektVYkbk6kC&printsec=frontcover&dq=Gerard+Blokdijk,Ivanka+Menken+software+testing&hl=fi&ei=LxDZTZ-LOMqVswbepJD4Ag&sa=X&oi=book_result&ct=result&resnum=1&ved=0CC4Q6AEwAA#v=onepage&q&f=false

Clifton, M. 2003. Advanced Unit Testing, Part I – Overview. Viitattu 22.5.2011. http://www.codeproject.com/KB/cs/autp1.aspx#Black%20Box%20vs.%20White%20Box%20Test3

Etnoteam. 2010. Käytettävyystesti mittaa käytettävyyttä ja välittää käyttäjän ääntä. Viitattu 305.2011. http://www.etnoteam.fi/koti/artikkelit/kaytettavyystesti-mittaa-kaytettavyytta-ja-valittaa-kayttajan-aanta/

Eulenberger-Karvetti, K. 2005. Linssintarkastusjärjestelmän käyttöliittymän käytettävyyden arviointi. Viitattu 30.5.2011 http://www.netlab.tkk.fi/opetus/s38310/04-05/Kalvot_04-05/Dty%F6_seminaari_eulenberger010305-1.pdf

Fewster, M. & Graham, D. 1999. Software Test Automation: Effective use of test execution tools. Harlow: Addison-Wesley

Godase, S. 2003. An Introduction to Software Test Automation. Viitattu 15.5.2011 http://www.indicthreads.com/1329/an-introduction-to-software-test-automation/

Hinz, J & Gijsen, M. Fifth Generation Scriptless and Advanced Test Automation Technologies. Viitattu 22.5.2011. http://www.testars.com/docs/5GTA.pdf

Hoffman, D. 1999. Test Automation Architectures: Planning for Test Automation. Viitattu 22.5.2011. http://www.softwarequalitymethods.com/Papers/autoarch.PDF

HowTo. 2011. Viitattu 23.5.2011. http://code.google.com/p/robotframework-ride/wiki/HowTo

InstallationInstructions. 2011. Viitattu 21.5.2011. http://code.google.com/p/robotframework-ride/wiki/InstallationInstructions

Kelly, D. 1997. Software Test Automation and the Product Life Cycle. Viitattu 22.5.2011. http://www.mactech.com/articles/mactech/Vol.13/13.10/SoftwareTestAutomation/

Klarck, P. 2011. Installation. Viitattu 21.5.2011. http://code.google.com/p/robotframework/wiki/Installation

Kuutti, W. 2003. Käytettävyys, suunnittelu ja arviointi. Saarijärvi: Gummerus Kirjapaino Oy

Käyttötuotteen heuristinen arviointi. Viitattu 30.5.2011 http://www.mlab.uiah.fi/polut/Design/tyokalu_heuristinen_arvio.html

Larman, G & Vodde, B. 2010. Acceptance Test-Driven Development with Robot Framework. Viitattu 22.5.2011. http://code.google.com/p/robotframework/wiki/ATDDWithRobotFrameworkArticle

Marick, B. 1998. When Should a Test Be Automated?. Viitattu 22.5.2011. http://www.exampler.com/testing-com/writings/automate.pdf

Mielonen, S & Hintikka, K. A. 1998. Heuristisen arvioinnin muistilista – pitkä versio. Viitattu 21.5.2011. http://www.uiah.fi/mediastudio/survey4/liitea1.html

Nielsen, J. 1993. Usability engineering. Boston (MA): Academic Press, cop

Ovaska, S. 2004. Käytettävyyden perusteet. Viitattu 30.5.2011
http://www.cs.uta.fi/~ov/usab/hlinna/luennot/1-johdanto.pdf

Pan, J. 1999. Software Testing. Viitattu 19.5.2011.
http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/

Python Programming Language – Official Website. 1990-2011. Viitattu 21.5.2011.
http://www.python.org/

Robot Framework. 2011. Viitattu 23.5.2011. http://code.google.com/p/robotframework/

robotframework-ride. 2011. Viitattu 21.5.2011.
http://code.google.com/p/robotframework-ride/

Robot Framework User Guide Version 2.5.7. 2011. Viitattu 21.5.2011.
http://robotframework.googlecode.com/hg/doc/userguide/RobotFrameworkUserGuide.html?r=2.5.7

Rubin, J. & Chisnell, D. 2008. Handbook of usability testing. Indianapolis: Wiley publishing, Inc

SeleniumLibrary. 2011. Viitattu 21.5.2011.
http://code.google.com/p/robotframework-seleniumlibrary/

SeleniumLibrary. 2011. Viitattu 21.5.2011.
http://robotframework-seleniumlibrary.googlecode.com/hg/doc/SeleniumLibrary.html?r=2.7

Sinkkonen, I. 2002. Mikä on käytettävyystesti. Viitattu 23.5.2011.
http://www.adage.fi/blogi/2002/mika-on-kaytettavyystesti/

Sinkkonen, I., Kuoppala, H., Parkkinen, J. & Vastamäki, R. 2006. Käytettävyyden psykologia. 3. painos. Helsinki: Edita Prima Oy

Wiio, A. 2004. Käyttäjäystävällisen sovelluksen suunnittelu. Helsinki: Edita Prima Oy

Kuvat

Kuviot

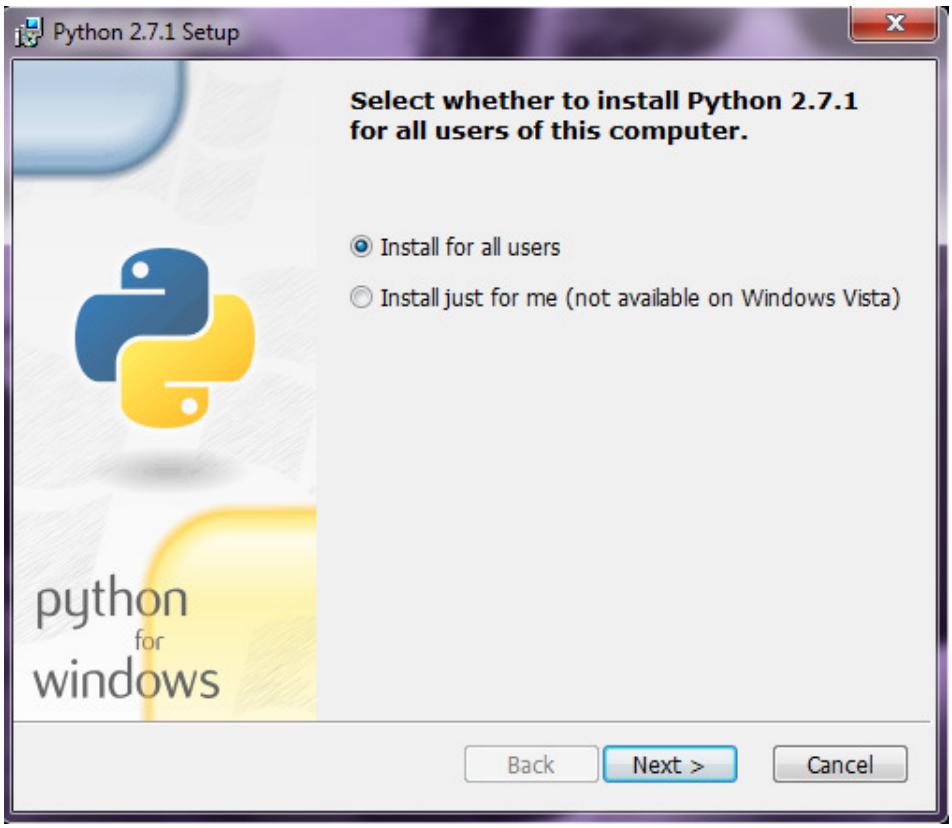
Taulukot

Liitteet

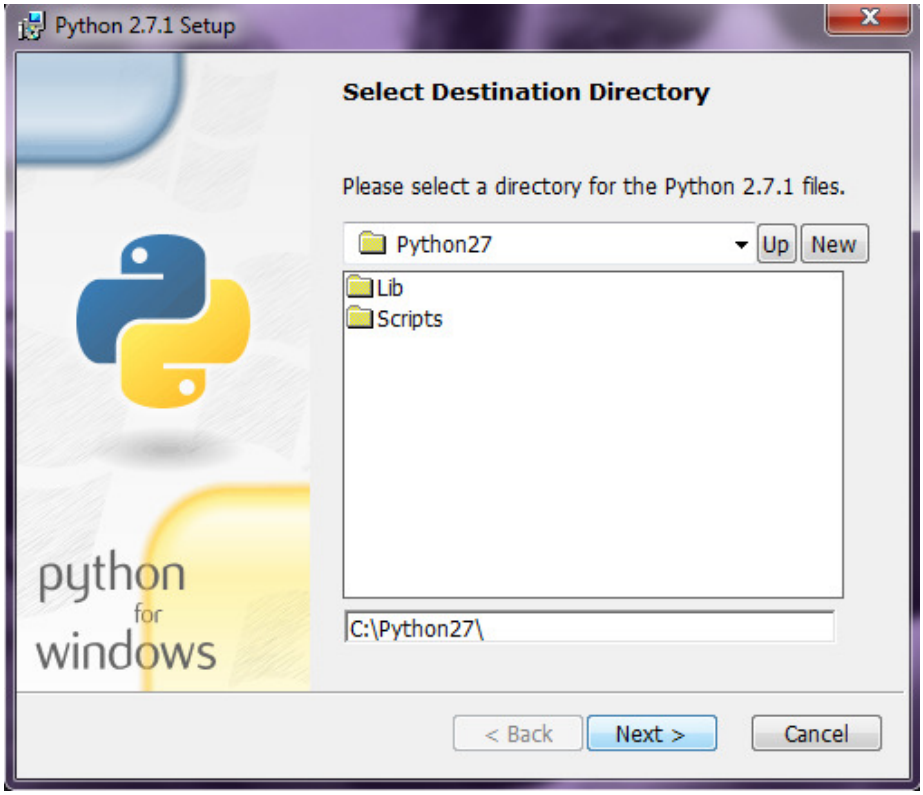
Liite 1: Pythonin asennus

Robot Framework tarvitsee asennuspohjaksi Pythonin, joka on ohjelmointikieli. Pythonin asennus on helppoa ja nopeaa. Asennusprosessi voidaan toteuttaa pelkillä oletusarvoilla. (Python Software Foundation 1990-2011.)

Pythonin asennus tapahtuu seuraavanlaisesti.

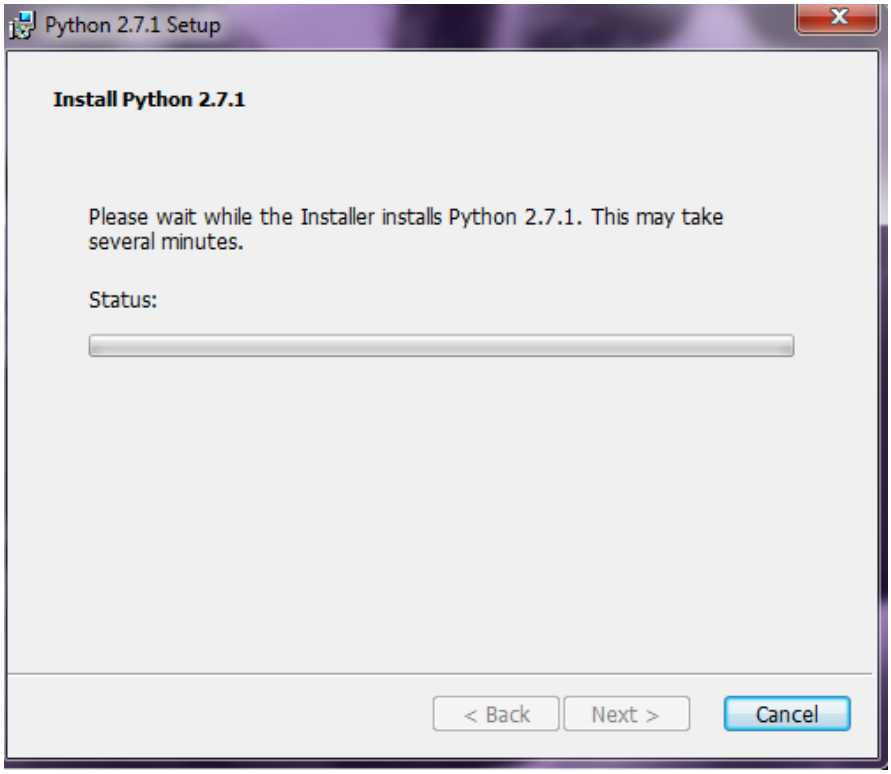


Kuva 12: Asennuksen aloitus.



Kuva 13: Valitaan asennuspaikka.

Kuva 14: Valitaan mitä ominaisuuksia halutaan asentaa.



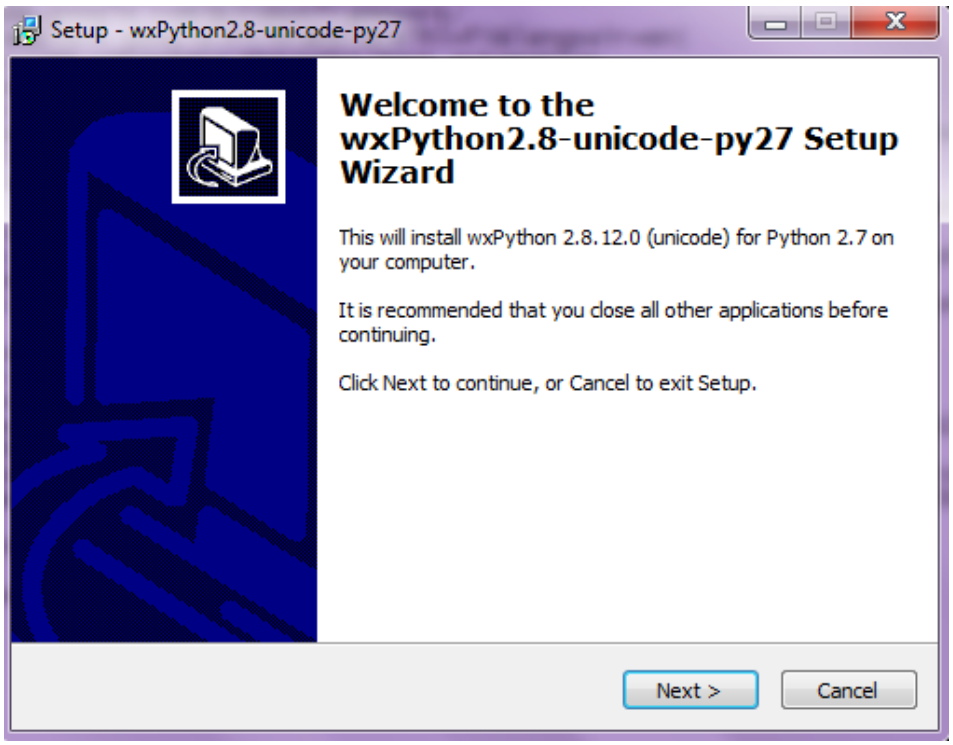
Kuva 15: Asennus suoritetaan.

Kuva 16: Asennus on suoritettu.

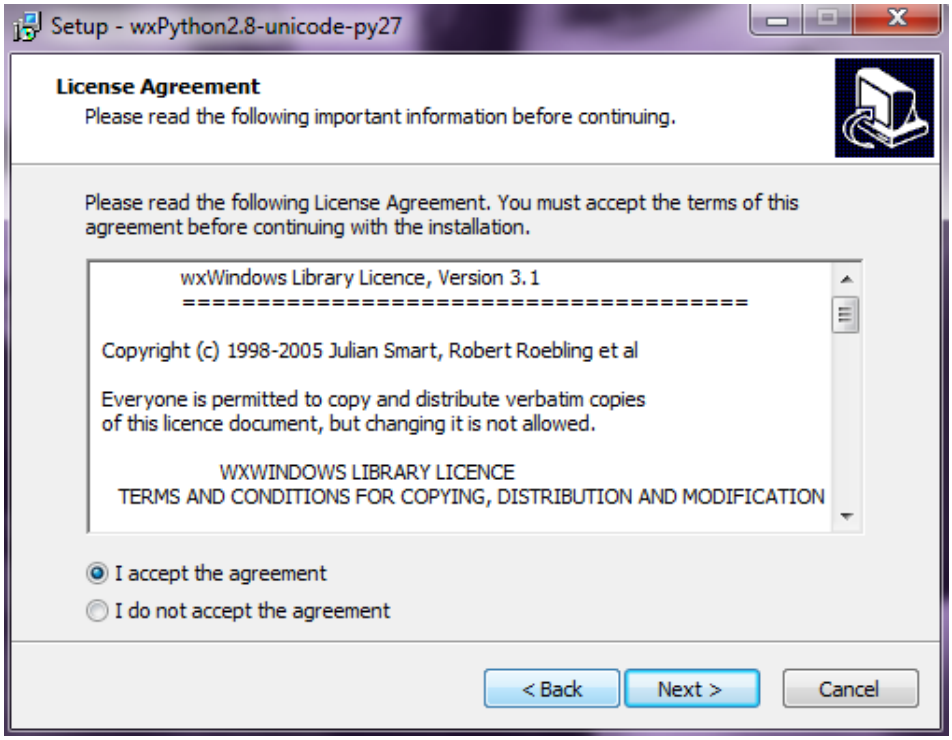
Liite 2: wxPythonin asennus

Jos halutaan asentaa myös RIDE, täytyy Pythonin lisäksi pohjalle ottaa wxPython, joka on Pythoniin lisättävä työkalupaketti. (InstallationInstructions 2011.)

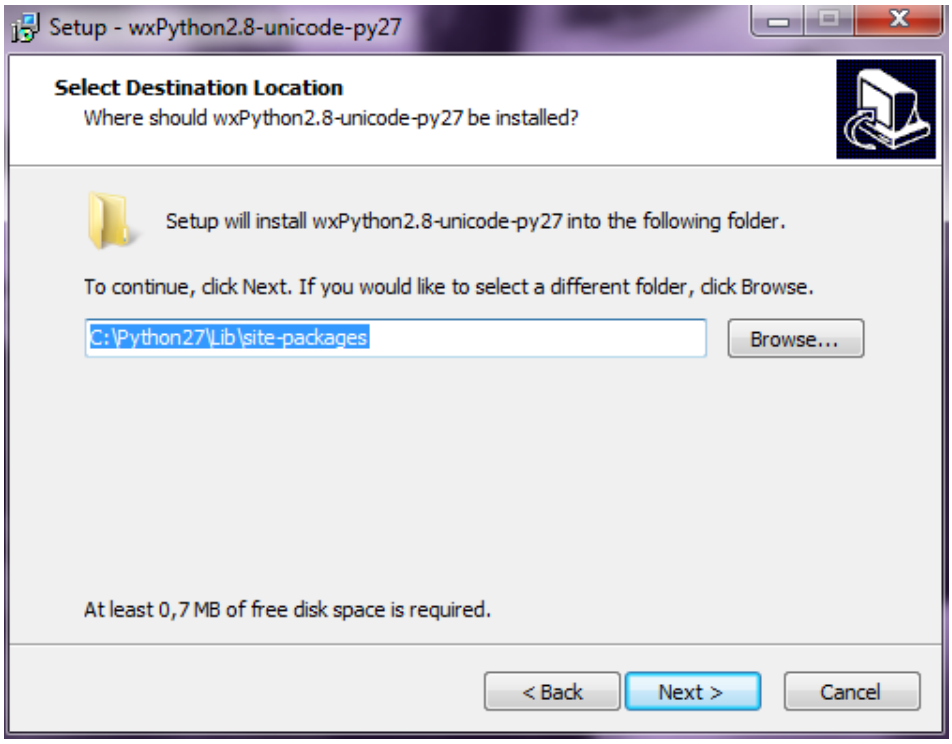
wxPythonin asennus tapahtuu seuraavanlaisesti.



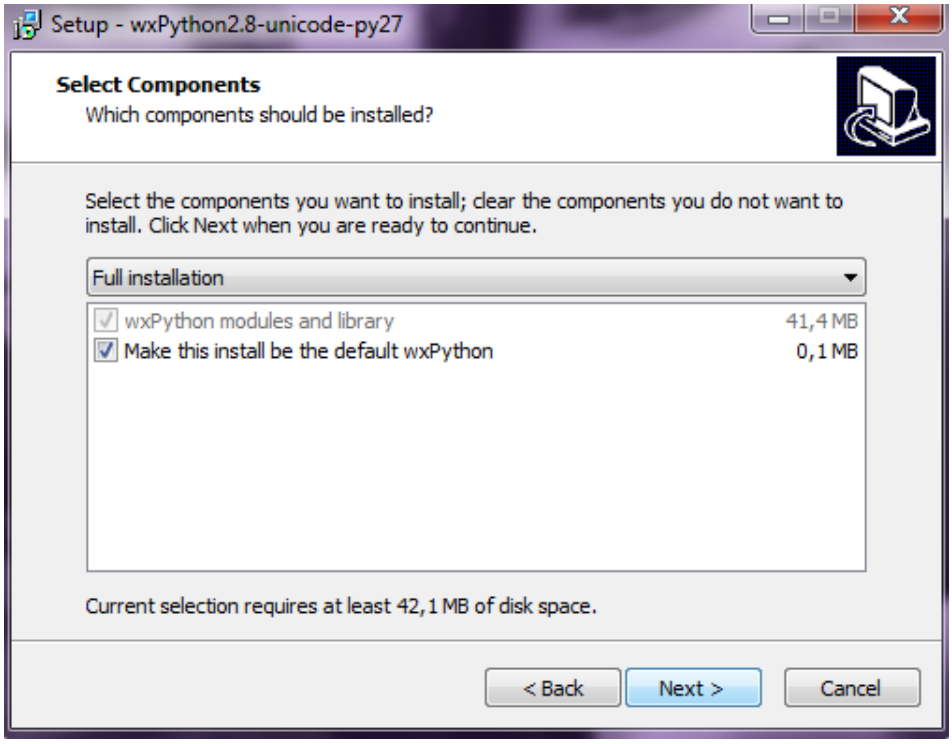
Kuva 17: wxPythonin asentamisen aloittaminen.



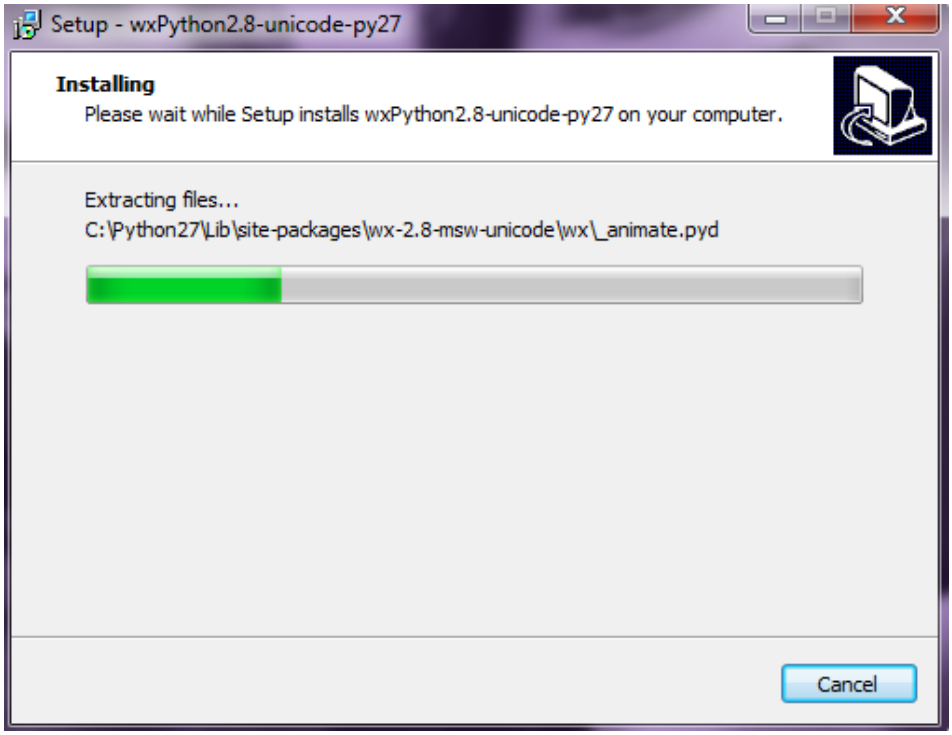
Kuva 18: Hyväksytään lisenssiehdot.



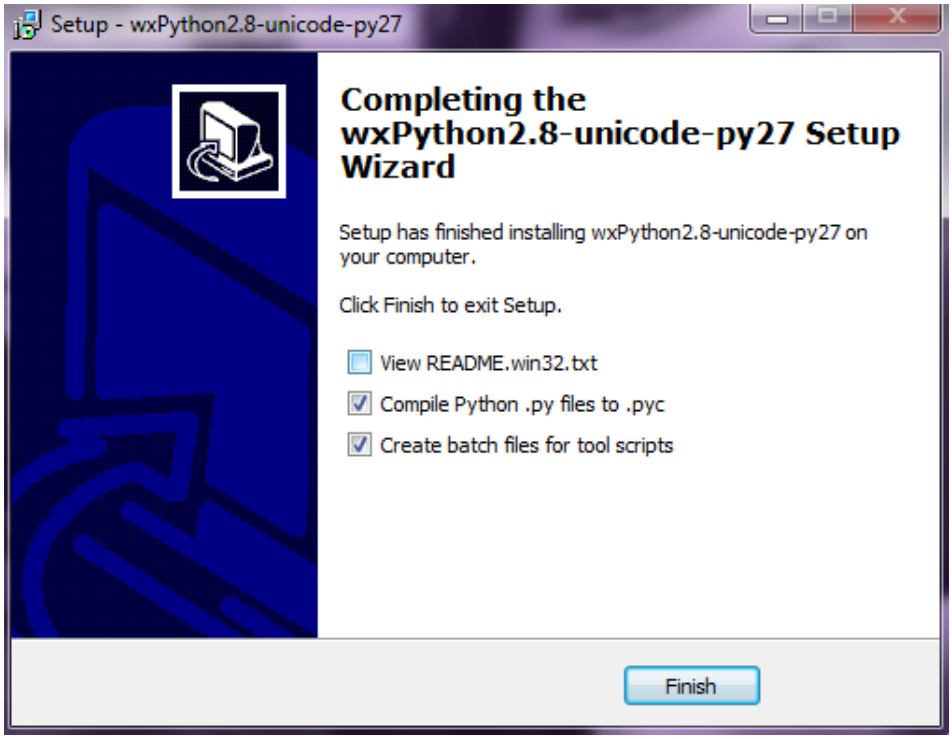
Kuva 19: Asennus tulee sijoittaa Pythonin alle.



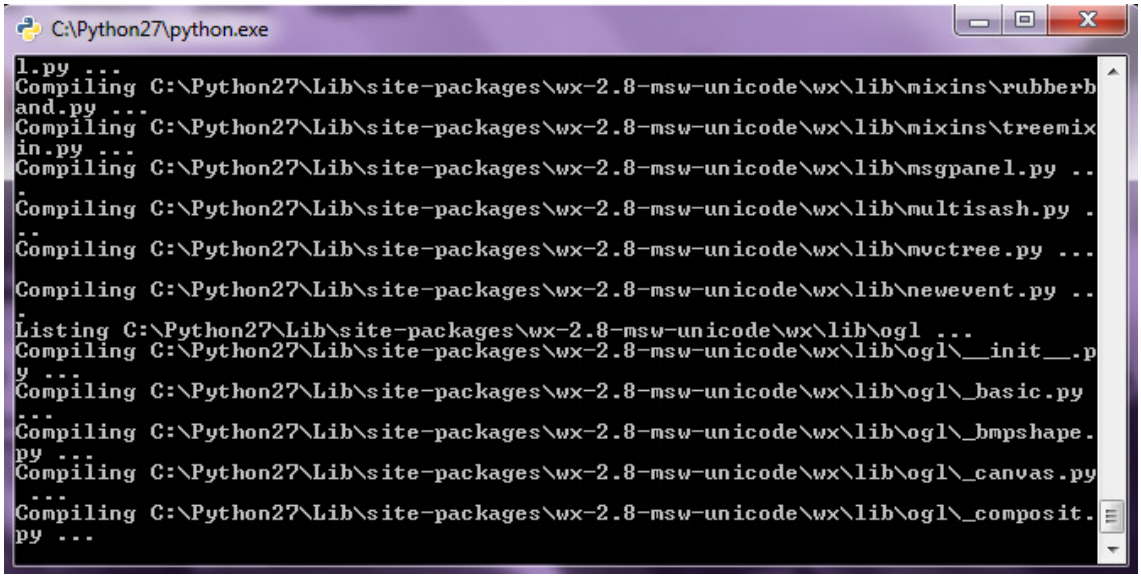
Kuva 20: Valitaan asennettavat komponentit.



Kuva 21: Asennusta suoritetaan.



Kuva 22: wxPythonin asennus on valmis.



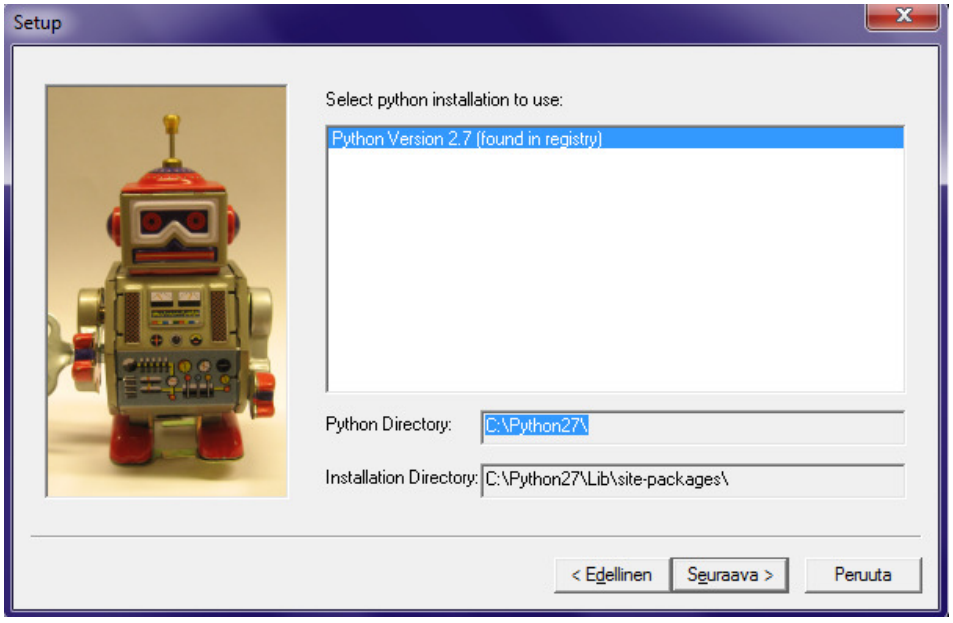
Kuva 23: py-muotoisten tiedostojen muuttaminen pyc-muotoisiksi.

Liite 3: Robot Frameworkin asennus

Robot Frameworkin asennus tapahtuu Pythonin päälle. (Klarck 2011.)

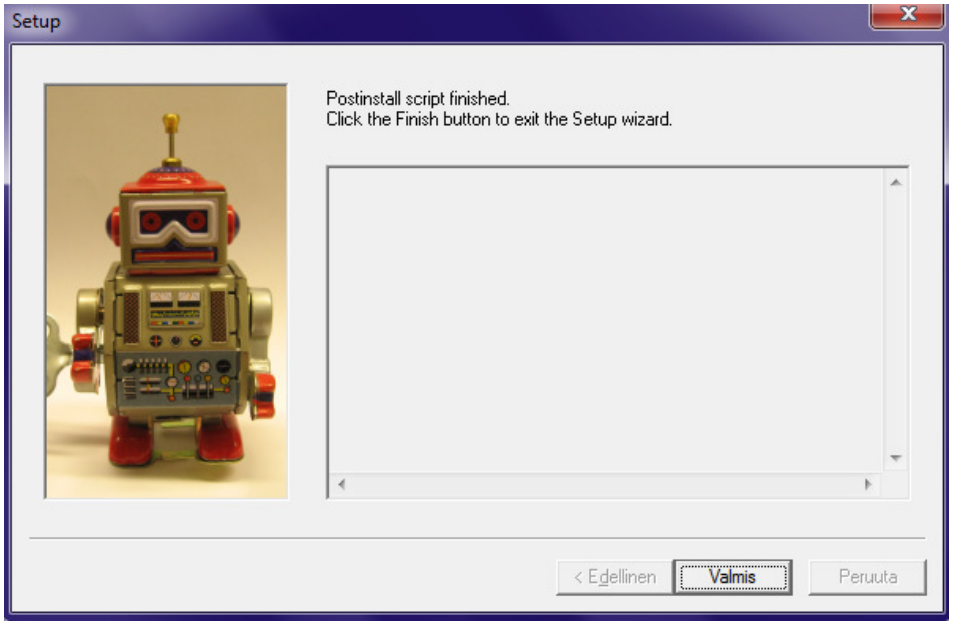
Asennus tapahtuu seuraavanlaisesti.

Kuva 24: Robot Frameworkin asennuksen aloittaminen.



Kuva 25: Robot Framework asennetaan myös Pythonin alle.

Kuva 26: Robot Frameworkin asennus suoritetaan.

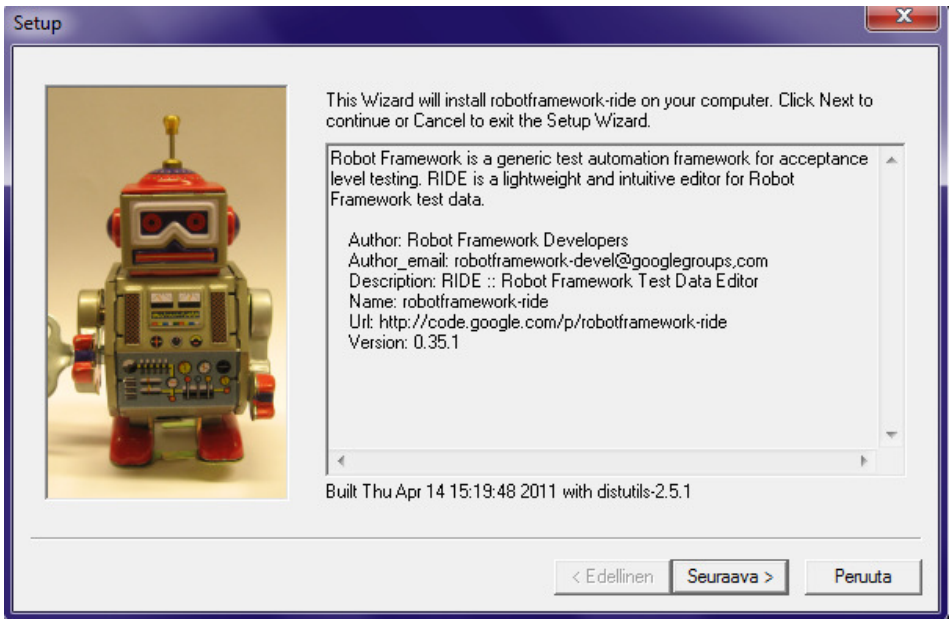


Kuva 27: Robot Frameworkin asennus suoritettu.

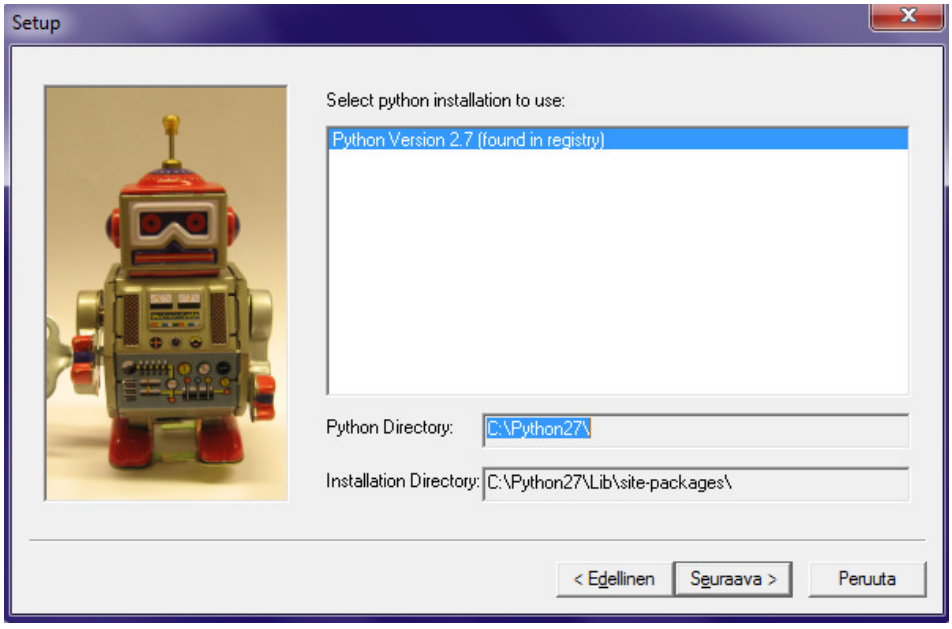
Liite 4: RIDE:n asennus

RIDE on Robot Frameworkin IDE-käyttöliittymä. Sen avulla tehdään ja muokataan testitapauksia. (robotframework – ride 2011.)

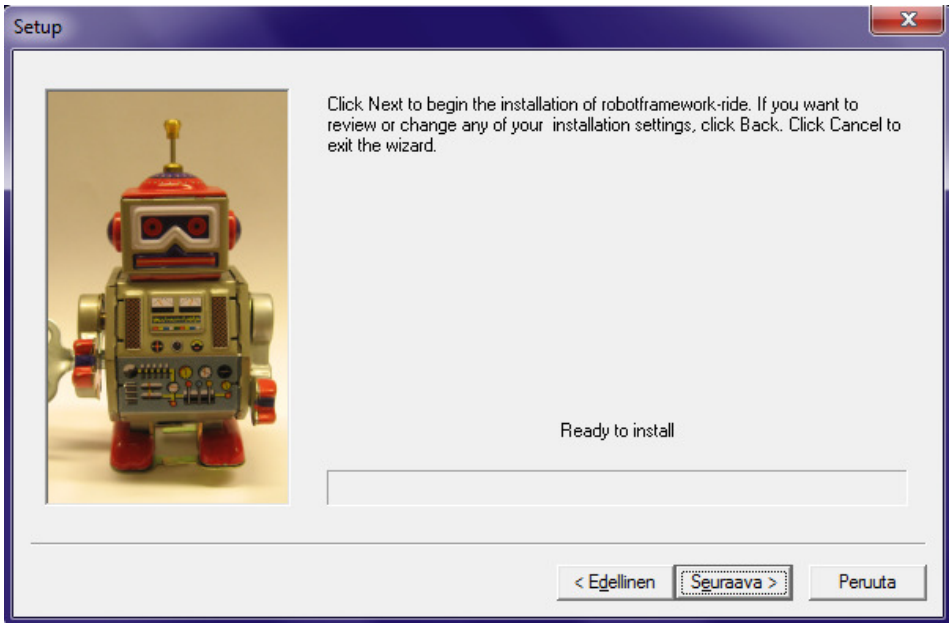
RIDE:n asennus tapahtuu seuraavanlaisesti.



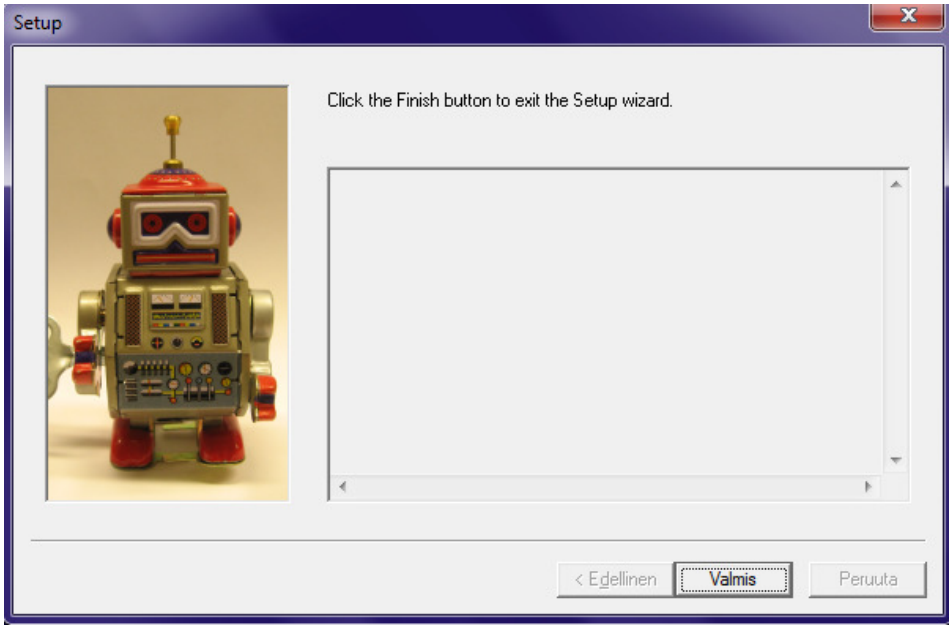
Kuva 28: RIDE:n asennuksen aloittaminen.



Kuva 29: RIDE asennetaan Pythonin alle.



Kuva 30: RIDE:n asennus on valmis suoritettavaksi.

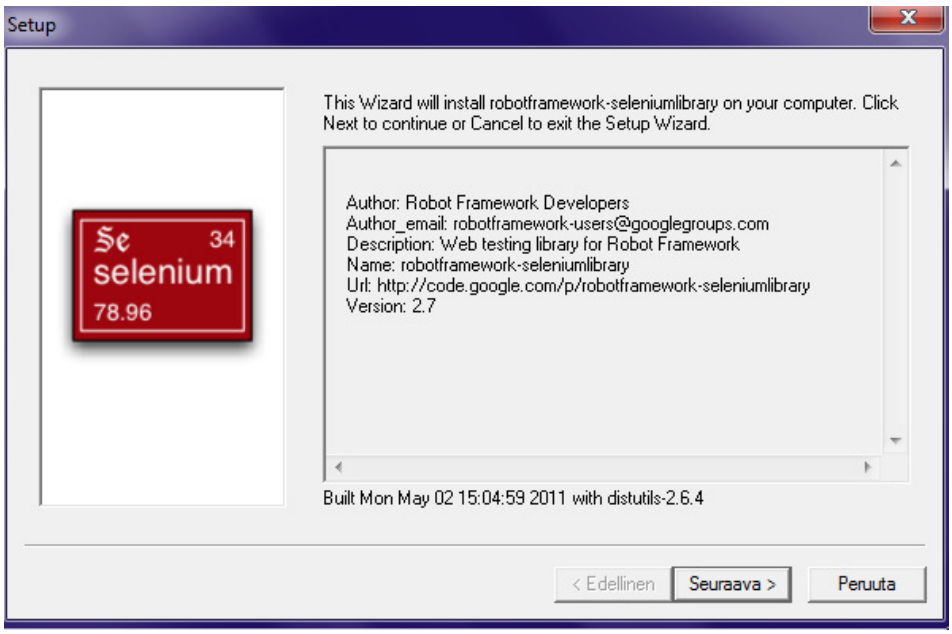


Kuva 31: RIDE:n asennus on suoritettu.

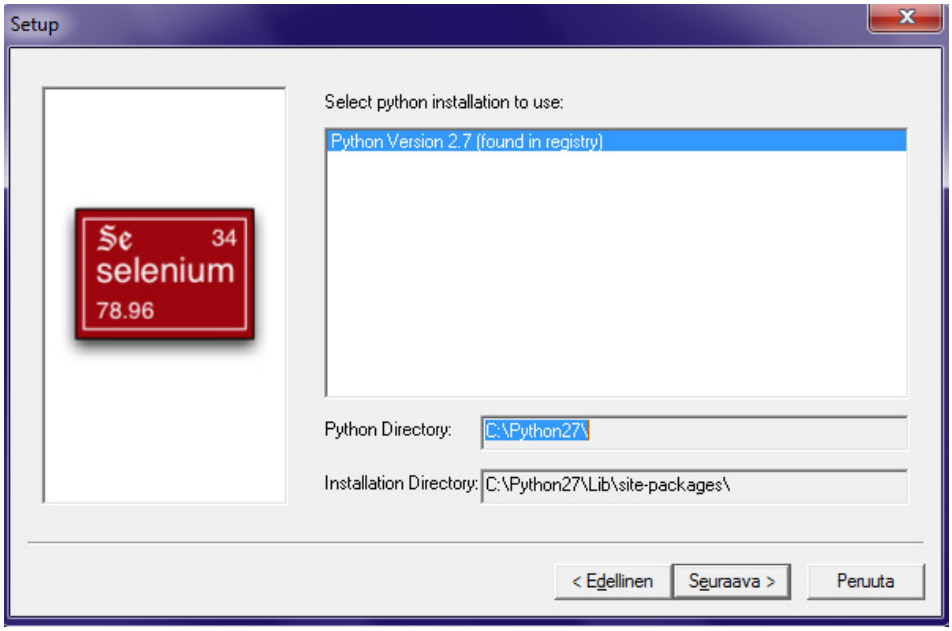
Liite 5: SeleniumLibraryn asennus

SeleniumLibrary on kirjasto, jota tarvitaan selaintestaukseen. SeleniumLibrary sisältää selaintestaukseen tarvittavat avainsanastot. (SeleniumLibrary 2011.)

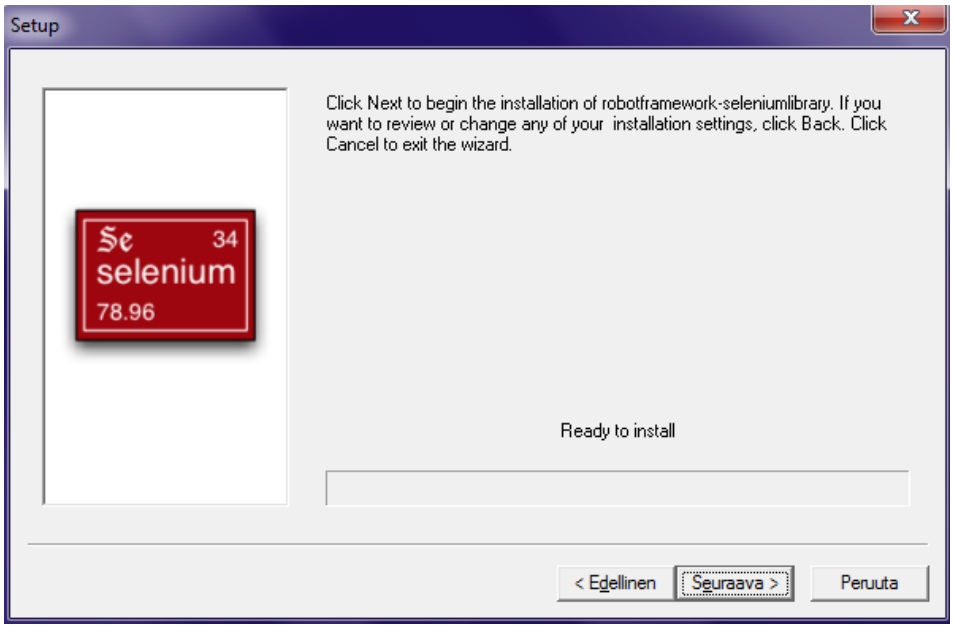
SeleniumLibraryn asennus suoritetaan seuraavanlaisesti.



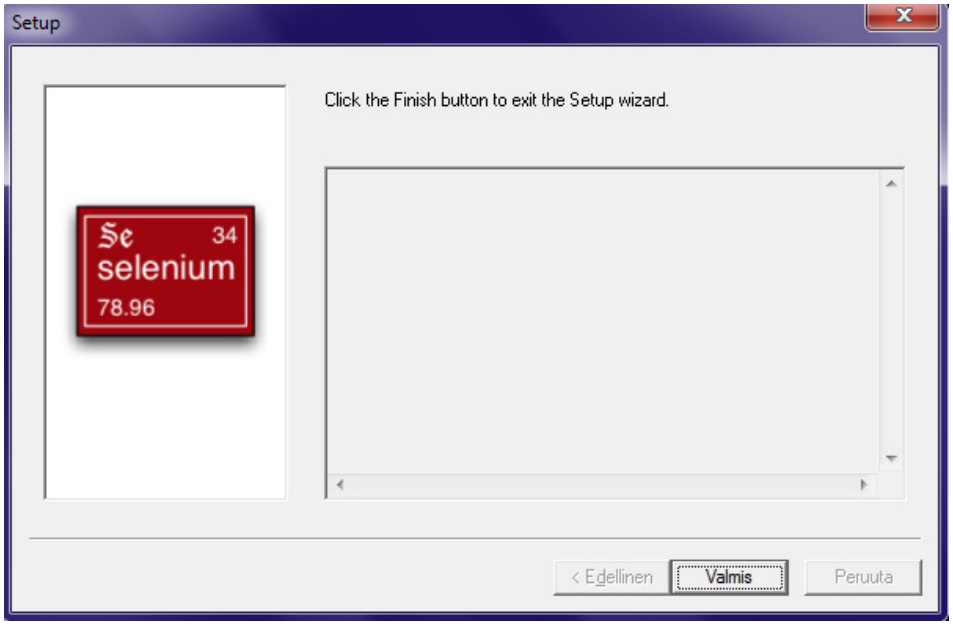
Kuva 32: SeleniumLibraryn asennuksen aloitus.



Kuva 33: SeleniumLibrary tulee myös asentaa Pythonin alle.



Kuva 34: Valmis asentamaan SeleniumLibraryn.



Kuva 35: SeleniumLibrary asennettu.