OAMK
OULUN AMMATTIKORKEAKOULU

Markku Nokkala

OHJELMISTOTESTAUKSEN AUTOMATISOINTI

Robot Framework – SikuliX

Ohjelmistotestauksen automatisointi

Robot Framework - SikuliX

Markku Nokkala
Opinnäytetyö
Syksy 2018
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehitys

Tekijä(t): Markku Nokkala
Opinnäytetyön nimi: Ohjelmistotestauksen automatisointi, Robot Framework - SikuliX
Työn ohjaaja: Veikko Tapaninen
Toimeksiantajan ohjaaja: Rami Potinkara, Aava Mobile Oy
Työn valmistumislukukausi ja -vuosi: Syksy 2018
Sivumäärä: sivut 45 + liitteet 5

Opinnäytetyön tavoitteena oli tutkia erilaisia ohjelmistotestausympäristöjä ja toteuttaa testiautomaatio Windows-alustalla. Työssä käydään läpi ohjelmistotestausta yleisesti ja työkaluja, joilla testiautomaatiota harkittiin toteutettavaksi. Viimeiseksi käydään läpi valittua testausympäristöä ja näytetään esimerkein sen toimintaa. Opinnäytetyön toimeksiantajana toimi Aava Mobile Oy, ja koska yrityksen kehitystyö on salaiseksi luokiteltua, opinnäytetyössä käsitellään aiheita yleisellä tasolla. Työn tarkoituksena oli tuottaa tietoa ohjelmistotestauksesta ja sen automatisoinnin mahdollistavista ympäristöistä.

Esiteltäviä ohjelmistokehyksiä vertailtiin keskenään työnantajalta saaduilta vaatimuksilla. Ohjelmistokehyksissä tuli olla tuki Windows-käyttöjärjestelmälle ja testauksen tuli tapahtua pääasiassa käyttöliittymän kautta hiiri- ja näppäinsimuloinnein. Näiden lisäksi ympäristön tuli ainakin olla hyvin tuettu, mahdollisimman kustannustehokas, muokattava sekä luotettava. Työ ei pyri kertomaan, mikä ohjelmistokehys on ainoa oikea valinta testiautomaation tekemiseen Windows-käyttöjärjestelmälle, vaan kertomaan ohjelmistokehyksistä tietopakettimaisesti ja vertailla niitä toimeksiantajalta saatuihin vaatimuksiin. Toteutus tehtiin lopulta SikuliX API:a hyödyntäen SikuliLibrary Robot Framework -kirjastolla.

Työ pääsi tavoitteisiinsa ja ohjelmistokehyksen valinnan jälkeen testiautomaatio luotettavuustestaukseen toteutettiin kokonaisuudessaan.

Asiasanat: Ohjelmistotestaus, Ohjelmistorobotit, Ohjelmistoautomaatio, Robot Framework, SikuliX

ABSTRACT

Oulu University of Applied Sciences
Bachelor of Science in Information Technology
Software Engineering

Author(s): Markku Nokkala
Title of thesis: Automating software testing, Robot Framework - SikuliX
Supervisor(s): Veikko Tapaninen
Client's supervisor: Rami Potinkara, Aava Mobile Oy
Term and year when the thesis was submitted: Fall 2018
Number of pages: 45 pages, 5 supplemental pages

The objective of this thesis was to examine different testing platforms and to implement test automation on Windows platform. Thesis will go through software testing on a general level and also introduce different test automation platforms. Finally, thesis will introduce the chosen platform and show some examples. The thesis was commissioned by Aava Mobile Oy and due to their R&D being confidential the subject of the thesis was only covered in general terms. The purpose of the thesis was to produce information about software testing in general and also introduce the reader to some of the existing test automation platforms.

Software platforms were compared to the requirements which were obtained from the employer. Chosen platform must have support for Windows operating system and also be able to test through graphical user interface with simulated mouse and keyboard controls. In addition, platform should also be well supported, cost-effective as possible, maintainable and reliable. Thesis is not answering to the question "Which software testing platform is the ultimate choice on Windows operating system?", but it will introduce the reader to a bunch of testing platforms and compare them within the requirements.

Thesis reached its goals successfully. Testing platform was chosen, and test automation was built completely for robustness testing.

Keywords: Software Testing, Software robots, Software automation, Robot Framework, SikuliX

SISÄLLYS

TIIVISTELMÄ ... 1
ABSTRACT ... 2
SISÄLLYS ... 3
SANASTO ... 5
1 JOHDANTO ... 6
2 OHJELMISTOTESTAUS ... 7
2.1 Ohjelmiston virheet ... 7
2.2 Testausprosessien mallit ... 8
2.2.1 V-malli ... 8
2.2.2 Spiraalimalli ... 9
2.3 Staattinen testaus ... 10
2.3.1 Katselmoinnit ... 10
2.3.2 Staattinen analyysi ... 11
2.4 Dynaaminen testaus ... 11
2.4.1 Laatikkomalli ... 12
2.4.2 Regressiotestaus ... 13
2.4.3 Hyväksymistestaus ... 14
2.4.4 Destruktiivinen testaus ... 14
2.4.5 Järjestelmätestaus ... 14
2.4.6 Yksikkötestaus ... 17
2.4.7 Integraatiotestaus ... 17
2.4.8 Validointitestaus ... 17
2.4.9 Sanity-testaus ... 18
2.5 Esimerkki testauksen kulusta ... 18
3 TESTIEN AUTOMATISOIMINEN ... 19
3.1 Automatisoinnin syyt ... 19
3.2 Automatisoinnin heikkoudet ... 19
4 OHJELMISTOKEHYKSEN VALINTA ... 21
4.1 Valintakriteerit automatisoinnin ohjelmistokehykselle ... 21
4.2 Tutkitut ja vaihtoehtoina olleet testiautomaatiot ... 22
4.3 Automatisointiohjelmistokehyksen valinta ... 27

5 SIKULIX ... 29
5.1 Käyttökohteet ... 29
5.2 Ympäristön toiminta ... 29
5.2.1 Jython ... 30
5.2.2 Java.awt.Robot ... 31
5.2.3 Java Native Interface ... 31
5.2.4 OpenCV ... 32
5.2.5 Tesseract ... 32
5.3 Käyttö ohjelmoinnissa ... 32
6 ROBOT FRAMEWORK ... 34
6.1 Testidatan rakenne ... 35
6.2 Valmiit kirjastot ... 36
6.3 Omat kirjastot ... 37
6.4 Suoritettavat tiedostot ... 37
7 SIKULILIBRARY ... 38
7.1 Toiminta ... 38
7.2 XML-RPC ... 39
8 TESTIAUTOMAATION TOTEUTUS ... 40
8.1 Testin toteutus ... 40
8.2 Testitapauksien suorittaminen ... 43
9 YHTEENVETO ... 44
LÄHTEET ... 46

SANASTO

.NET	Microsoftin kehittämä ohjelmistokomponenttikirjasto.
API	Application Programming Interface. Tarkoittaa suomeksi ohjelmointirajapintaa.
UI	User Interface eli käyttöliittymä
GUI	Graphical User Interface eli graafinen käyttöliittymä.
OCR	Lyhenne sanoista Optical Character Recognition. Yleistermi teknologialle, jota käytetään koneelliseen merkkien tunnistukseen.
OS	Tulee sanoista Operating System eli käyttöjärjestelmä
RS	Redstone, Microsoftin käyttämä koodinimitys uusille Windows 10 päivityspaketeille.
WYSIWYS	What You See Is What You Script. “Näet mitä skriptaat” -filosofia, jota SikuliX mainostaa noudattavan.

1 JOHDANTO

Testaus on tutkimusta, jota tehdään testattavana olevan tuotteen tai palvelun laadun määrittämiseksi. Määrittelyllä pyritään löytämään testauksen kohteena olevan asian puutteita taikka vikoja. Kuten kaikessa kehitystyössä, myös ohjelmistoja on testattava ennen niiden käyttöönottoa. Laajamittainen testaaminen on hyvin aikaa vievää ja itseään toistavaa, sillä samat asiat täytyy testata useasti kehitystyön eri vaiheissa. Tästä syystä testausta tulisi pyrkiä myös automatisoimaan, jotta testaukseen varatut henkilöresurssit voidaan käyttää hyödyllisemmin ja testaajien kannalta mielekkäämmin.

Opinnäytetyön toimeksiantajana toimii Aava Mobile Oy, joka valmistaa mobiililaitteita ammattikäyttöön. Yrityksellä oli tarve löytää ohjelmistokehys, jolla saadaan automatisoitua luotettavuustestaus ja osa suorituskykytestauksesta Windows-käyttöjärjestelmällä sekä kykenemään myöhemmin muuttamaan alusta toimimaan myös Android-alustalla. Luotettavuustestit toimivat ohjelmiston luotettavuuden mittarina ja niissä toistetaan usein asioita satoja tai tuhansia kertoja uudelleen ja yhä uudelleen. Toistojen määrät ovat niin suuria, että ihminen kyllästyisi niiden suorittamiseen nopeasti. Tylsä ja itseään toistava työ myös lisää inhimillisten virheiden mahdollisuutta, jolloin luotettavuustestauksessa tärkeä toistettavuus kärsisi. Toistettavuuden kärsiessä testi päätyisi todennäköisesti aina eri tulokseen. Tästä lähtöasetelmasta alkoi testiautomaatio-ohjelmistokehyksien tutkiminen ja sopivan alustan löytäminen automatisointiin, jonka jälkeen automaatio toteutettiin kokonaisuudessaan luotettavuustestaukseen.

Tässä opinnäytetyössä käsitellään ohjelmistotestausta ja sen automatisointia yleisellä tasolla sekä arvioidaan erilaisia automaatiotestaukseen soveltuvia ohjelmistokehyksiä ja valitaan valintakriteerien perusteella tavoitteeseen sopivin ratkaisu. Lopuksi työssä kerrotaan valitusta ohjelmistokehyksestä ja kerrotaan mitä kaikkea automatisoinnin tekemiseen tarvittiin. Opinnäytetyön tarkoituksena on tuottaa tietoa ohjelmistotestauksesta sekä sen automatisoinnista Robot Frameworkin ja SikuliX API:n avulla.

2 OHJELMISTOTESTAUS

Ohjelmistotestauksen perusperiaate on testata ja parantaa ohjelmiston laatua. Ohjelmistotestauksessa pyritään yleensä määrittämään ohjelmiston toimivuus, kehityksen tila ja turvallisuus. Testaaminen ei kuitenkaan voi vahvistaa, että ohjelmisto toimisi kaikissa mahdollisissa olosuhteissa sekä tilanteissa oikein. Testausta voidaan kutsua tekniseksi-tutkimukseksi, jota käytetään laadunvalvonnan työkaluna. Ohjelmistoja voidaan testata dynaamisilla ja staattisilla tavoilla. Staattiset tavat keskittyvät koodiin suorittamatta sitä, dynaamiset taasen keskittyvät nimenomaan sen suorittamiseen. (Kautto 1996.)

2.1 Ohjelmiston virheet

Ohjelmiston virheeksi voidaan käsittää mikä tahansa tila tai tapahtuma, joka poikkeaa vaatimuksista tai odotetusta lopputuloksesta (Software testing Overview 2018). Ohjelmiston virheet ovat yleensä kirjoitus- tai logiikkavirheitä koodissa, jotka aiheuttavat sovelluksen odottamattoman käytöksen tai kaatumisen. Yleensä virheet syntyvät niin ohjelmointi- kuin suunnitteluvaiheissa. Joskus virheet saattavat kuitenkin tapahtua myös kääntäjässä tai laitteistossa, jossa ohjelmaa suoritetaan (Burnstein 2003, 39—50.).

Testauksen yhteydessä virhe (engl. error, mistake, bug) on poikkeama spesifikaatiosta. Tästä tullaan lopputulokseen, jossa testausta ei voida tehdä ilman spesifikaatioita, sillä oikeaa lopputulosta ei voida todeta. Tavallisimmat spesifikaatiot, joita käytetään testauksessa, ovat toiminnallinen ja tekninen määrittely. Virheiden vakavuus voi vaihdella ja pienimmät virheet voivat olla lähinnä käyttäjää ärsyttäviä yksityiskohtia. Suurimmillaan virheet voivat estää koko järjestelmän käytön. (Haikala — Märijärvi 2004, 283—287.)

Ilkka Haikala ja Jukka Marijärvi arvioivat kirjassaan Ohjelmistotuotanto (2004, 283—287.): ”Ohjelmissa arvioidaan yleensä olevan ohjelmoinnin jälkeen yksi virhe muutamaa kymmentä ohjelmariviä kohden. Jopa pitkään käytössä olleissa ohjelmissa arvioidaan yleensä olevan noin yksi virhe tuhatta ohjelmariviä kohden.”. Osa virheistä ovat myös sellaisia, että niitä ei havaita ”koskaan”. Tätä selittää osittain mm. syöteavaruuden suuri koko ja se, että virheelliset kohdat eivät automaattisesti tuota virhetoimintoa, joka näkyisi käyttäjälle. (Haikala — Märijärvi 2004, 283-287.)

Tunnetuimpia ohjelmistovirheitä on esimerkiksi Y2K-bugiksi nimetty virhe, joka johtui osittain vaatimussuunnittelusta sekä toteutusvaiheessa tehdyistä päätöksistä. Y2K-bugin alkuperäinen syy johtui tallennustilan kalleudesta ja siitä, että ohjelmoijat säästivät resursseja ilmoittaen vuosiluvut kahdella numerolla. Yhtiöt ja organisaatiot päivittivät ja tarkistivat järjestelmiään maailmanlaajuisesti, jotta ongelmilta säästyttäisiin vuosiluvun vaihtuessa uudelle vuosituhannelle vuodesta 1999 vuoteen 2000. (Y2K Bug 2018.)

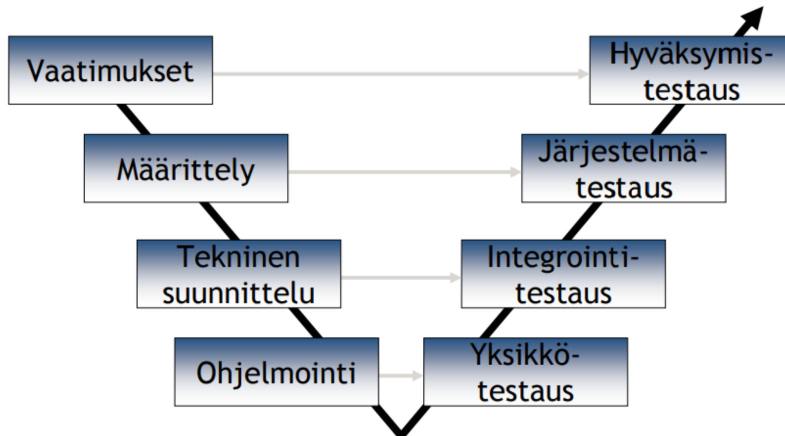
2.2 Testausprosessien mallit

Testaukseen on kehitelty useita eri malleja, joiden käyttö vaihtelee organisaatiosta ja tuoteprojektista toiseen. Mallit pyrkivät kuvaamaan koko testausprosessin yksinkertaisena kaaviokuvana. (Eichberg 2018.)

2.2.1 V-malli

Monimutkaiset ja laajemmat järjestelmät tarvitsevat abstraktiokerroksia tai rajapintoja jakamaan ohjelmiston komponentteja pienempiin yksiköihin. Samalla tulee kuitenkin myös varmistaa, että yksiköt toimivat kokonaisessa järjestelmässä ja tekevät tehtävänsä oikein. (V-mallin mukainen testausmenetelmä 2002.)

V-malli määrittelee testausprosessin ylläolevan mallin mukaisesti aloittamalla pienien ohjelmistokomponenttien testauksesta edeten kohti valmista kokonaisuutta. V-mallissa testausprosessi jaetaan loogisiin tasoihin (kuva 1), joissa jokainen taso täytyy läpäistä ennen siirtymistä seuraavalle tasolle. Ohjelmisto kootaan pienistä rajattujen toimintojen paloista edeten suurempiin kokonaisuuteen ja lopulta päädytään järjestelmätestaukseen, jolloin kyseessä on koko ohjelmiston testaus. (V-mallin mukainen testausmenetelmä 2002.)



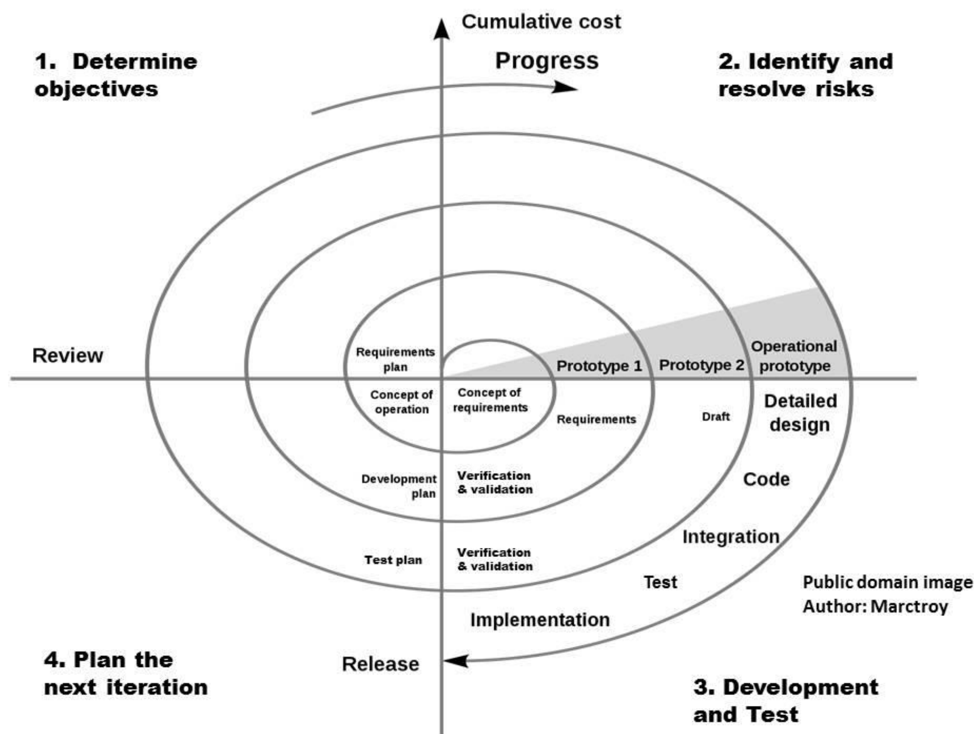
KUVA 1 V-Malli ja sen tasot (Pyhäjärvi 2006.)

2.2.2 Spiraalimalli

Boehmin malli eli spiraalimalli on Barry Boehmin vuonna 1986 kuvailema ohjelmistoprosessimalli, jolla pyritään kuvaamaan ohjelmistoprojektit riskimallien ja analyysien mukaan. Spiraalimallissa jokainen pyörähdys vastaa yhtä iteraatiota. Pyörähdys voi venyä riskin, esimerkiksi kustannuksien mukaisesti. Spiraalimalli toimii hyvin isojen ja kriittisen järjestelmien ohjelmistoprosessina. Spiraalimalli on esitettynä kuvassa 2 (kuva 2). Muita mallille tyypillisiä ominaisuuksia ovat

- ensimmäiset ohjelmistoversiot luodaan aikaisessa vaiheessa prosessia
- ominaisuuksia voidaan helposti lisätä ja poistaa
- vahva hyväksyttäminen ja dokumenttien hallinta
- riskianalyysin tekeminen vaatii todella spesifistä osaamista
- projektin onnistuminen perustuu todella paljon riskianalyysiin
- ei toimi hyvin pienten projektien kanssa.

(Maritta 2008.)



KUVA 2 Spiraalimalli eli Boehmin malli (Wikimedia Commons - Spiral Model 2018.)

2.3 Staattinen testaus

Staattisessa testauksessa keskitytään koodiin suorittamatta sitä. Analyysiin voidaan käyttää analyysityökaluja tai ihan yksinkertaisesti ihmismieltä. Staattisen testauksen tavoitteena on nostaa laatua ja poistaa vikoja mahdollisimman aikaisessa vaiheessa. (Tuovinen 2013.)

2.3.1 Katselmoinnit

Katselmoinneissa käytetään asiantuntevia ihmisiä asioiden arvioinnissa ja tarkistamisessa. Katselmoinneilla pyritään löytämään vikoja ja epäjohdonmukaisuuksia dokumenteista tai ohjelmakoodista, jolloin niiden vaikutukset eivät peilaudu myöhemmissä projektin kehitysvaiheissa. (Burnstein 2003, 301—307.)

Katselmoinnit voivat vähentää dynaamisen testauksen aikana muutoin ilmeneviä bugeja, jolloin ne olisivat myös kalliimpia korjata. (Tuovinen 2013.)

Erilaisia katselmointityyppejä ovat esimerkiksi:

- läpikäynnit (walkthrough)
- tarkastukset (inspection)
- tekninen katselmointi (technical review)
- epämuodollinen katselmointi (informal review).

Erilaisten katselmointityyppien rajat ovat lopulta häilyviä ja organisaatiot muokkaavat katselmoinneistaan itselleen sopivia kohteen mukaisesti. Katselmointien perusidea on kumminkin ylläpitää laatua ja löytää kohteesta virheitä, poikkeavuuksia ja löytää vaihtoehtoisia ratkaisutapoja. (Burnstein 2003, 301—307.)

2.3.2 Staattinen analyysi

Staattisessa analyysissä tarkoituksena on löytää katselmointien tapaan vikoja tai niille alttiita kohtia tarkasteltavasta dokumentista tai ohjelmakoodista. Apuna käytetään automaattisia työkaluja käsin tehtävän analyysin sijaan. Työkaluja voidaan käyttää myös mittausten tekemiseen laadun arvioimiseksi. Staattinen analyysi voi paljastaa epäilyttäviä ja vaarallisia ohjelmarakenteita, joita dynaaminen testaus ei paljasta. (Tuovinen 2013.)

Staattisella analyysillä voidaan löytää potentiaalisesti ongelmallisia rakenteita kuten

- syntaksivirheet
- poikkeamat koodauskäytännöistä ja –standardeista
- kontrollivuon säännöttömyydet
- datavuon säännöttömyydet
- tietoturvan kannalta vaaralliset koodirakenteet.

(Tuovinen 2013.)

2.4 Dynaaminen testaus

Dynaamisessa testauksessa keskitytään ohjelman tai koodin käyttäytymiseen suorituksen aikana. Dynaaminen testauksen suorittamiseksi tarvitaan siis jokin toimiva ympäristö tai ohjelma, missä ainakin osan funktioista voidaan suorittaa. (Kautto 1996.)

2.4.1 Laatikkomalli

Dynaaminen testaus jaetaan perinteisesti kahteen pääluokkaan, musta- (eng.. blackbox) sekä valkolaatikkotestaukseen (eng. whitebox testing). Näiden lisäksi on olemassa myös harmaalaatikkotestaus (eng. greybox testing) joka jakaa musta- sekä valkolaatikkotestauksen ominaisuuksia. Eri laatikkomallit kuvaavat testitapauksien suunnittelemisen lähtökohtia. (Kautto 1996.)

Musta laatikko -testaus

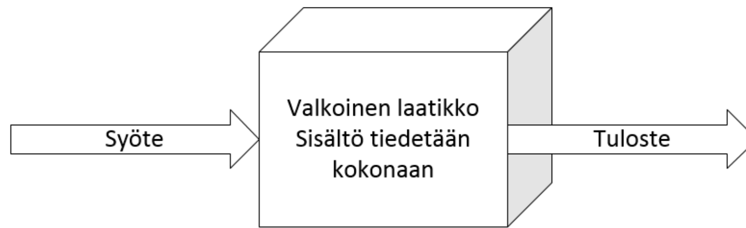
Musta laatikko eli Black box -testaus perustuu testattavan asian syöte - tuloste tuloksiin (kuva 3). Tässä testaustyylissä ei välitetä testauskohteen rakenteesta tai sisällöstä, vaan tutkimuksessa välitetään tulosteesta erilaisilla syötteillä. (Kautto 1996.)

Testauksen kohde on siis tuntematon, jota kuvataan mustalla laatikolla. Kohteesta saatuja tulosteita vertaillaan haluttuihin ja odotettuihin tuloksiin. Testitapaukset tehdään tässä mallissa aina kohteen määrittelyn perusteella. (Kautto 1996.)

KUVA 3 Musta laatikko -testauksessa ei tiedetä testattavan kohteen sisällöstä mitään

Valkoinen laatikko -testaus

Valkoinen laatikko eli White box -testauksessa testitapaukset johdetaan koko ohjelman sisäisestä rakenteesta ja logiikasta. Tässä testaustyylissä tiedetään testattavan asian sisältö kokonaisuudessaan (kuva 4). Testitapaukset määritetään niin, että kaikki mahdolliset koodin osa-alueet tulevat testattua. (Kautto 1996.)



KUVA 4 Valkoinen laatikko -testauksessa testattavan asian sisältö tiedetään kokonaisuudessaan

Harmaa laatikko -testaus

Harmaa laatikko eli Grey box -testaus yhdistää edellä mainittujen valkoinen ja musta laatikko testauksen malleja. Grey box -testauksessa testaaja tietää osittain, mutta ei kokonaan, testattavan asian rakenteen (kuva 5). Grey box -testaaja tarvitsee testattavasta kohteesta perustavanlaatuisen dokumentaation, jonka avulla testitapaukset voidaan määrittää. (Eriksson 2016a.)

KUVA 5 Harmaa laatikko -testauksessa testattavan asian sisältö tiedetään vain osittain

2.4.2 Regressiotestaus

Regressiotestaus pyrkii keskittymään virheiden etsimiseen ja löytämiseen laajan ohjelmistomuutoksen jälkeen. Regressiota on esimerkiksi ominaisuuksien häviäminen tai toiminnan yleinen heikkeneminen. Ohjelmistoissa myös korjattujen bugien uudelleenilmaantuminen on regressiota. (Holopainen 2005.)

Yleiset metodit regressiotestauksessa on suorittaa alueelle määritellyt testitapaukset uudelleen muutoksen jälkeen, jotta voidaan olla varmoja, ettei uusia vikoja tai aiemmin korjattuja vikoja ole ilmaantunut uudelleen. Tarvittava testauksen määrä voidaan määritellä testausvaiheen sekä lisäyksen tuoman riskin perusteella. (Holopainen 2005.)

2.4.3 Hyväksymistestaus

Ohjelmistoprojekteissa hyväksymistestauksen idea on selvittää, että vaatimukset, spesifikaatiot tai sopimus on täytetty. Asiakkaan tulisi määrittää mitkä ovat vaatimukset, josta ohjelmiston tulisi suoriutua. (Katara 2011.)

Ohjelmistoprojektin alussa smoke-testausta (kutsutaan myös sanity testaukseksi) ensimmäiselle koontiversiolle voidaan pitää eräänlaisena hyväksymistestauksena, jolla määritellään, onko ohjelmiston maturiteetti tarpeeksi korkea, jotta voidaan siirtyä syvempään testaukseen. (Katara 2011.)

2.4.4 Destruktiivinen testaus

Destruktiivisen testauksen ideana on yrittää kaataa koko ohjelmisto tai jokin sen osista. Näin voidaan varmistaa, että ohjelmisto toimii luotettavasti myös saadessaan esimerkiksi vääriä syötteitä. (Destructive testing 2017.)

Fuzzing on yksi destruktiivisen testauksen tyyleistä, jossa automatisoituna syötetään vääriä, odottamattomia tai satunnaista dataa ajettavaan ohjelmistoon. Ohjelman toimintaa seurataan yleisesti kaatumisten, muistivuotojen yms. varalta. Fuzzing kuuluu myös oleellisesti luotettavuustestauksen piiriin. (Destructive testing 2017.)

2.4.5 Järjestelmätestaus

Järjestelmätestauksessa testataan kokonaista ohjelmaa ja tarkastellaan vastaako se sille asetettuja vaatimuksia ja määrittelyjä. Ympäristöön kuuluu käytettävä laitteisto ja ohjelmistot.

Järjestelmätestauksen vaiheessa tulisi keskittyä ohjelman toiminnan varmistamiseen ja hienosäätöön. Mikäli aikaisemmissa projektin vaiheissa ei ole testattu tarpeeksi ja huomioitu virheiden välttämistä, se yleensä huomataan viimeistään tässä testauksen vaiheessa. Ohjelmointivirheiden korjaamisen järjestelmätestauksen yhteydessä on yleensä ohjelmistoprojekteissa kallista ja aikaa vievää, joten siksi tässä vaiheessa keskitytäänkin olemassa olevien ominaisuuksien hiomiseen. (Testauksen tasot 2017.)

Järjestelmätestaus on testaamisen eri tasoista laajin, ja se jaetaan tästä syystä useisiin eri osa-alueisiin. Jokaisella osa-alueella on omat tavoitteensa, joita vasten ohjelmaa testataan. Järjestelmätestauksen osa-alueita voi olla

- tietoturva
- käytettävyys
- suorituskyky
    - rasituksen sieto
    - stressin eli kapasiteettia koettelevien rasituspiikkien sieto
- luotettavuus.

(Testauksen tasot 2017.)

Tietoturvatestaus

Testiprosessi jolla voidaan määrittää vikoja tai puutteita tuotteen turvallisuudessa. Tietoturvatestaus ei kuitenkaan vastaa kuin ennalta määritettyihin tapauksiin jolloin se ei voi sata prosenttisesti varmistaa, ettei tuotteessa ole tietoturvan kannalta puutteita. Tietoturvatestaus koostuu useasta eri tyypistä, joita ovat

- heikkouksien skannaus
- turvallisuus skannaus
- penetraatiotestaus
- riskien arvioiminen
- turvallisuus auditointi
- kokonaisuuden arviointi
- eettinen hakkerointi.

(What is Security Testing: Complete Tutorial 2018.)

Käytettävyystestaus

Käytettävyystestauksen tarkoituksena on mitata, kuinka käyttökelpoinen on järjestelmä sitä käyttävälle ihmiselle. Käytettävyysarviointi yleensä suoritetaan oikeiden peruskäyttäjien kanssa. Peruskäyttäjät käyttävät järjestelmää tiettyjen tehtävien suorittamiseen. Arvioinnissa pyritään

seuraamaan, kuinka helposti, nopeasti, virheittä ja miellyttävästi järjestelmän käyttö onnistui käyttäjältä. (Hangasmaa 2010, 25.)

Suorituskykytestaus

Suorituskykytestauksessa pyritään määrittämään, että ohjelmisto täyttää sen suorituskykyvaatimukset. Sitä käytetään työkaluna ohjelmiston laadun mittauksessa. Ohjelmiston suorituskykyvaatimuksina voivat olla

- ohjelmiston responsiivisuus
- ohjelmiston stabiilisuus erilaisilla kuormituksilla
- ohjelmiston resurssien syönti ajettavassa laitteistossa, esim. virrankulutus sulautetuissa järjestelmissä.

(What is Performance Testing? 2018.)

Suorituskykytestausten alle kuuluu erilaisia testaustyyppejä, joilla pyritään määrittämään suorituskykyä eri osa-alueilla. Erilaisia testaustyyppejä ovat

- rasituksen sieto -testaus
- stressitestaus
- soak-testaus
- rasituspiikkitestaus
- konfiguraatiotestaus
- internet-testaus.

(What is Performance Testing? 2018.)

Luotettavuustestaus

Luotettavuustestauksessa pyritään määrittämään testattavan kohteen luotettavuus pitkällä aikavälillä. Ohjelmistotestauksessa luotettavuutta pyritään mittaamaan, tekemällä paljon toistuvia asioita, jotka mahdollisesti saattaisivat horjuttaa ohjelmiston luotettavuutta. Luotettavuustestausta käytetään koko tuotteen suunnitteluvaiheen ja markkinoilla olemisen aikana. Esimerkiksi sulautetuissa järjestelmissä saatetaan käyttää erilaisia komponentteja tuotteen elinkaaren aikana, jotka vaikuttavat myös laitteessa pyörivän ohjelmiston toimintaan. Luotettavuustestauksella

pyritään varmistamaan tässä tapauksesa, että määritetyt vaatimukset täyttyvät myös uusilla kokoonpanoilla. (Reliability testing 2004.)

2.4.6 Yksikkötestaus

Yksikkötestauksella tarkoitetaan pienimmän mahdollisen ohjelman osan esimerkiksi yksittäisen funktion toiminnan testaamista. Yksikkötesteillä voidaan varmistaa, että ohjelman yksittäiset osaset toimivat odotetulla, sekä halutulla tavalla, ja että suoritus onnistuu myös mahdollisissa virhetilanteissa. Ohjelma ei saa myöskään tehdä mitään sellaista, mitä sen ei ole tarkoitus tehdä (Testauksen tasot 2017.).

Yksikkötestauksen hyödyt näkyvät kehitysprosessin aikana erityisesti silloin, kun jo kirjoitettuun koodiin joudutaan tekemään muutoksia. Automatisoiduilla yksikkötesteillä voidaan vaivattomasti ja ripeästi todeta, aiheuttavatko tehdyt muutokset virheitä. Kattavat yksikkötestit ovat tärkeitä erityisesti ketterissä menetelmissä (Testauksen tasot 2017.).

Useissa ketterissä menetelmissä hyödynnetään testivetoista kehitysprosessia, jossa yksiköiden toteutus aloitetaan kirjoittamalla sille testit ja koodaamalla itse yksikkö vasta tämän jälkeen (Testauksen tasot 2017.).

2.4.7 Integraatiotestaus

Pääpainopisteenä integrointitestausvaiheessa on yksittäisen komponentin ja siihen liittyvien komponenttien väliset rajapinnat. Komponentin sisäinen toiminta tulisi varmentaa jo yksikkötestausvaiheessa. Integraatiotesteissä testataan siis, miten useampi komponentti toimii yhdessä (Testauksen tasot 2017.).

2.4.8 Validointitestaus

Validointitesteillä voidaan varmistaa ohjelman vaatimusten mukainen toimivuus käytännön tilanteissa. Validointitestauksessa käyttötapauksista tehdään testejä, joiden avulla koitetaan toteamaan ohjelman toimivuus todellisissa käyttötilanteissa, sekä löytämään mahdollisia

ongelmakohtia. Validointitesteillä pyritään siis varmistamaan, että ohjelmisto vastaa sille asetettuihin vaatimuksiin ennen siirtymistä järjestelmätestaukseen (Testauksen tasot 2017.).

2.4.9 Sanity-testaus

Sanity-testaus (myös sanity check) on testaustapa, jolla voidaan määrittää, onko koontiversio tarpeeksi toimiva, jotta olisi mahdollista tai järkevää jatkaa syvempään testaukseen. Testaus sisältää yleisesti muutaman yksinkertaisen testitapauksen, joilla voidaan varmistaa, että ohjelmisto toimii johdonmukaisesti kaikilta osa-alueiltaan. Jos testaus epäonnistuu, ei ole järkevää jatkaa syvempään testaukseen. (Eriksson 2016b.)

Sanity-testauksen ideana on säästää aikaa turhalta testaukselta, mikäli ohjelmisto ei ole vielä tarpeeksi kypsä siihen. Ohjelmistojen sanity-testaus on yleensä isommissa organisaatioissa myös automatisoitu ja toimii osana ohjelmistokehitystä. (Eriksson 2016b.)

2.5 Esimerkki testauksen kulusta

Testauksen kulusta löytyy variaatioita yhtä paljon kuin on organisaatioitakin, mutta tyypillisesti testauksen kulku voidaan jakaa seuraavasti:

- vaatimusanalyysi
- testaamisen suunnittelu
- testien kehittäminen
- testiautomaation luonti (mikäli testaus on automatisoitu)
- testaus
- testaustulosten läpikäynti ja raportointi
- korjaukset ja uudelleentestaus
- toimitus ja ylläpito.

(Software testing lifecycle 2018.)

3 TESTIEN AUTOMATISOIMINEN

3.1 Automatisoinnin syyt

Automatisoinnille löytyy useita pieniä ja suurempia syitä, mutta suurimmat ovat

- tarkkuus - Automaatio hoitaa työnsä tarkasti ja aina samalla tavalla. Vähentää ihmisistä johtua virheitä.
- toistettavuus - Testiä voidaan ajaa useaan otteeseen nopealla tahdilla aina täsmälleen samalla tavalla kuin edellisellläkin kerralla.
- nopeus - Automaatio ei välitä kellonajoista ja sitä voidaan pyörittää kellonympäri. Automaatiolla ei ole myöskään ihmisten tarpeita kuten syöminen, juominen, vessassa käynti tai nukkuminen.
- kustannustehokkuus - Valmiin testin ja testausympäristön jälkeen suurimmat kulut koostuvat tietokoneen ylläpidosta jolla automaatio pyörii ja silloin tällöin tarvitusta ylläpidosta.

Mitä enemmän testauksesta saadaan automatisoitua työläitä testauksen vaiheita, sitä paremmin ja hyödyllisemmin testaukseen varatut henkilöresurssit voidaan käyttää. Tällöin muuten manuaalisesti toistettavat sovelluksen kontrolloinnit siirtyvät automaation kontolle ja henkilöt voivat keskittyä syvemmin järjestelmän toimintaan, analysoimaan paremmin löydettyjä virheitä tai kehittelemään uusia testitapauksia. (Miten käyttöliittymä testataan automaattisesti 2018.); (Potinkara 2017-2018.); (Kaner Cem — Bach James — Pettichord Bret 2002.)

3.2 Automatisoinnin heikkoudet

Ohjelmistojen automatisoinnilla on myös omat heikkoutensa, joita ovat

- laitteistorajoitukset.
- testiautomaatiolla ei voida varmistaa koko järjestelmän ja kokonaisuuden toimintaa kerralla. Jokainen asia on silti testattava erikseen (vaikkakin se voidaan automatisoida pitkälle).
- automaatioissa joissa käytössä on konenäköön pohjautuva pinta-automaatio, on otettava skaalaukset ja resoluutiot tarkasti huomioon. Konenäkö ei myöskään ole aina oikeassa ja sitä voidaan hämätä helposti.
- automaatti ympäristöt eivät ole vielä tarpeeksi fiksuja. Manuaalista työtä ei voida (ainakaan vielä) korvata kokonaan automatisaatiolla. Myös manuaalista työtä vielä tarvitaan.

- automaattisten testien lopputulosta kohtaan pitää olla myös kriittinen, etenkin kun automatisaatio on uusi. Automaattitestikin voi olla virheellinen tai siinä voidaan tulkita virhetilanne odotetuksi tilanteeksi.
- lisää työmäärää isojen muutosten tapahtuessa, koska testejä on päivitettävä. (Esim. Android käyttöjärjestelmä versio muuttuu M->O). Lisää väliaikaisesti myös testaukseen menevää aikaa.
- automatisointia ei ole yksinkertaisesti valmisteltu havaitsemaan esiintyvää virhettä, jolloin sitä ei huomata.

(Potinkara 2017-2018.); (Kaner Cem — Bach James — Pettichord Bret 2002.)

4 OHJELMISTOKEHYKSEN VALINTA

Testiautomaatioiden toteutukseen oli useita vaihtoehtoja, mutta tässä tapauksessa tarve oli kokonaiselle ohjelmistokehysratkaisulle, jolla onnistuu käyttöliittymän kautta tapahtuva testaus Windows-alustalla.

Alla olevassa luvussa on lueteltu muutama toimeksiantajan kriteeri ympäristölle ja niiden jälkeen ohjelmistokehysvaihtoehdot, jotka olivat ehdolla testiympäristön luomiseen. Viimeisenä luvussa tehdään taulukkomuotoinen vertaus, jossa käytetään luvun 4.1 listattuja kriteerejä. Kriteerien listaus on pyritty yksinkertaistamaan niin, että ne voidaan ilmaista taulukossa mahdollisimman lyhyesti, joten lukijan kannattaa käyttää lukua 4.1 laajempana referenssinä taulukon selitteistä (taulukko 1).

4.1 Valintakriteerit automatisoinnin ohjelmistokehykselle

Yrityksen luettelemat valintakriteerit ohjelmisto kehitykselle olivat:

1. Pinta-automaatio. Ympäristö käyttää kuvantunnistukseen perustuvaa automaatiota
2. Windows-tuki. Testaus on mahdollista Windows käyttöjärjestelmällä
3. Siirrettävyys. Ympäristö voidaan siirtää pienillä muutoksilla muille alustoille kuten Android.
4. Kustannustehokkuus. Esimerkiksi avoimen koodin ilmaiset järjestelmät
5. Laajennettavuus. Ympäristöön voidaan lisätä omia laajennuksia ja ympäristöllä voidaan myös suorittaa aikaisemmin kehitettyjä sovelluksia
6. Helppokäyttöisyys. Uusien testien kirjoittaminen on nopeaa ja helppoa. Ympäristö voidaan myös rakentaa niin, että kaikilla ympäristöä käyttävillä ei tarvitse olla laajaa ohjelmointikokemusta
7. Tuettavuus. Ohjelmistokehykselle löytyy tukea joko ison avoimen yhteisön kautta tai suuren yrityksen kautta
8. Luotettavuus. Ohjelmistokehys itsessään toimii kuten pitää, toimii luotettavasti ja ei vaikuta päätelaitteen suorituskykyyn luotettavuuteen huonontavasti
9. Soveltuu ensisijaisesti luotettavuus ja suorituskyky testaukseen. Toissijaisesti ympäristöä voidaan käyttää myös yksittäisten testien automatisointiin

10. Muokattavuus. Lähdekoodiin pääsee käsiksi. Tarvittaessa päästään kiinni lähdekoodiin joko avoimen koodin tai kaupallisen sopimuksen puitteissa ja itse koodia päästään tutkimaan ja muokkaamaan ja tarvittaessa jos on kääntäjäriippuvainen kieli toteutus myös kääntämään
11. Simuloitavuus. Hiiri- ja näppäinsimuloinnit. Automaatiolla voidaan simuloida oikean käyttäjän tekemiä liikkeitä käyttöjärjestelmässä
12. Ylläpidettävyys. Ympäristö on helppo ylläpitää ja sitä ei tarvitse rakentaa joka kerta uudelleen, kun käyttöjärjestelmäversio vaihtuu
13. Skaalautuvuus. Testiympäristöä voidaan kehittää niin, että yhdellä testikontrollerilla voidaan testata enemmän kuin yhtä laitetta

4.2 Tutkitut ja vaihtoehtoina olleet testiautomaatiot

Seuraavaksi käsitellään tutkittuja ja vaihtoehtoina olleita testiautomaatioita, joilla järjestelmää suunniteltiin toteutettavan:

Pywinauto

Pywinauto on alun perin Mark Mc Mahonin kirjoittama Pythoniin pohjautuva projekti, jolla voidaan automatisoida Windowsin käyttöliittymässä tapahtuvia hiiri- ja näppäimistöeleitä. Projekti lähti liikkeelle vuonna 2006, josta se on siirtynyt useiden vaiheiden jälkeen Open Source Communityn ylläpitämäksi projektiksi. Pywinauto ei tarjoa kuvantunnistukseen perustuvaa ratkaisua vaan se perustuu pelkkään Win32 API:n hyödyntämiseen. (What is pywinauto 2018.)

Pywinauto:n ominaisuudet ovat:

- ei pinta-automaatiota
- avoin lähdekoodi
- ilmainen käyttää kaupallisesti.

AutoIT

AutoIT on ilmainen automaatiokieli Windows-alustalle. Ohjelmistokehys oli alun perin pääasiassa tarkoitettu makrojen tekemiseen, mutta on myöhemmin kehittynyt kielenä sekä ominaisuuksiltaan.

AutoIT on alkujaan julkaistu vuonna 1999 ja on päässyt kehityskaaressaan kolmanteen versioon. Ensimmäiset kaksi versiota olivat lausekepohjaisia kieliä, jotka olivat tarkoitettu pääasiassa simuloimaan käyttäjän tekemiä liikkeitä graafisessa käyttöliittymässä. Kolmannesta versiosta eteenpäin AutoIT on muuttunut BASIC-ohjelmointikieliperheen tyyliseksi syntaksiltaan ja laajentanut toiminnallisuuttaan. Versio 3 AutoIT-kielestä tukee Windows-käyttöjärjestelmiä Windows XP SP3:sta ylöspäin. Vanhemmat versiot tukivat käyttöjärjestelmiä aina Windows 95-versiosta eteenpäin. (AutoIT overview 2017.)

AutoIT-skripti voidaan muuntaa kompressoiduksi itsenäiseksi suoritettavaksi tiedostoksi, jota voidaan ajaa laitteilla, joissa ei AutoIT-tulkkia ole. AutoIT sisältää mukanaan laajan skaalan erilaisia funktio kirjastoja vakiona. Funktiokirjastoja jaetaan myös internetin erinäisissä palveluissa ladattavaksi. Asennuspaketti sisältää SciTE-editoriin perustuvan kehitysympäristön. (AutoIT overview 2017.)

Auto IT:n ominaisuudet ovat:

- ei pinta-automaatiota
- skriptikieli BASIC -tyylisellä rakenteella
- lisättäviä kirjastoja ja moduuleja tietyille applikaatioille
- tukee TCP ja UDP -protokollia
- tukee COM-objekteja
- mahdollisuus kutsua Win32 API:n funktioita
- mahdollista ajaa konsoliohjelmia ja päästä käsiksi standardidatavirtoihin
- tukee tiedostojen pakkausta suoritettavaan tiedostoon
- graafinen käyttöliittymä käyttäjän syötteille
- soittaa, keskeyttää, jatkaa, lopettaa ja hakee tämän hetkisen kohdan audiotiedostoista. Tukee myös audiotiedoston pituuden hakemisen.
- hiiren liikkeiden simuloiminen
- ikkunoiden ja prosessien manipulointi

- mahdollistaa syötteiden ja näppäinpainallusten automatisoinnin sovelluksiin
- skriptit voidaan kääntää erilleen suoritettaviksi
- tukee säännöllisiä lausekkeita
- toimii yhteen Windowsin UAC:n kanssa (UAC = User Account Control)
- olioperustainen suunnittelu kirjastojen kautta.

(AutoIT overview 2017.)

Esimerkki AutoIT -skriptistä liitteessä 1.

Selenium

Selenium on avoimen lähdekoodin ohjelmistokehys, joka on kehitetty pohjimmiltaan webapplikaatioiden, kokonaisten nettisivujen tai selaimien testaukseen. Ohjelmistokehys on alun perin kehitetty Jason Hugginssin toimesta vuonna 2004 ThoughtWorks -nimisen yrityksen sisäiseen käyttöön. (Selenium history 2017.)

Selenium tukee useita eri ohjelmointikieliä, joiden avulla toiminnallisuutta pystytään laajentamaan. Selenium soveltuu pohjimmiltaan ainoastaan web-selaimilla tapahtuvaan testaukseen. Selenium tarjoaa käyttäjälle mahdollisuuden skriptikielien käyttämisen lisäksi, myös nauhoittaa toiminnot, joita halutaan tehdä, jolloin käyttäjän ei tarvitse osata ohjelmoida.
(Selenium history 2017.)

Squish

Squish on Frologic-nimisen yrityksen kehittämä ja ylläpitämä maksullinen UI-testaukseen käytettävä ohjelmistokehys. Versio 1.0 työkalusta julkaistiin vuonna 2003. Squish tukee kaikkia yleisiä työpöytä- ja mobiilikäyttöjärjestelmiä. Ohjelmointi- ja skriptikielistä Squish tukee: JavaScript, Perl, Python, Ruby ja Tcl. Squishin testausympäristö koostuu kahdesta komponentista, suorittajasta ja serveristä, joka kontrolloi testauksen alaista sovellusta. (Automated GUI testing for Windows applications 2018.)

Squishin Windows-version ominaisuudet:

- ei pinta-automaatiota
- tukee natiiveja Windows-applikaatioita
- tukee standardeja ja kompleksisia kontrolleja standardeissa Windows, MFC (Microsoft Foundation Class library), .NET forms, WPF (Windows Presentation Foundation) -applikaatioissa
- Excel tuki, joka mahdollistaa mm. työkirjojen ja solujen käsittelyn
- applikaatioiden etätestaus
- useamman sovelluksen samanaikainen testaus.

(Automated GUI testing for Windows applications 2018.)

WorkFusion

WorkFusion tai tarkemmin kutsuttuna WorkFusion RPA express on ilmainen ohjelmisto, jolla voidaan automatisoida web-, työpöytä- tai terminaalisovelluksia. WorkFusion tarjoaa myös kattavan valikoiman erilaisia valvonta- ja analytiikkatyökaluja. WorkFusionia kuvataan TIVI:n artikkelissa raskaaksi käyttää ja suositellaan lähinnä isompien organisaatioiden käyttöön.
(Laitila 2018, 56—58.)

WorkFusion sisältää graafisen käyttöliittymän, jossa voidaan suunnitella testi kaavamaiseen tapaan. WorkFusion perustuu pinta-automaation kuten SikuliX.
(Laitila 2018, 56—58.)

WorkFusionin ominaisuudet ovat:

- ilmainen käyttää
- ei avointa lähdekoodia
- pinta-automaatio
- oppiva automaatio.

(Laitila 2018, 56—58.)

UIPath

UIPath on RPA (robotic process automation) -työkalu, jolla voidaan WorkFusionin tapaan automatisoida web-, työpöytä- tai terminaalisovelluksia. Työpöytäautomaatiossa UIPath on yksi markkinajohtajista. Toisin kuin WorkFusion ja SikuliX, UIpath ei perustu pelkkään pinta-automaatioon vaan pyrkii kontrolloimaan sovelluksia pintaa syvemmältä. Pinta-automaatiota käytetään vain, mikäli törmätään elementtiin, mitä ei voida tunnistaa. UIPath sisältää myös graafisen käyttöliittymän, jossa voidaan suunnitella testit vuokaaviomaiseen tapaan. (Laitila 2018, 56—58.)

UIPathin ominaisuudet ovat:

- pinta-automaatio jota käytetään API -tason kutsujen tukena
- API -tason kutsut
- automaatio oppii mitä enemmän se tekee
- kaupallinen
- ei avointa lähdekoodia

(Laitila 2018, 56—58.)

SikuliX

SikuliX on avoimen lähdekoodin erityisesti pinta-automaatioon fokusoitunut projekti, joka lähti alun perin liikkeelle vuonna 2009 MIT -opiskelijoiden projektista. Automaatioprojekti käyttää kuvantunnistukseen laajasti käytettyä OpenCV-rajapintaa, joka sisältää yli 2500 optimoitua algoritmia kuvan, mallien ja hahmojen tunnistukseen. SikuliX tukee myös tekstintunnistusta, johon käytetään Tesseract OCR (Optical Character Recognition) -ohjelmistomoottoria. SikuliX on yksi tunnetuimmista testiautomaatioprojekteista ja saavuttanut suosion ilmaisuudellaan. (What is Sikulix 2017.) SikuliX käsitellään laajemmin seuraavassa luvussa.

SikuliX ominaisuudet:

- pinta-automaatio
- tekstintunnistus
- ilmainen
- voidaan pakata erilliseen Java -sovelmaan, jota voidaan kutsua ohjelmallisesti

- avoin lähdekoodi
- tuki usealle eri skripti- ja ohjelmointikielelle kuten Python
- hiiri- ja näppäimistösyötteiden simulointi
- saavuttanut laajan käyttäjäkunnan ilmaisuudellaan
- nykyinen kehittäjäporukka on aktiivinen ja pääkehittäjä vastaa itse mieluusti kysymyksiin.

4.3 Automatisointiohjelmistokehyksen valinta

Automatisointiohjelmistokehyksen valintaan käytettiin puhtaasti luvussa 4.1 esitettyjä vaatimuksia. Vaatimukset perustuivat toimeksiantajan kuvailemiin tarpeisiin.

	PyWinAuto	AutoIT	Selenium	Squish	Workfusion	Uipath	SikuliX
Pinta-automaatio					X	X	X
Windows-tuki	X	X	X	X	X	X	X
Siirrettävyys		X			X	X	X
Kustannustehokkuus	X	X	X		X		X
Laajennettavuus	X	X		X	X	X	X
Helppokäyttöisyys		X		X	X	X	X
Tuettavuus		X	X	X	X	X	X
Luotettavuus	X	X	X	X	X	X	X
Muokattavuus	X		X				X
Simuloitavuus		X			X	X	X
Ylläpidettävyys		X			X	X	X
Soveltuu luotettavuus- ja suorituskykytestaukseen		X			X	X	X
Skaalautuvuus	X				X	X	X

TAULUKKO 1 Vertailu ohjelmistokehyksistä

SikuliX ainoana ohjelmistokehyksenä vastasi kaikkiin vaatimuksena oleviin kohtiin. WorkFusion ja UIpath ovat hyviä vaihtoehtoja, mutta ne eivät ole avoimen lähdekoodin projekteja. Tämän takia mahdollisissa ongelma- tai jatkokehitystilanteissa olisi riippuvainen näiden projektien tuesta, jolloin asioiden ratkaiseminen voi olla todella hidasta tai kallista.

5 SIKULIX

SikuliX on vuoden 2009 aikana aloitettu avoimen lähdekoodin tutkimusprojekti. Projektin aloittivat Tsung-Hsiang Chang sekä Tom Yeh. Projekti alkoi nimellä Sikuli, mutta myöhemmin vuonna 2012, kun alkuperäiset jäsenet lähtivät projektista, nimi muutettiin SikuliX:ksi ja tiimi vaihtui uuteen. Nykyisin projektia vetää saksalainen Raimund Hocke. (What is Sikulix? 2017.)

SikuliX:llä voidaan automatisoida mitä tahansa ruudulla näkyvää asiaa kuvantunnistuksen avulla. Sikulix käyttää kuvantunnistuksessa OpenCV (Open Source Computer Vision Library) -kirjastoa. Kuvantunnistukseen perustuvalla automatisaatiolla saadaan simuloitua oikean käyttäjän tekemiä liikkeitä graafisessa ympäristössä. (What is Sikulix? 2017.)

SikuliX tukee myös tekstin tunnistusta (OCR) ja sitä voidaan käyttää etsimään ruudulta tai kuvista tekstiä. Tekstintunnistuksella voidaan etsiä esimerkiksi tekstimuotoista sanaa "testi" laitteen ruudulta. Tekstintunnistamiseen SikuliX käyttää Tesseract-moottoria. (What is Sikulix? 2017.)

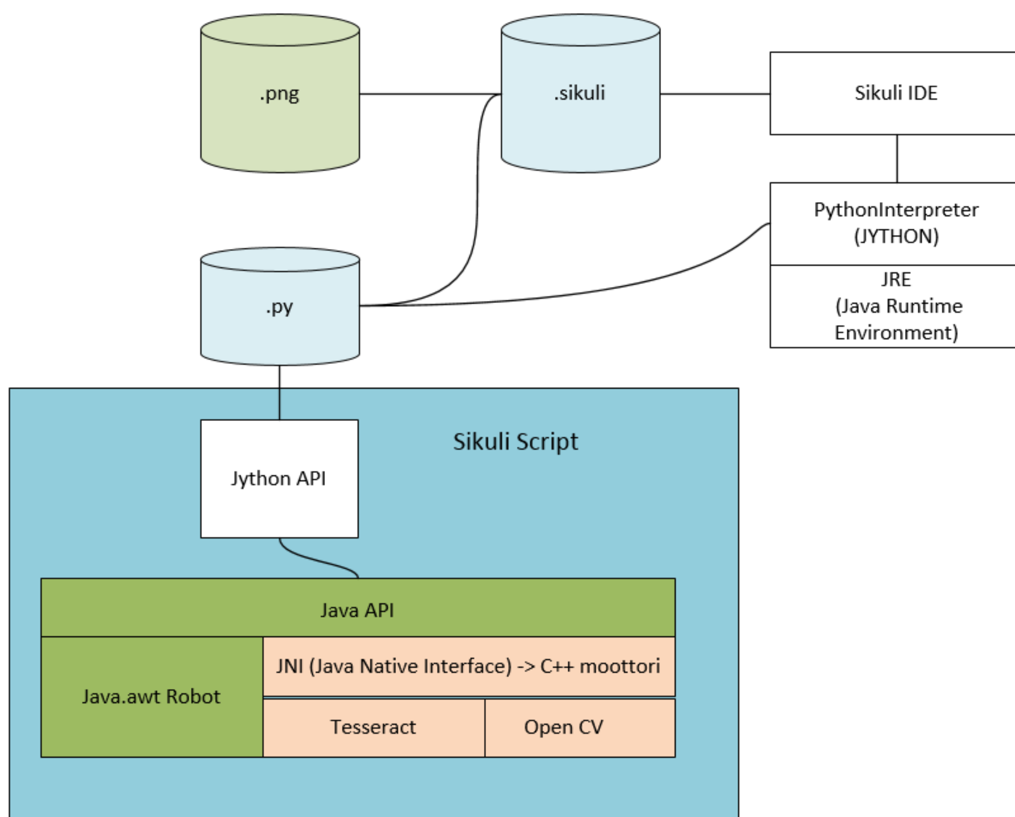
5.1 Käyttökohteet

SikuliX:llä voidaan automatisoida kaikkea ruudulla näkyvää. SikuliX mainostaa WYSIWYS "What You See Is What You Script" -filosofiaa. Käytännössä automaatiota voidaan käyttää kaikessa, missä perinteinen sovelluksen kontrollointi manuaalisesti on joko turhaa tai vaivalloista. Pinta-automaatioilla voidaan yleisesti automatisoida kaikkea tietokoneen ruudulla näkyvää lähes ilman rajoitteita. Tästä syystä myös usea verkkopalvelu on joutunut ottamaan pinta-automaatioita häiritsevän Captcha tunnisteen käyttöön, jolla pyritään varmistamaan, että käyttäjä on oikea ihminen. (What is Sikulix? 2017.)

5.2 Ympäristön toiminta

Lyhykäisyydessään SikuliX on Jython- ja Java-kirjasto, jolla voidaan automatisoida graafisessa käyttöliittymässä tapahtuvia näppäimistö- ja hiirieleitä joko suorilla komennoilla tai kuvantunnistukseen perustuvilla vertailuilla. (How SikuliX Works 2018.)

Sikuli Script on Java-kirjasto, joka koostuu kahdesta eri osasesta; java.awt.Robot -kirjastosta, joka käsittelee näppäimistön- ja hiirenliikkeet oikeisiin lokaatioihin, ja OpenCV:n perustuvasta C++ -moottorista, joka etsii haluttuja kuvioita ruudulta. C++ -moottori toimii Javan kanssa JNI:n (Java Native Interface) kautta. Java-kirjaston päälle on lopulta toteutettu Jython-kerros, jonka avulla loppukäyttäjät voivat antaa kokonaisuudelle helppoja ja selkeitä komentoja. Jython-kerroksen lisäksi kirjaston päälle voidaan lisätä tuki käytännössä mille tahansa ohjelmointikielelle. (How SikuliX Works 2018.) Kuva 6 kuvaa SikuliX:n toimintaa kaaviona.



KUVA 6 Kaaviokuva SikuliX:n toiminnasta

5.2.1 Jython

Jython on implementaatio Python kielestä Java-ympäristöön. Jython-sovellukset voivat tuoda ja käyttää mitä tahansa Java-luokkaa toteutuksessaan. Jython on Javan virtuaalikonetoteutus Python

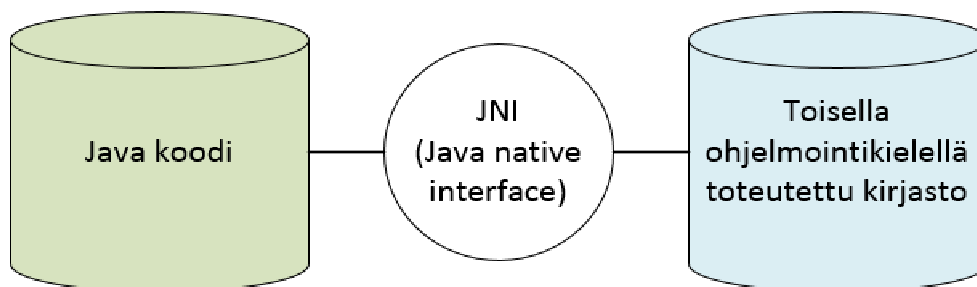
kielestä, joka on suunniteltu yhteensopivaksi Java-alustalla. Jython kehitettiin alkujaan CPythonin rinnalle vuonna 1997 laajentamaan tukea Java -ympäristöön. Jython -implementaatio seuraa tarkasti CPythonin rakennetta ja ne jakavat myös saman versionumeroinnin. Jython-ohjelmat voivat tuoda ja käyttää mitä tahansa Java-luokkaa. Jython-ohjelma käännetään bittikoodiksi (bytecode), kuten Java. (Jython 2017.)

5.2.2 Java.awt.Robot

Java-luokka, jolla voidaan antaa järjestelmälle natiiveja syötteitä. Luokkaa voidaan käyttää apuna muun muassa testiautomaatioissa tai sovelluksissa, joissa on tarvetta antaa hiiri- tai näppäimistösyötteitä järjestelmälle ohjelmallisesti. Robot-luokka on suunniteltu pääasiassa testiautomaatioiden tarpeisiin. (Class Robot 2018.) Esimerkki Java.awt.Robotin käyttämisestä puhtaalla Javalla löytyy liitteestä 3.

5.2.3 Java Native Interface

Java Native Interface eli JNI on ohjelmistokehys, joka mahdollistaa virtuaalikoneessa pyörivälle Java koodille kutsut sisään ja ulos järjestelmän natiiveihin applikaatioihin, sekä muihin kieliin kuten C, C++ ja Assembly (kuva 7). JNI:n avulla sovellus voi käyttää muita kieliä, kun ohjelmaa ei voida toteuttaa pelkästään Javalla tai esimerkiksi jokin valmis kirjasto käyttää toista kieltä. (JNI 2018.)



KUVA 7 Java-koodi voi keskustella eri kielillä toteutettujen kirjastojen kanssa JNI-rajapinnan kautta

5.2.4 OpenCV

OpenCV on avoimen lähdekoodin konenäkö- ja koneoppimisohjelmistokirjasto. OpenCV on rakennettu tarjoamaan yleisiä rakenteita konenäköä tarvitseviin ratkaisuihin, sekä kiihdyttämään tietokoneoppimisen tulemista kaupallisissa tuotteissa. (About OpenCV 2017.)

Kirjasto koostuu yli 2500 optimisoidusta algoritmista. Kirjasto sisältää algoritmeja mm. kasvojen, objektien ja ihmisten liikkeiden tunnistuksen. Kirjastoa käytetään valvonnassa, sotateollisuudessa, autoteollisuudessa ja monessa muussa. OpenCV tarjoaa valmiit C++-, C-, Python-, Java- ja MATLAB -rajapinnat ja tukee Windows-, Linux-, Android- sekä Mac OS -käyttöjärjestelmiä. (About OpenCV 2017.)

5.2.5 Tesseract

Tesseract on avoimen lähdekoodin OCR (Optical Character Recognition) -moottori. Sitä voidaan käyttää tunnistamaan käsin tai koneella kirjoitettua tekstiä kuvista. Tesseract on alun perin kehitetty Hewlett-Packard Laboratories Bristolin- ja Hewlett-Packard Co Greeley Coloradon -toimipisteiden yhteistyönä vuosien 1985 ja 1994 välillä. Vuonna 1996 moottori sai käännöksen Windows-alustalle ja paria vuotta myöhemmin käännöksen C++ -kielelle. 2005 HP (Hewlett-Packard) päätti muuttaa projektin avoimen lähdekoodin projektiksi. Vuodesta 2006 eteenpäin sitä on aktiivisesti kehittänyt Google. (Tesseract – Main Wiki page. 2017.)

Moottori tukee yli sataa puhuttua kieltä valmiina, mutta sitä voidaan myös opettaa tunnistamaan mitä tahansa kirjoitusta tarvittaessa. Tesseractilla ei ole sisäänrakennettua graafista käyttöliittymää, mutta niitä on saatavilla useita kolmansien osapuolien tekemänä. (Tesseract - About and history of project 2017.)

5.3 Käyttö ohjelmoinnissa

SikuliX:n ydin on kirjoitettu kokonaan Javalla. Tästä syystä SikuliX API:a voidaan käyttää normaalina Java -kirjastona missä tahansa nativisti kirjoitetussa ohjelmassa. SikuliX API:a voidaan hyödyntää myös millä tahansa skriptikielellä, mikä ymmärtää Javaa, kuten Jython, JRuby, Scala,

Groovy ja Clojure. Tällöin SikuliX voi olla laajentamassa yhtenä komponenttina laajempaa projektia.

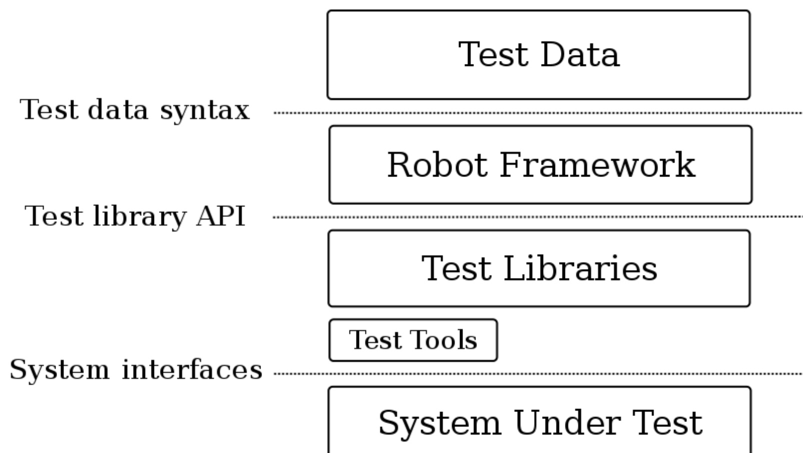
Työn aiheena ollut testiautomaatio hyödynsi SikuliX API:a SikuliLibrary:n muodossa, joka on Robot Frameworkille tehty avoimen lähdekoodin kirjasto. Kirjasto pitää sisällään SikuliX API:n, jolle voidaan antaa komentoja Robotin Frameworkin Remote -kirjaston pohjaisen toteutuksen avulla. SikuliLibrary sisältää serveritoteutuksen, joka käsittelee saadut Remote -toteutuksen komennot ja antaa ne edelleen SikuliX API:lle. Lisää SikuliLibrarystä ja toiminnasta luvussa 7.

6 ROBOT FRAMEWORK

Robot Framework on suomalaisen Pekka Klärckin alun perin gradutyöhönsä vuonna 2005 idean tasolla kehitetty geneerinen testiautomaatiokehys. Ensimmäinen toimiva versio näki päivänvalon Nokia Networks:llä samana vuonna. Robot Framework on sittemmin muuttunut open source -pohjaiseksi ja on kenen tahansa muokattavissa ja kehitettävissä. Pekka Klärck jatkaa edelleen projektin pääkehittäjänä. (Robot Framework introduction 2017.)

Robot Framework käyttää avainsanoihin perustuvaa testauksen lähestymistapaa ja sisältää liudan valmiita kirjastoja avainsanoineen. Kehys on suunniteltu modulaarisuutta silmällä pitäen ja sitä voidaan laajentaa omilla Python- tai Java -testikirjastoilla. Ohjelmistokehys on toteutettu kokonaisuudessaan Pythonilla. Kuva 8 kuvaa Robot Frameworkin arkkitehtuuria kaaviona. (Robot Framework introduction 2017.)

Testit kirjoitetaan tabulaarisella mallilla taulukkomaisesti. Testien kirjoittamiseen voidaan käyttää käytännössä mitä tahansa tekstinmuokkaukseen soveltuvaa ohjelmistoa tai tekniikkaa. Edellä mainitun lisäksi testien kirjoittamiseen ja ajamiseen voidaan käyttää RIDE:ä (Robot Integrated Development Environment). RIDE yksinkertaistaa testien kirjoittamisen ja suorittamisen tarjoamalla esimerkiksi testitapauksien kirjoittamiseen soveltuvaa taulukkomaista editoria. (Robot Framework introduction 2017.) Esimerkki Robot Framework testistä löytyy liitteestä 4.



KUVA 8 Robot Frameworkin arkkitehtuurikaavio (Robot Framework introduction 2017.)

6.1 Testidatan rakenne

Testidatan rakenne koostuu neljästä eri taulukkotyypistä. Nämä tyypit merkataan ensimmäiseen soluun tabulaarisessa mallissa. Tyypit ovat nimeltään Settings, Variables, Test Cases, Keywords. Tyypeillä on testidatan rakenteessa jokaiselle oma tehtävänsä (Robot Framework User Guide 2017.):

Settings
Sisältää määritykset lisättäville testikirjastoille, resurssitiedostoille sekä muuttujatiedostoille. Sisältää myös metadata-määritykset testisuitelle ja testitapauksille. (Robot Framework User Guide 2017.)

Variables
Sisältää määritykset muuttujille, joita voidaan käyttää missä tahansa testidatan sisällä. (Robot Framework User Guide 2017.)

Test Cases
Testitapaukset toteutetaan tämän alle. Testitapauksia luodaan käyttämällä avainsanoja joko valmiista tai itsetehdyistä kirjastoista. (Robot Framework User Guide 2017.)

Keywords
Tyyppi avainsanoja varten. Tämän alle voidaan toteuttaa omia avainsanoja, jotka voivat hyödyntää alemman tason avainsanoja tai kirjastoja. (Robot Framework User Guide 2017.)

6.2 Valmiit kirjastot

Robot Framework Core tuo mukanaan useita valmiita kirjastoja. Kirjastot sisältävät yleisimmin tarvittavia avainsanoja (keywordeja) valmiina toteutuksina. Kirjastojen avainsanat ovat kutsuttavissa sen jälkeen, kun kirjasto on tuotu suite-tiedostoon joko RIDE:llä visuaalisesti import-ominaisuudella tai kirjoittamalla *** Settings *** osion alle Library ja haluamasi kirjaston nimi. (Robot Framework introduction 2017.)

Valmiit kirjastot:
BuiltIn - Geneeriset usein tarvitut avainsanat. Tuodaan jokaiseen projektiin automaattisesti.
Collections - Sisältää avainsanat listojen käsittelyyn.
DateTime - Sisältää avainsanat päivän ja ajan määrittämiseen ja laskutoimitukseen päivien välillä.
Dialogs - Sisältää avainsanat dialogeille millä käyttäjältä voidaan ottaa tarvittaessa syötteitä.
OperatingSystem - Mahdollistaa useiden käyttöjärjestelmäkomentojen käytön, esim. ohjelmien suorituksen.
Process - Mahdollistaa prosessien luomisen ja ajamisen järjestelmässä.
Remote – Kirjasto, jossa ei ole lainkaan kutsuttavia valmiita avainsanoja. Remote -kirjasto sisältää mahdollisuuden muodostaa yhteys serveriin.
Screenshot - Tarjoaa avainsanoja ruutukaappauksien ottamiseen ja tallentamiseen.
String - Sisältää avainsanoja merkkijonojen manipuloimiseen ja sisällön verifioimiseen.
Telnet - Sisältää avainsanat Telnet -serveri yhteyksien luomiseen ja komentojen suorittamiseen.
XML - Sisältää avainsanat XML -dokumenttien muokkaamiseen ja verifioimiseen
(Robot Framework user guide 2017.)

6.3 Omat kirjastot

Robot Framework tukee käyttäjän itsensä kirjoittamia kirjastoja. Niitä voidaan lisätä suite-tiedostoon samalla tapaa, kuin robotin mukana tulleita kirjastoja. Kirjastot voidaan kirjoittaa natiivisti Python- tai Javakielillä. Java-kieli tarvitsee tosin toimiakseen sen, että Robot Frameworkia ajetaan Jython -profiililla. Kirjastoja voidaan tehdä myös käytännössä millä tahansa ohjelmointikielellä, mikä voidaan paketoida natiivisti toimivien kielien sisään. (Robot Framework introduction 2017.)

Pieni esimerkki Robot Framework -kirjastosta Python toteutuksella löytyy liitteestä 5.

6.4 Suoritettavat tiedostot

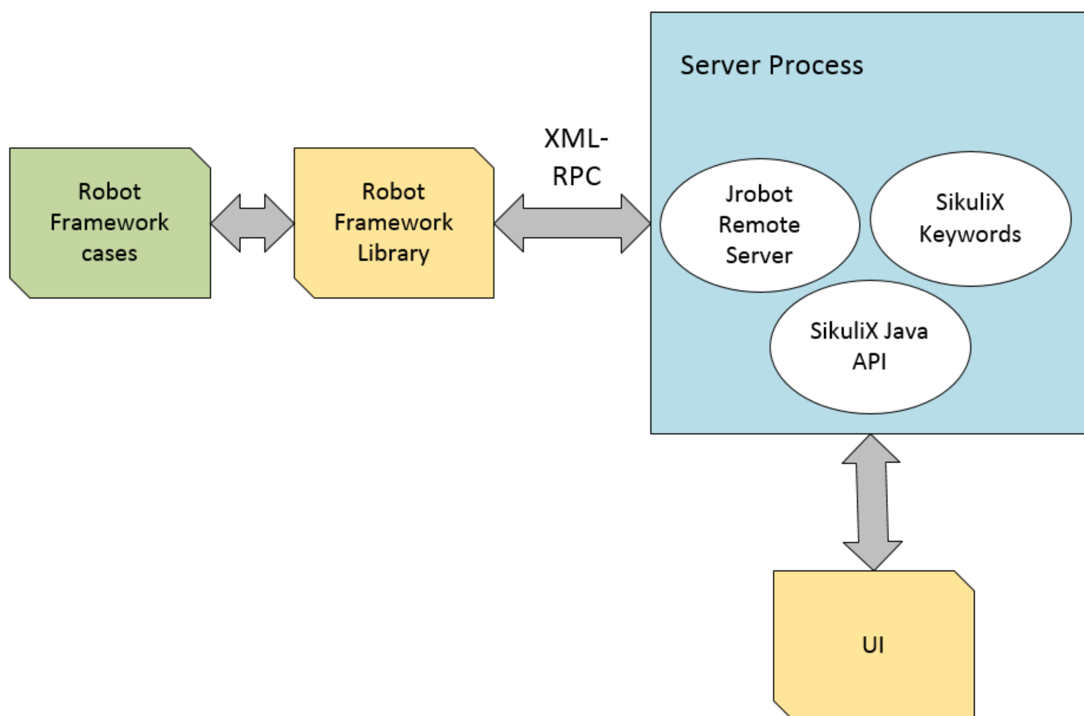
Robot Frameworkilla voidaan käynnistää ja ajaa käytännössä mitä tahansa tiedostoja mihin järjestelmässä on vain tulkki. Tämän takia Robot Framework -testitapauksia voidaan laajentaa helposti monimutkaisempaan testaukseen tai hyödyntämään esimerkiksi aikaisemmin luotuja automaatioita, jotka suoritetaan testin osana. Yleisimmin käytetyt skriptikielet toiminnallisuuden laajentamiseen ovat Robot Frameworkin kohdalla Python ja Ruby, jotka ovat siitä käteviä, että ne käännetään suorittaessa (Robot Framework User Guide 2017.). Esimerkki suoritettavasta skriptistä Python-kielellä löytyy liitteestä 2.

7 SIKULILIBRARY

SikuliLibrary on avoimen lähdekoodin projekti, joka on luotu mahdollistamaan käyttöliittymän testaamisen SikuliX API:n avulla Robot Frameworkista. Robot Framework toimii kokonaiskuvassa siis suorittajana, joka välittää kutsuja SikuliLibraryn kautta suoraa SikuliX API:lle. SikuliX API taas hallitsee syötelaitteiden hallitsemisen, pinta-automaation ja tekstintunnistuksen.

7.1 Toiminta

SikuliLibrary on toteutettu niin, että se lähettää ja vastaanottaa XML-RPC -komentoja, jotka lähetetään Robot Frameworkin Remote -kirjastolla tehtyä toteutusta hyödyntäen. Komennot vastaanotetaan itsenäisenä ajettavaan Java-server prosessiin, joka käsittelee saadut komennot ja kutsut edelleen SikuliX API:lle. SikuliX API toimii, kuten normaalikin SikuliX hyödyntäen OpenCV:tä, Tesseract:ia ja Java AWT Robot -luokkaa. (SikuliLibrary readme 2017.) Kuva 9 kuvaa kaaviona SikuliLibrary kokonaisuutta.



KUVA 9 SikuliLibrary -kaaviokuva

7.2 XML-RPC

RPC eli Remote Procedure Call on mekanismi, jolla voidaan kutsua aliohjelmaa tai funktiota toiselta tietokoneelta. RPC tarjoaa kehittäjille mekanismin, jolla voidaan määritellä verkon ylitse kutsuttavia rajapintoja. (XML-RPC 2017.)

XML-RPC -protokolla kehitettiin vuonna 1998 Microsoftin sekä UserLand Software yrityksessä työskennelleen Dave Winer:in toimesta. Microsoft oli projektissa mukana, koska näki sen olevan tarpeellinen sen yritys liiketoimintaan keskittyvissä teknologioissa.

XML-RPC -protokolla toimii lähettämällä XML koodatun viestin HTTP:n yli laitteeseen, jossa serveri toteutus vastaanottaa sen. Client -puolen sovellus voi lähettää XML koodatussa viestissään useita parametrejä, mutta vastaanottaa takaisin silti vain yhden arvon. (XML-RPC specification 2018.)

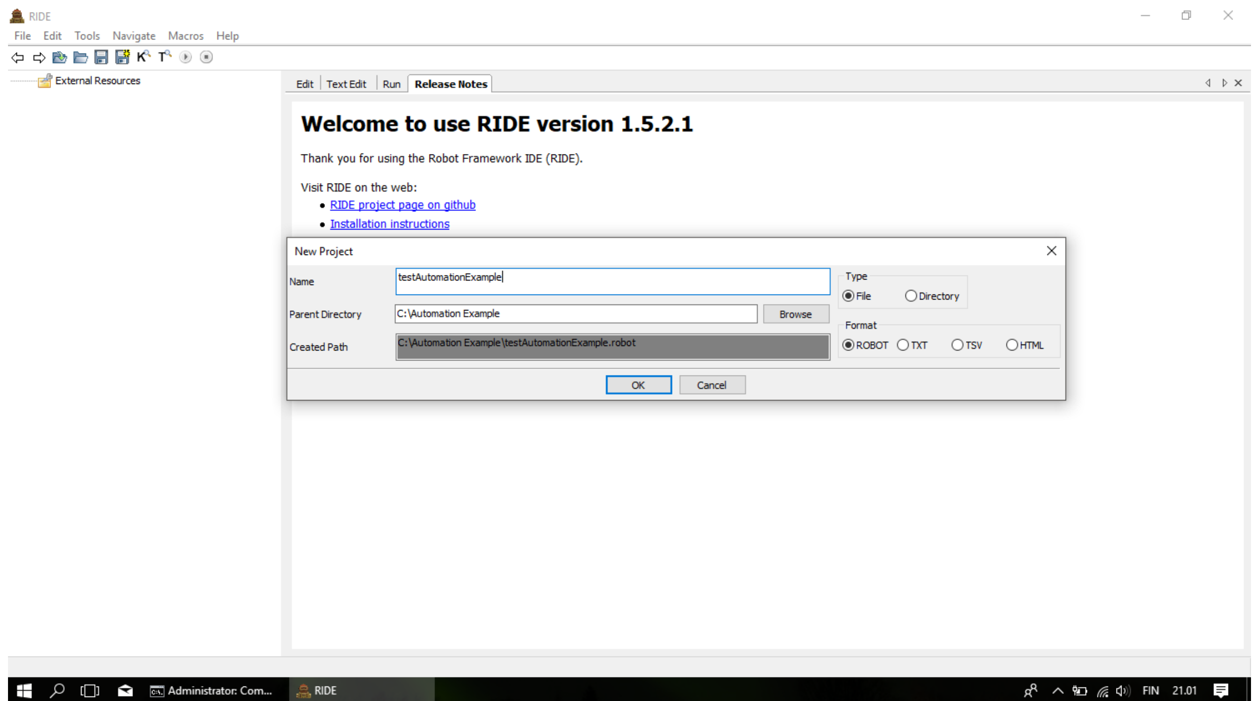
8 TESTIAUTOMAATION TOTEUTUS

Yritykselle toteutetun testiautomaation toiminnasta tai laitteista ei voitu salassapidon takia näyttää mitään, joten opinnäytetyö keskittyy näyttämään automaation toteutusta eri laite konfiguraatiolla sekä testitapauksilla. Automaatiosta ei voida mainita muuta kuin se, että se keskittyi luotettavuustestaukseen ja se koostuu yhteensä noin 15 testitapauksesta, joiden suorittaminen kestää vaatimuksista riippuen useamman viikon.

Testitapaukset toteutettiin Robot Framework:lla, jota pyöritettiin kontrollikoneella, josta taasen luotiin Robot Frameworkin Remote -kirjaston avulla yhteys testattavassa laitteessa pyörivälle serveriprosessille. Laitteiden väliin lisättiin ristiin kytkettävää RJ45 -kaapelia, jolloin laitteiden välille saatiin luotua yhteys. Kontrollikoneen valmisteluun kuului Python 2.7.X, wxPython 2.8.12.1, JDK 8, sekä Robot Framework ja sen RIDE -editorin asennus. Testattavan laitteen valmisteluun kuului Java -sovelman siirtäminen laitteeseen sekä SikulixAPI:n kuvantunnistuksen vaatimat kuvat. Idea kontrollikoneen ja erillisen testilaitteen välillä on se, että testien käynnistykseen menevä aika saadaan minimoitua sekä se, että testattavan laitteen ohjelmisto pysyy mahdollisimman muuttumattomana.

8.1 Testin toteutus

Testin toteuttaminen alkoi RIDE -editorin käynnistämisellä ja uuden projektin tekemisellä (kuva 11) RIDE ei ole pakollinen osa Robot Framework -ympäristöä ja testejä voidaankin toteuttaa millä tahansa tekstieditorilla. RIDE toimi tässä tapauksena kehitysympäristönä, koska se niputtaa helposti kehitys- ja suoritusympäristöt.



KUVA 11 Uusi projekti testitiedostoineen voidaan luoda painamalla File -> New Project, tai pikanäppäimellä CTRL+N

Testidata koostuu neljästä eri osasta, jotka ovat: Settings, Variables, Test cases ja Keywords. Ensimmäisenä projektiin kuvaillaan Settings alueen alle esimerkiksi se, mitä kirjastoja testeissä käytetään (kuva 12). Robot Framework -kirjastot pitävät sisällään yleisimmin käytettyjä avainsanoja valmiina toteutuksina.

```
1   *** Settings ***
2   Suite Setup     Start Sikuli Process    port=8270
3   Library         SikuliLibrary
4   Library         DateTime
5
```

KUVA 12 Settings alueen alle kuvataan testin tarvitsemat kirjastot

Variables pitää sisällään muuttuja määritykset (kuva 13), jotka ovat saatavilla missä tahansa testidatan sisällä. Muuttujia voidaan luoda myös missä tahansa testin tai avainsanan alueella, mutta mikäli niitä haluaa käyttää muualla testissä, niitä pitää kuljettaa ja vastaanottaa argumentteina jatkuvasti.

```
6   *** Variables ***
7   ${imagePath}     C:\\img
```

KUVA 13 Variables -alueelle tehdään muuttujamääritykset, jotka ovat saatavilla koko testitiedoston alueella

Testitapaukset kuvataan Test cases -osion alle (kuva 14). Testitapaukset kootaan joko valmiiden kirjastojen tarjoamista avainsanoista, omista avainsanakirjastoista tai sitten suoraan testisuitetiedostoon toteutettujen avainsanojen avulla.

```
9   *** Test Cases ***
10  Calculator example
11      Set Min Similarity    0.98
12      Add Image Path    ${imagePath}
13      Open calculator
14      Check calculation result
```

KUVA 14 Testitapaukset kuvataan test cases -alueen alle

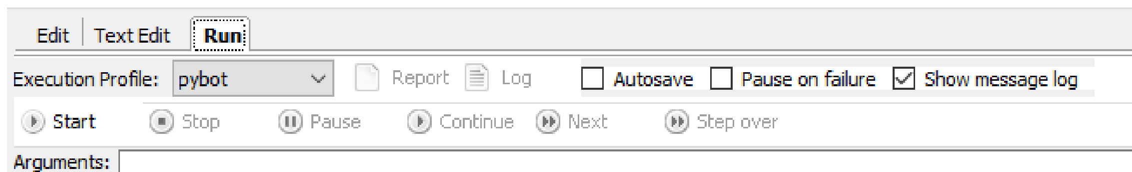
Avainsanat voidaan kuvata erillisiin kirjastotiedostoihin tai suoraa testitiedostoon. Keywords -alueen alle voidaan tehdä toteutuksena avainsanoja (kuva 15), jotka voivat yhdistellä muita avainsanoja tai toteuttaa täysin uuden toiminnallisuuden.

```
16  Ping stability example
17
18  *** Keywords ***
19  Open calculator
20      Log to console    Opening calculator ${\n}
21      Press Special Key    WIN
22      Input text    \    calc
23      Press Special Key    ENTER
24      ${doesCalculatorExists}=    Exists    calculator_title.png    10
25      Log to console    Does calculator Exists: ${doesCalculatorExists}
26      Run keyword if    '${doesCalculatorExists}==True'    Count to 10
27
28  Count to 10
29      Log to console    Counting to 10 ${\n}
30      Click    button_5.png
31      Click    button_add.png
32      Click    button_5.png
33      Click    button_equal.png
34
35  Check calculation result
36      Log to console    Checking calculation result ${\n}
37      ${doesResultMatch}=    Exists    calculation_result.png    10
38      Log to console    Does Result Match: ${doesResultMatch}
39      Run keyword if    '${doesResultMatch}==True'    Log    Calculation result matches. Test PASS.
```

KUVA 15 Kuvassa esimerkkitestin avainsanoja, jotka koostuvat sekalaisesti SikuliLibraryn ja Robot Frameworkin sisäänrakennetuista avainsanoista

8.2 Testitapauksien suorittaminen

Esimerkkitestitapaus pitää sisällään laskimen avauksen, laskutoimituksen tekemisen ja lopputuloksen tarkastamisen. Toteutuksen jälkeen testi voidaan käynnistää ruksaamalla valinta halutun testitapauksen eteen ja siirtymällä näkymässä RUN -välilehdelle (kuva 16)



KUVA 16 Testitapaukset voidaan ajaa testin valitsemisen jälkeen siirtymällä RUN -välilehdelle ja painamalla Start

Testi suoritetaan, jolloin konsolinäkymä näyttää testissä tapahtuvat asiat. Konsolinäkymä jakautuu kahteen osaan, jossa ylempään kirjoitetaan testin statuksiin tai tehtyihin viesteihin perustuvat merkinnät ja alempaan kirjoitetaan jatkuvasti mitä testissä tapahtuu tällä hetkellä ja mahdollisesti myös sen, että mikä testissä meni pieleen.

Kun testitapaus on suoritettu, lokeista voidaan nähdä testin eteneminen vaiheittain (kuva 17). SikuliLibrary tukee myös lokien tekoa niin, että nappaa ruuduntunnistuksella jatkuvasti pieniä kuvia ja säilöö ne lokitiedoston kanssa samaan kansioon, jolloin kirjastoon liittyvät avainsanat näyttävät kuvan lokimerkinnän yhteydessä.

KUVA 17 Testitapauksen suorituksen jälkeen lokeista voidaan nähdä mitä testissä tapahtui. Mikäli testissä olisi virhe, auttaa lokien tutkiminen vikatilanteen selvittelyä

9 YHTEENVETO

Tämän työn tavoitteena oli tutkia erilaisia testausympäristöjä ja toteuttaa testiautomaatio Windows-alustan testaukseen. Testiautomaation tavoitteena oli automatisoida ohjelmistotestauksesta luotettavuustestaus ja osa suorituskykytestauksesta, jotka veivät paljon henkilöresursseja muusta testauksesta. Luotettavuustestauksessa myös samanlaisten toistojen määrän takia ihmisen suorittamana lopputulos olisi todennäköisesti aina erilainen. Työ esitteli joukon ohjelmistokehyksiä ja kertoi sitten lisää valituista ohjelmistokehyksistä.

Lopputuloksensa saavutettiin erilaisten testausympäristöjen vertailu, jonka jälkeen rakennettiin testiautomaatio kokonaisuudessaan. Testausympäristöksi valikoitui SikuliX API, jota käytettiin erilliseen Java-sovelmaan paketoituna (SikuliLibrary). Paketoitua Java-sovelmaa voitiin kutsua Robot Frameworkin puolelta Remote-kirjastolla tehdyn toteutuksen kutsujen avulla. SikuliX API valikoitui testausympäristön toteutukseen, koska se oli ainoa ympäristö, joka vastasi jokaiseen esitettyyn vaatimukseen. Testiautomaation rakentamista ja toimintaa ei voitu työssä esitellä, koska se oli salaiseksi luokiteltua, mutta automaatio kuitenkin toteutettiin kokonaisuudessaan ja se mahdollistaa automaattisen luotettavuustestauksen Windows-käyttöjärjestelmällä.

Testiautomaation toteutusvaiheessa nousi useampi ongelma, joihin ei pystytty etukäteen varautumaan. Ongelmiksi ilmeni mm. kuvantunnistuksen skaalautuvuus käyttöjärjestelmässä erilaisilla laitteilla testattuna. Windows skaalaa käyttöliittymän kokonaisuudessaan jokaiselle laitteelle ja resoluutiolle sopivaksi. Tästä syystä pinta-automaation tarvitsemat kuvat jouduttiin ja tulevaisuudessakin joudutaan ottamaan useaan otteeseen uudestaan eri laitteille ja resoluutioille. Ongelma saatiin ratkaistua tekemällä automaatiosta dynaamisempi, mutta se lisää silti osittain isojen päivityksien tullessa ylläpidon määrää. Ongelma sai myös osittain jatkoa Windowsin tuodessa Windows 10 -käyttöjärjestelmäänsä syksyllä 2017 Redstone 3 -päivityspaketin. Redstone 3 toi käyttöjärjestelmään DPI-skaalausta parantavan päivityksen, joka omalta osaltaan rikkoi korkeanresoluution laitteissa pintatunnistuksen ja käytössä olevan Java-version syötteiden toimivuuden. Tämäkin ongelma saatiin osittain kierrettyä, mutta ongelman perusteellinen korjaus on tulossa vasta tuleviin Java JDK-versioihin.

Toteutusvaiheessa automaation kehityksessä törmättiin myös kohdistus ongelmaan, joka aiheutti sen, että käyttöliittymän ikkunoiden sulkeutuessa syötteiden kohdistus hävisi tyhjyyteen. Ongelma

vaikutti kohdistuksen häviämisen jälkeisiin syötteisiin niin, että niitä ei voitu antaa laitteelle oikein. Ongelma saatiin kierrettyä luotettavuustestauksessa niin, että taka-alalle avataan turha ikkuna, joka nappaa kohdistuksen itseensä.

Testiautomaatio opinnäytetyönä oli haastava, koska aihe oli todella laaja ja se vaati paljon kokeilua ja perehtymistä. Lisäksi kehitystyötä hidasti myös muu testaustyö, jota piti tehdä samanaikaisesti. Testiautomaation kehityksen aikana tuli kohdattua useita haasteita, joiden ratkaisemiseen piti käyttää osaamista Pythonista, Javasta ja pinta-automaation logiikasta ja ylipäätään laajaa tuntemusta Windows-alustan toiminnasta.

Toteutettua testiautomaatiota voitaisiin kehittää vielä lisäämällä useamman testilaitteen samanaikainen tuki. Tällä hetkellä testaus onnistuu vain yhdellä laitteella kohti Robot Framework -istuntoa, jolloin laajemman luotettavuustestauksen lopputulosten saanti hidastuu. Järjestelmää voitaisiin kehittää myös paremmin prosessoimaan kerättyä dataa niin, että manuaalinen tulosten läpikäynti voitaisiin pienentää minimiin. Testiautomaatio on myös toteutettu tällä hetkellä ainoastaan Windows-käyttöjärjestelmälle sopivaksi ja sitä voitaisiin jatkokehittää tekemällä siitä Androidille yhteensopiva.

Testiautomaation soveltuvuus yrityksen käyttöön oli hyvä ja sitä käytetään luotettavuustestauksessa. Ympäristö soveltuu myös muuhun ohjelmistotestaukseen, kuten suorituskykymittauksiin, jotka toimivat tämän työn jatkumona luotettavuustestauksen automatisoinnin jälkeen. Erillinen SikuliX toimii hyvin myös pienempien tehtävien automatisointiin ja sitä voidaan käyttää hyvin usein toistuvien sekä monotonisten testitapauksien automatisointiin. Tällaisia testitapauksia ovat mm. tiedostojen siirto käyttöjärjestelmässä tai käyttöliittymän asetusliukujen toistuva päälle ja pois laitto.

LÄHTEET

About OpenCV. 2017. Saatavissa: https://opencv.org/about.html. Hakupäivä: 18.10.2017

AutoIT overview. 2017. Saatavissa: https://www.autoitscript.com/site/autoit/. Hakupäivä: 18.10.2017

Automated GUI testing for Windows applications. 2018. Saatavissa: https://www.froglogic.com/squish/editions/automated-windows-gui-testing/. Hakupäivä: 10.1.2018

Class Robot. 2018. Saatavissa: https://docs.oracle.com/javase/7/docs/api/java/awt/Robot.html. Hakupäivä 27.1.2018

Destructive testing. 2017. Saatavissa: https://www.tutorialspoint.com/software_testing_dictionary/destructive_testing.htm. Hakupäivä: 16.10.2017

Eichberg, Michael. 2018. Software process models. Saatavilla: https://stg-tud.github.io/eise/WS11-EiSE-12-Software_Process_Models.pdf. Hakupäivä: 17.7.2018

Eriksson, Ulf. 2016a. Grey box testing. Reqtest. Saatavilla: https://reqtest.com/testing-blog/grey-box-testing/. Hakupäivä: 17.7.2018

Eriksson, Ulf. 2016b. Sanity testing. Reqtest. Saatavilla: https://reqtest.com/testing-blog/sanity-testing-2/. Hakupäivä: 17.7.2018

Haikala, Ilkka – Märijärvi, Jukka 2004. Ohjelmistotuotanto. 10. painos. Helsinki: Talentum.

Hangasmaa, Heikki 2010. Käytettävyyden merkitys ja käytettävyystestaus. AMK Lopputyö. Vaasan Ammattikorkeakoulu. Tietojenkäsittelyn koulutusohjelma. Saatavilla: http://urn.fi/URN:NBN:fi:amk-210062112421.

Holopainen, Juha 2005. Regressiotestaus ja testien valintatekniikat. Pro gradu-tutkielma. Kuopion yliopiston tietojenkäsittelytieteen laitos. Tietojenkäsittelytiede. Saatavissa: http://www.cs.uku.fi/tutkimus/sose/material/Regressiotestaus.pdf.

How SikuliX Works. 2018. SikuliX Guide. Saatavissa: http://sikulix-2014.readthedocs.io/en/latest/devs/system-design.html. Hakupäivä 26.1.2018

JNI. 2018. Oracle docs. Saatavissa: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html#wp9502. Hakupäivä: 27.1.2018

Kaner, Cem – Bach, James – Pettichord, Bret 2002. Lessons learned in software testing. John Wiley & Sons, Inc New York.

Katara, Mika. 2011. Ohjelmistojen testaus – Hyväksymistestaus. Luentomateriaali. Saatavissa: www.cs.tut.fi/~tie21201/s2011/luennot/OHJ-3060_2011_110-170.pdf. Hakupäivä: 26.5.2018

Kautto, Tuomas 1996. Ohjelmistotestaus ja siinä käytettävät työkalut. Seminaarimateriaali. Saatavissa: http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus/. Hakupäivä: 17.10.2017

Laitila, Teemu 2018. Ulkoista rutiinit robotille. TIVI 02/18. S. 56-58

Maritta, Anne. 2008. Ohjelmistotuotannon mallit – Spiraalimalli. Saatavissa: https://hlab.ee.tut.fi/hmopetus/vpsist-oppimateriaali/4-menetelmia-ja-malleja/4-3-suunnittelumalleja/4-3-1-ohjelmistotuotannon-malli.html. Hakupäivä: 26.5.2018

Miten käyttöliittymä testataan automaattisesti. 2018. Develore. Saatavissa: http://www.develore.com/artikkeli/miten-kayttoliittyma-testataan-automaattisesti/. Hakupäivä: 10.1.2018

Potinkara, Rami 2017-2018. R&D Manager, Aava Mobile Oy. Haastattelut ja katselmoinnit välillä syksy 2017- kevät 2018

Pyhäjärvi, Maaret 2006. Ohjelmistojen testaus. Saatavissa: http://users.jyu.fi/~kolli/testaus2006/materiaali/Maaret_27102006.pdf. Hakupäivä: 19.11.2017

Reliability testing. 2004. Weibull. Saatavilla: https://weibull.com/hotwire/issue41/hottopics41.htm. Hakupäivä: 10.8.2018

Robot Framework introduction. 2017. Robot Framework. Saatavissa: http://robotframework.org/#introduction. Hakupäivä: 18.10.2017

Robot Framework User Guide 2017. Robot Framework. Saatavissa: http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html. Hakupäivä: 29.10.2017

Selenium history. 2017. Selenium. Saatavissa: https://www.seleniumhq.org/about/history.jsp. Hakupäivä: 18.10.2017

SikuliLibrary readme. 2017. SikuliLibrary. Saatavissa: https://github.com/rainmanwy/robotframework-SikuliLibrary/blob/master/README.md. Hakupäivä: 17.10.2017

Software testing lifecycle. 2018. Etestinghub. Saatavissa: http://www.etestinghub.com/testing_lifecycles.php#2. Hakupäivä: 10.1.2018

Software testing overview. 2018. Tutorialspoint. Saatavilla: https://www.tutorialspoint.com/software_testing/software_testing_overview.htm. Hakupäivä: 17.7.2018

Tesseract - About and history of project. 2017. Tesseract Saatavissa: https://github.com/tesseract-ocr/tesseract. Hakupäivä: 18.10.2017

Tesseract - Main Wiki page. 2017. Tesseract. Saatavissa: https://github.com/tesseract-ocr/tesseract/wiki. Hakupäivä: 18.10.2017

Testauksen tasot. 2017. Smart Education. Saatavissa: http://smarteducation.jyu.fi/projektit/systech/Periaatteet/suunnittelun-periaatteet/testaus/testauksen-tasot. Hakupäivä: 17.10.2017

Tuovinen, Antti-Pekka 2013. Staattinen testaus. Saatavissa: https://www.cs.helsinki.fi/u/aptuovin/testaus/Ohj_testaus_2013_5.pdf. Hakupäivä: 29.10.2017

Jython. 2017. Tutorialspoint. Saatavissa https://www.tutorialspoint.com/jython/index.htm. Hakupäivä: 18.5.2017

What is performance testing? 2018. Softwaretestingclass. Saatavilla: https://softwaretestingclass.com/what-is-performance-testing/. Hakupäivä: 8.8.2018

What is pywinauto. 2018. Pywinauto. Saatavilla: https://pywinauto.readthedocs.io/en/latest/. Hakupäivä: 30.9.2018

What is security testing: Complete Tutorial. 2018. Guru99. Saatavilla: https://www.guru99.com/what-is-security-testing.html. Hakupäivä: 17.7.2018

What is Sikulix? 2017. SikuliX. Saatavissa: http://sikulix.com/#home1. Hakupäivä: 16.10.2017

V-mallin mukainen testausmenetelmä. 2002. Confuse. Saatavissa: http://www.soberit.hut.fi/T-76.115/01-02/palautukset/groups/Confuse/t3/docs/vmalli/vmalli.pdf. Hakupäivä: 22.11.2017

XML-RPC specification. 2018. XML-RPC. Saatavissa: http://xmlrpc.scripting.com/spec.html. Hakupäivä: 18.5.2018

XML-RPC. 2018. Tutorialspoint. Saatavissa: https://www.tutorialspoint.com/xml-rpc/xml_rpc_intro.htm. Hakupäivä: 29.10.2017

Y2K Bug. 2018. Encyclopedia Britannica. Saatavissa: https://www.britannica.com/technology/Y2K-bug. Hakupäivä: 15.5.2018

AUTOIT ESIMERKKI LIITE 1

Esimerkki joka kertoo, onko nykyinen vuosi karkausvuosi vai ei. Automaatio käyttää hyödykseen Windowsista löytyvää laskinta.

```
#include <MsgBoxConstants.au3>
; Näyttää 6 sekunnin mittaisen viestilaatikon.
MsgBox($MB_OK, "Attention", "Avoid touching the keyboard or mouse during automation.", 6)
; Suorittaa Windowsin laskimen.
Run("calc.exe")
; Odota että laskin tulee aktiiviseksi 10 sekunnin timeoutilla.WinWaitActive("[CLASS:CalcFrame]", "", 10)
; Jos laskin ei ilmaantunut 10 sekunnissa poistu skriptistä.
If WinExists("[CLASS:CalcFrame]") = 0 Then Exit
; Syötä laskimeen nykyinen vuosi.
Send(@YEAR)
; Nukutaan 600 millisekuntia.
Sleep(600)
; Syötä /4 = jaa neljällä, nuku 600 millisekuntia.
Send("/4")Sleep(600)
; Paina entteriä ja nuku 600 millisekuntia.
Send("{ENTER}")Sleep(600)
; Kopioi tulos leikepöydälle ctrl+c.
Send("^c")
; Siirrä leikepöydän tulos muuttujaan.
Local $fResult = ClipGet()
; Tarkista onko tuloksessa desimaali pistettä vai ei.
If StringInStr($fResult, ".") Then
	; Tämä vuosi ei ole karkausvuosi.
	MsgBox($MB_OK, "Leap Year", @YEAR & " is not a leap year.", 5)
Else
	; Tämä vuosi on karkausvuosi.	MsgBox($MB_OK, "Leap Year", @YEAR & " is a leap year.", 5)
EndIf
; Sulje laskin.
WinClose("[CLASS:CalcFrame]")
```

Esimerkksi suoritettavasta Python-skriptistä

```
from robot.libraries.Remote import Remote
import time
remote = Remote('http://169.254.100.100:8270')
remote.get_keyword_names()
remote.run_keyword("Add Image Path", ["C:\\img"], {})
time.sleep(1)
remote.run_keyword("Press Special Key", ["WIN"], {})
time.sleep(1)
remote.run_keyword("Paste Text", [], {"calc"})
remote.run_keyword("Press Special Key", ["ENTER"], {})
time.sleep(2)
remote.run_keyword("Click", ["number1.png"], {})
remote.run_keyword("Click", ["add.png"], {})
remote.run_keyword("Click", ["number5.png"], {})
remote.run_keyword("Click", ["multiply.png"], {})
remote.run_keyword("Click", ["number6.png"], {})
remote.run_keyword("Click", ["equals.png"], {})
```

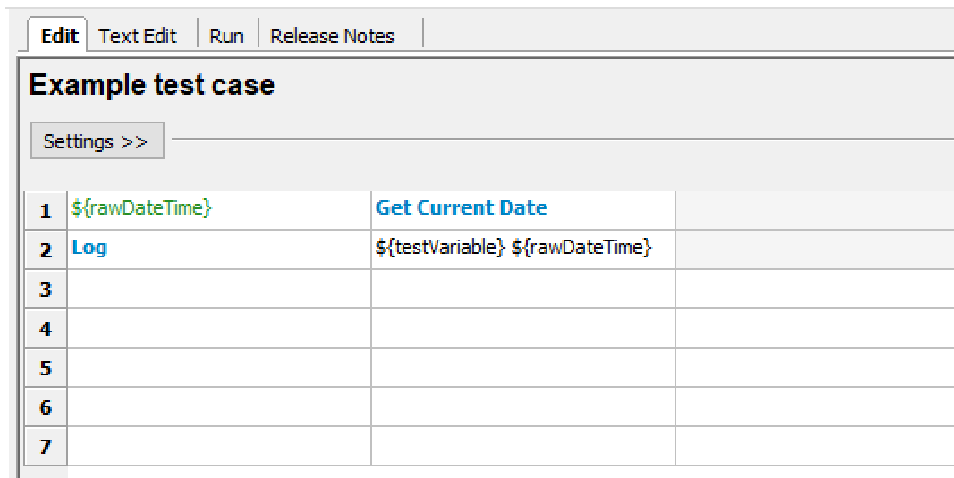
Esimerkissä hyödynnetään Robot Frameworkin Remote kirjastoa, jota käytetään XML-RPC yhteyden tekemiseen. Kirjastolla voidaan syöttää XML koodauksella HTTP:n yli toiselle laitteelle komentoja. Toisella laitteella täytyy olla serveri joka vastaanottaa komennot ja tekee suoritukset. Lisää XML-RPC protokollasta löytyy luvusta: 6.2

Esimerkki suorittaa laskutoimituksen Windowsista löytyvällä laskin -sovelluksella. Automaatio tarvitsee myös etukäteen otettuna kuvat painikkeista joita automaation pitää painaa.

```
import java.awt.AWTException;
import java.awt.Robot;
import java.awt.event.KeyEvent;
import java.io.*;

public class robo
{
  public static void main(String[] args) throws IOException,
                      AWTException, InterruptedException
  {
    String command = "notepad.exe";
    Runtime run = Runtime.getRuntime();
    run.exec(command);
    try {
      Thread.sleep(2000);
    }
    catch (InterruptedException e)
    {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

    // Create an instance of Robot class
    Robot robot = new Robot();

    // Press keys using robot. A gap of
    // of 500 mili seconds is added after
    // every key press
    robot.keyPress(KeyEvent.VK_H);
    Thread.sleep(500);
    robot.keyPress(KeyEvent.VK_E);
    Thread.sleep(500);
```

```
        robot.keyPress(KeyEvent.VK_L);
        Thread.sleep(500);
        robot.keyPress(KeyEvent.VK_L);
        Thread.sleep(500);
        robot.keyPress(KeyEvent.VK_O);
        Thread.sleep(500);
        robot.keyPress(KeyEvent.VK_SPACE);
        Thread.sleep(500);
        robot.keyPress(KeyEvent.VK_W);
        Thread.sleep(500);
        robot.keyPress(KeyEvent.VK_O);
        Thread.sleep(500);
        robot.keyPress(KeyEvent.VK_R);
        Thread.sleep(500);
        robot.keyPress(KeyEvent.VK_L);
        Thread.sleep(500);
    robot.keyPress(KeyEvent.VK_D);
        Thread.sleep(500);
    }
}
```

```
*** Settings ***
Library          DateTime

*** Variables ***
${testVariable}    testVariable contains this text

*** Test Cases ***
Example test case
    ${rawDateTime}    Get Current Date
    Log    ${testVariable} ${rawDateTime}
```



Testin kuvaus tabulaarisesti esitettynä

Testi määrittelee aloittaessaan muuttujaan tekstin ”testVariable contains this text”, jonka jälkeen siirtyy itse testiin. Testi hakee muuttujaan sen hetkisen päivä ja kellonaika tiedon, jonka jälkeen kirjoittaa lokitiedostoon: ”testVariable contains this text” + sen hetkisen päivä ja kellonaikatiedon.

```
def test_keyword_function():
    print "Hello World"
```

```
*** Settings ***
Library         exampleLibrary.py

*** Test Cases ***
Example test case
        Test keyword function
```

Result logiin printataan "hello world" joka on määritelty testikirjastossa.