Tampereen yliopisto

Kalle Nyman

ROBOT FRAMEWORKIN YHDISTÄMINEN QML-OHJELMISTOON

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
03/2022

TIIVISTELMÄ

Kalle Nyman: Robot Frameworkin yhdistäminen QML-ohjelmistoon
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma, tietotekniikka
03/2022

Tämän kandidaatintutkielman tavoitteena oli yhdistää Robot Framework osaksi QML-sovellusta. Käyttöliittymätestien automatisointi on Qt-sovelluksissa melko heikosti tutkittu aihe, ja tämän tutkielman tarkoituksena oli esitellä eräs tapa testiautomaation luomiselle käyttäen pelkästään avoimen lähdekoodin työkaluja.

Ohjelmistoja testattaessa pyritään varmistumaan ohjelmiston oikeanlaisesta toiminnasta tai löytämään siitä virheitä. Testausta tehdään kaikille osille ohjelmistoa, esimerkiksi ohjelman graafiselle käyttöliittymälle. Käyttöliittymän testaaminen on tärkeää, vaikkakin usein toistuvaa työtä, joka on mahdollista automatisoida suurilta osin. Käyttöliittymätestauksen automatisointiin löytyy monenlaisia työkaluja. Yksi suosituimmista ja arvostetuimmista työkaluista on Robot Framework, muun muassa sillä tehtyjen testien ylläpidettävyyden ja modulaarisuuden ansiosta. Robot Framework ei kuitenkaan yksin riitä testien kirjoittamiseen, vaan se pitää yhdistää johonkin toiseen työkaluun, jonka avulla Robot voi hallita testattavaa ohjelmistoa.

Robot Framework yhdistettiin Spix-nimiseen avoimen lähdekoodin työkaluun. Työ tehtiin Piceasoft Oy:n toimeksiantona heidän mobiililaitteiden huollossa käytettävään QML-ohjelmistoonsa. Työssä käytiin läpi jonkin verran testiautomaation teoriaa, esiteltiin testattavan ohjelmiston toiminta, ja sitten testityökalut Robot Framework ja Spix. Lopuksi käsiteltiin Spixin yhdistämisprosessia Robot Frameworkiin, sekä pohdittiin testiautomaation toiminnallisuutta ja luodun testiautomaation tulevaisuuden näkymiä.

Työn tuloksena saatiin lupaava alku testiautomaatiolle. Spix saatiin yhdistettyä melko vaivattomasti Robotiin. Robot Frameworkiä voidaan tämän työn perusteella käyttää Spixin kanssa QML-ohjelmistossa. Testiautomaation toiminnallisuus kaipaa kuitenkin vielä laajentamista.

Avainsanat: Robot Framework, Qt, QML, Spix, käyttöliittymätestaus, testiautomaatio, GUI-testaus, GUI-testiautomaatio, käyttöliittymätestiautomaatio

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

C++, JavaScript: Ohjelmointikieliä
GUI: Graafinen käyttöliittymä, eng. Graphical User Interface
UI: Käyttöliittymä, engl. User Interface.
HTML: HyperText Markup Language, käyttöliittymämerkintäkieli
Robot: Käytetään lyhenteenä Robot Frameworkistä
XML: Käyttöliittymämerkintäkieli
WYSIWYG: engl. What You See Is What You Get, suom. *mitä näet, sitä saat*, koodieditori, jossa luotu sisältö näyttää lähes samalta kuin lopputulos

1. JOHDANTO

Huonosti toimiva ohjelmisto altistaa sen käyttäjän parhaassa tapauksessa turhautumiselle ja pahimmassa tapauksessa jopa onnettomuuksille tai tietomurroille. Oikeanlaisesta toiminnasta voidaan varmistua testauksen avulla, jota tehdään ohjelman jokaiselle osalle taustalogiikasta käyttöliittymään.

Graafista käyttöliittymää (engl. lyhenne GUI) on testattu historiallisesti lähinnä käsin. Testaaja antaa syötteitä ohjelmistolle esimerkiksi painelemalla nappeja ja kirjoittamalla tekstiä, ja pyrkii varmistumaan ohjelmiston oikeanlaisesta toiminnasta. Tällaiset nappien painallukset ja muut käyttöliittymälle tehtävät toimenpiteet kootaan testitapauksiksi, joissa toimenpiteet listataan oikeassa järjestyksessä dokumenttiin. Täten sama testi voidaan tehdä myös myöhemmin useaan otteeseen. Testitapauksia taas kootaan yhteen suuremmaksi dokumentiksi, jota sanotaan testisarjaksi (engl. test suite).

Suuri osa näistä testitapauksista ja -sarjoista on käytännössä täysin mahdollista automatisoida ohjelmalla, joka hoitaa automaattisesti syötteiden antamisen testattavalle ohjelmistolle. Tässä työssä tutkitaan UI-testiautomaatiotyökalua nimeltä Robot Framework, ja tarkemmin sitä, miten se voidaan yhdistää osaksi QML-ohjelmistoa. Työ tehdään Piceasoft Oy:n tilauksesta heidän mobiililaitteiden huollossa käytettävään ohjelmistoonsa. Työssä Robot Framework yhdistetään Spix-nimiseen avoimen lähdekoodin työkaluun. Spixin ja Robotin yhdistämisprosessia dokumentoidaan, minkä jälkeen pohditaan testiautomaatiota ja sen mahdollisia jatkokehityskohteita.

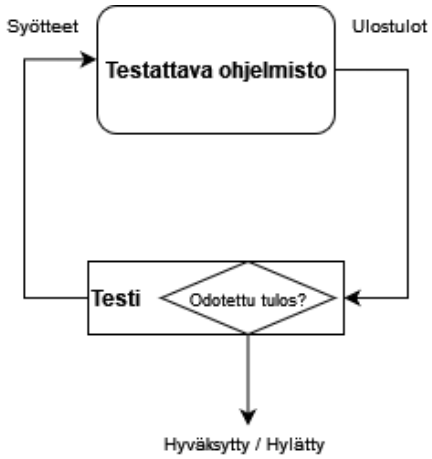
Automatisoitujen UI-testien kirjoittamisesta Qt-ohjelmistoihin on olemassa melko vähän materiaalia, ja nekin harvat julkaisut ja sanomalehtiartikkelit, joissa kirjoitetaan aiheesta, keskittyvät Qt:n Squish-testikehykseen. Robot Frameworkin käyttämisestä Qt-ohjelmistoissa ei ole käytännössä yhtään kirjallista materiaalia. Työn aihe on siis ohjelmistojen testauksen alalla jokseenkin vähän käsitelty aihe, ja voi olla hyödyksi kenelle tahansa, joka pyrkii käyttämään Robot Frameworkiä Qt-ohjelmistossa.

Vaikka Qt:n ja Robotin yhdistämisestä on vähän materiaalia, Robot Frameworkistä sitä on kuitenkin runsaasti. Tutkielmia aiheesta ovat tehneet mm. Anu Malm [1], Sami Mäkinen [2].

2. TESTIAUTOMAATIO

2.1 Testiautomaatio yleisesti

Ohjelmisto voidaan hyvin yksinkertaistetusti määritellä systeeminä, joka ottaa sisään syötteitä, prosessoi niitä ja tuottaa ulostulon. Automatisoitu testi (kuva 1) on ohjelma tai skripti, joka antaa syötteitä testauksen alla olevalle ohjelmistolle, vertaa ulostuloa tai sen osaa johonkin ennalta määriteltyyn tulokseen, ja antaa omana ulostulonaan vertailun tuloksen. [3, kpl. 6] Tämän jälkeen testin tulokset säilötään jollain tavalla.

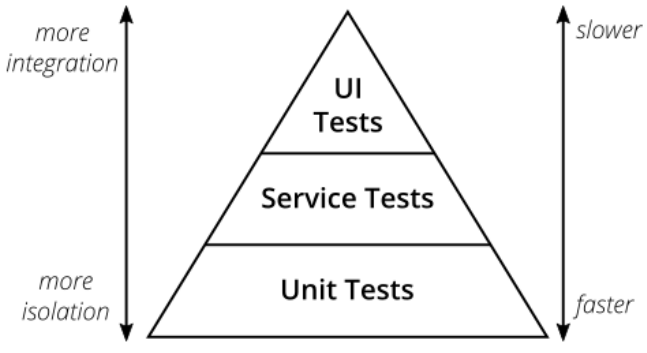


Kuva 1: Automatisoitu testi [3]

Testiautomaatiossa luodaan ja ylläpidetään tällaisia automatisoituja testejä.

2.2 Testiautomaation suunnittelu

Tyypillinen tapa mallintaa ohjelmistojen testauksen osia on niin sanottu testipyramidi (kuva 2), joka on tullut tunnetuksi Mike Cohnin kirjasta *Succeeding with agile* [4].



Kuva 2: Testipyramidi (Cohn, 2009)

Pyramidin on tarkoitus ilmaista ohjelmiston testaus kolmessa tasossa: UI-testit, palvelutestit ja yksikkötestit, sekä kuinka paljon testausta jokaiselle tasolle kuuluu. UI-testien olisi Cohnin mukaan tarkoitus olla verrattain pieni osa testiautomaatiosta, johtuen niiden suurista integraatio- ja aikakustannuksista. Kuvassa 2 *more integration* viittaa näihin kustannuksiin. *Slower* (hitaampi) ja *faster* (nopeampi) viittaavat mm. testien ajoon ja luomiseen kuluvaan aikaan. Käytännössä kuitenkin jokaisella ohjelmistolla on eri tarpeet testauksessa. Joillekin pyramidi pätee, kun taas joissain ohjelmistoissa, kuten tämän työn tapauksessa, UI-testit ovat itseasiassa olennaisempia kuin yksikkötestit ohjelmiston laadun valvonnassa. Tämä johtuu ohjelmiston alkuperäisestä suunnittelusta, jossa ei ole yksinkertaisesti otettu huomioon yksikkötestausta, mikä tekee sen toteuttamisesta vaikeaa ohjelmiston ollessa jo melko suuri. UI-testaus ottaa suuremman osan testitaakasta kuin yksikkötestaus tämän työn tilanteessa.

Axelrod puhuu kirjassaan testiautomaation suunnittelutavasta, [3, kpl. 3] jossa testiautomaatio koostuu uudelleen käytettävistä modulaarisista rakennuspalikoista. Nämä rakennuspalikat ja testien infrastruktuuri ovat kokeneempien ohjelmoijien vastuulla. Testien kirjoittamisesta vastaavat tässä lähestymistavassa testaajat tai vähemmän kokeneet ohjelmoijat. Lähestymistavan modulaarisuus vähentää työmäärää testien luomisessa ja ylläpitämisessä tehden koodista luettavampaa. Työmäärä jakautuu järkevästi testaajien ja ohjelmoijien välillä. Toinen yleinen tapa on palkata tehtävään erillinen testiautomaatioinsinööri, jonka vastuulla on pitää yllä testiautomaatiota.

2.3 Ylläpidettävyys

Tärkeimpiä asioita testiautomaation suunnittelussa on testien ylläpidettävyys. Testit, joita tulee jatkuvasti korjailla ja muuttaa, saavat aikaan suuria kuluja. Nämä kulut voivat pahimmassa tapauksessa olla niin suuria, että itse testiautomaatiosta on enemmän haittaa kuin hyötyä ohjelmistotiimille. Ohjelmistokehittäjät, testaajat tai testiautomaatioinsinöörit joutuvat käyttämään arvokasta aikaa pelkästään testien ylläpitämiseen, vaikka automatisoitujen testien on nimenomaan tarkoitus vähentää vaivaa testauksessa. [5]

2.4 Lokaattorit

Testejä kirjoittaessa ylläpidettävyyden kannalta yksi keskeisempiä asioita on, miten käyttöliittymäkomponenttien paikantaminen tehdään. Komponenttien paikantamiseen voidaan käyttää mm. koordinaatteja, komponentille annettua erillistä nimeä tai komponentin muita ominaisuuksia. Paikantamiseen käytettävää tietoa sanotaan *lokaattoriksi*.

Koordinaatteja pidetään yleisesti ottaen huonona lokaattorina, koska komponentin koordinaatit voivat vaihdella. Tämä voi johtua mm. muutoksista käyttöliittymäkoodissa tai siitä, minkä kokoisella näytöllä testejä ajetaan. [5]

Yleisesti ottaen parhaana lokaattorina voidaan pitää yksilöivää tunnistetta (engl. id), joka määritellään jo käyttöliittymää ohjelmoitaessa. Tunnisteen pystyy myös antamaan dynaamisesti käyttöliittymää ladattaessa, mutta tämä ei ole suositeltavaa. [5]

Lokaattorit usein säilötään jonkinlaiseen tiedostoon, jossa niille annetaan erilliset yksilöivät nimet, jotka yhdistetään lokaattoriin. Nimiä voidaan sitten käyttää testikoodissa monessa paikkaa, ja jos lokaattori muuttuu, sitä tarvitsee muuttaa vain kyseisessä lokaattoritiedostossa. [5] Yksi komponentti voi olla osana kymmeniä testitapauksia, ja lokaattorin muuttaminen jokaisesta on turha vaiva.

3. TESTATTAVAN OHJELMISTON TOIMINTA

Työssä testattava ohjelmisto on Piceasoft Oy:n mobiililaitteiden huollossa käytettävä Picea® Services, jota voidaan käyttää muun muassa tietojen siirtämiseen kännykästä toiseen tai laitteen resetointiin. Ohjelmisto kokonaisuudessaan on luotu Qt-ohjelmointiympäristössä ja käyttöliittymä QML-kielellä.

3.1 Qt

QtCreator on suomalaisyritys Qt Groupin valmistama ohjelmointiympäristö, jolla voidaan luoda C++-, JavaScript- ja QML-ohjelmia. Qt sisältää koodieditorin lisäksi mm. visuaalisen debuggerin, WYSIWYG-tyylisen integroidun GUI-suunnittelutyökalun ja tuen monelle eri C++-kääntäjälle. Koodieditori tarjoaa aiemmin mainituille ohjelmointikielille koodin värityksen ja automaattisen tekstintäytön. [6]

3.2 Qt-tapahtumat

Kuten monissa muissakin ohjelmointiympäristöissä, Qt-ohjelmissa tapahtumat (engl. event) tarkoittavat asioita, jotka tapahtuvat ohjelman sisällä tai jonkin ulkoisen toiminnan johdosta, joista ohjelman tulisi tietää. *Ulkoisia* tapahtumia voivat olla mm. hiiren tai näppäimistön painallus tai ikkunan koon muutos. *Sisäisiä* tapahtumia ovat esim. lasten lisäys komponentin alle tai ohjelman tilan muutos. Suurin osa Qt:n tapahtumista ovat kuitenkin ulkoisia tapahtumia, joka voidaan nähdä `QEvent`-luokan dokumentaatiosta [8, *QEvent Class*]. [6, *The Event System*]

Tapahtumat on toteutettu Qt:ssa luokkina, jotka periytyvät abstraktista `QEvent`-luokasta. Esimerkiksi `QMouseEvent`-aliluokalla voidaan käsitellä kaikenlaisia hiiren tapahtumia, kuten hiiren paikan muutoksia tai hiiren painalluksia. `QEvent`-luokan lapsilla on yleensä tapahtumaan ominaisia funktioita ja jäsenmuuttujia, esimerkiksi `QResizeEvent` säilöö näytön aiemman ja uuden koon, jotta käyttöliittymäkomponentit osaavat sopeutua niihin. Tapahtumiin on oltava käsittelijä, joka toteutetaan Qt:ssa yleensä virtuaalifunktiona. [6]

3.3 QML

Qt Modeling Language eli QML on deklaratiivinen ohjelmointikieli graafisten käyttöliittymien luomiseen Qt-applikaatiohin. Graafinen komponentti ilmaistaan sen *visuaalisella*

tyypillä, jonka jälkeen kaarisulkujen sisään määritellään komponentin ominaisuudet ja komponenttiin mahdollisesti sisältyvät alikomponentit.

```
Item {
    id: rectContainer
    width: 320
    height: 480

    Rectangle {
        id: rectangle
        color: "#272822"
        width: 320
        height: 480
    }
}
```

Koodi 1: Esimerkki QML-komponenteista [6, *Use Case – Visual Elements in QML*, muokattu]

Yllä koodissa 1 `Rectangle` (suorakulmio) on komponentin visuaalinen tyyppi, ja `id`, `color`, `width` ja `height` ovat komponentin ominaisuuksia (attribuutteja), jotka määrittevät sen ulkonäköä ja toiminnallisuutta visuaalisen tyypin lisäksi [6, *The QML type system*]. Olennaisimmat visuaaliset tyypit ja tietotyypit löytyvät Qt:n omasta Qt Quick-kirjastosta, minkä lisäksi käyttäjä pystyy lisäämään ulkoisia tai itse luomiaan kirjastoja QML-applikaatioonsa. Komponentit kootaan *qml*-päätteellisiin tiedostoihin.

Tämän työn kannalta kahta QML-ominaisuutta voidaan pitää kaikista olennaisimpina: `id`, ja `objectName`. Näitä molempia voidaan käyttää *lokaattoreina*, eli niitä voidaan käyttää komponenttien paikantamiseen näytöltä. Lokaattoreista muodostuu *komponenttipolkuja*, joiden avulla testaustyökalu pystyy helposti paikantamaan eri osassa ohjelmiston rakennetta olevia komponentteja. Komponenttipolku koostuu komponenttien id- tai objectName- arvoista, joita yhdistetään vinoviivoilla ” / ” poluksi, joka johtaa komponenttiin. Esimerkiksi koodissa 1 merkkijono

`”rectContainer/rectangle”`

johtaisi `rectContainer` - nimisen komponentin alla olevaan suorakulmioon `rectangle`.

QML-käyttöliittymä vastaa tapahtumiin *signaali-käsittelijä* -mekanismilla. QtQuick-kirjaston QML-komponenteille on määritelty joitain signaaleja valmiiksi, kuten `Button`-tyypin `clicked`-signaali, jota kutsutaan, kun nappia painetaan. [6]

```
Button {
    anchors.bottom: parent.bottom
    anchors.horizontalCenter: parent.horizontalCenter
    text: "Change color!"
    onClicked: {
        rect.color = Qt.rgba(0.3, 0.2, 0.7, 1);
    }
}
```

Esimerkki QML-signaalin käsitelystä

Yllä olevasta koodista voidaan nähdä, että signaalin nimi kirjoitetaan komponentin sisään `on<Signaali>:`, jossa signaalin nimen ensimmäinen kirjain kirjoitetaan isolla. Signaalin käsittelijäksi annetaan kaksoispisteen jälkeen JavaScript-funktio. [6]

3.4 Huomioitavaa

Työssä testattava ohjelmisto toimii tiiviisti yhdessä erilaisten mobiililaitteiden kanssa. Testattava ohjelmisto yhdistyy suoraan mobiililaitteisiin, lukee niistä tietoja ja muokkaa laitteiden tilaa. Tynkien käyttäminen on varsinkin tässä työssä tärkeää, koska testien kannalta oikeiden laitteiden kytkeminen koneeseen on hyvä pitää melko vähissä. Tyngällä (engl. stub) tarkoitetaan tässä tapauksessa fyysisen mobiililaitteen korvaamista valmiilla pohjalla, jotta testattava ohjelmisto käyttäytyy niin kuin siihen olisi kytketty oikea laite, vaikka laitetta ei ole oikeasti kytketty. Kaikki mobiililaitteeseen liitettävät tiedot, esimerkiksi valmistaja ja malli, voidaan lukea tiedostosta, johon tiedot on kirjoitettu valmiiksi.

Testattavan ohjelmiston käyttöliittymä reagoi dynaamisesti erilaisiin tiloihin, joissa mobiililaitteet voivat olla, ja tynkien käyttäminen mahdollistaa luotettavien testien ajamisen. Kyseessä ei ole itse laitteiden testaaminen, tai miten ne integroituvat ohjelmistoon, vaan tarkoituksena olisi käyttää simuloituja laitteita ja testata pelkästään ohjelman käyttöliittymää.

Testejä toki ajetaan oikeilla laitteilla varsinkin testauksen alkuvaiheessa, mutta mobiililaitetyngät voidaan nähdä tärkeänä jatkokehityksen kannalta. Automaattitestejä ajaessa testiohjelma ei välttämättä käyttöliittymän perusteella tiedä, onko ongelma testattavassa ohjelmistossa vai laitteessa. Mobiililaitteiden tila, kuten esimerkiksi muistissa olevat tiedot, voivat myös muuttua testien ajon aikana.

4. SPIX

4.1 Yleistä

Spix on avoimen lähdekoodin UI-testityökalu Qt/QML-applikaatioihin. QML-komponenttien paikantaminen tapahtuu kappaleessa 3.3 esitellyn komponenttipolun avulla. Esimerkiksi seuraava koodirivi: `mouseClick("mainWindow/ok_button");` painaa hiirellä QML-komponenttia, jonka id tai objectName on `ok_button`. Komponenttipolku sopeutuu käyttöliittymän suunnittelumuutoksiin, kuten paikan tai muiden attribuuttien muutoksiin. [7] Spixin ominaisuuksiin kuuluvat:

- hiiren painallukset, liikuttaminen, raahaaminen ja pudottaminen,
- tekstin kirjoittaminen,
- komponenttien näkyvyyden tarkastelu,
- QML-ominaisuuksien tarkastelu,
- näyttökuvien ottaminen ja tallentaminen,
- testattavan ohjelman sulkeminen ja
- komponenttien paikantaminen näytöltä. [7]

Spixin asentamiseen löytyy ohjeet projektin Githubista. Huomioitavaa on, että Spixissä ei ole muuta tapaa paikantaa käyttöliittymäkomponentteja kuin id tai objectName-kenttä. [7]

Testien luomiseen Spixissä on kaksi tapaa. Ensimmäinen on luoda Qt-tapahtumia sisäisesti, mikä voidaan tehdä C++-yksikkötestinä tai RPC-yhteyden yli. Sisäisten tapahtumien luominen johtaa esimerkiksi siihen, että hiiri ei liiku näytöllä, koska tapahtumat menevät suoraan testattavaan ohjelmistoon. [7]

Toinen tapa, joka täydentää ensimmäisen toiminnallisuutta, on käyttää Spixiä toisen GUI-automaatiokirjaston, kuten `pyautogui`:n kanssa, joka hoitaa hiiren painallukset, liikuttamisen ja muut syötteiden annot Spixin paikantamisfunktioiden avulla [7]. Kun nämä tekniikat yhdistetään, Spixillä voidaan päästä käsiksi suurin piirtein kaikkiin osiin testattavaa ohjelmistoa.

4.2 Spixin RPC-yhteys

Spix kommunikoi testattavan ohjelmiston kanssa ns. etäproseduurikutsujen avulla. Etäproseduurikutsu (engl. Remote Procedure Call, RPC) on tapa ajaa jokin proseduuri (aliohjelma, funktio) eri osoiteavaruudessa kuin missä proseduurin toiminnallisuus muuten olisi. Osoiteavaruus sijaitsee tyypillisesti toisessa tietokoneessa tai on toisen prosessin käytössä samassa tietokoneessa, mikä mahdollistaa datan hajauttamisen monen ohjelman käyttöön. RPC-yhteyttä käytetäänkin hajautettujen järjestelmien kommunikaatiotekniikkana. Proseduuri välittyy asiakas-palvelin -tyyppisesti *asiakkaalta*, esimerkiksi Spixiltä, *palvelimelle*, joka voi olla esimerkiksi Spixillä testattava ohjelmisto. Asiakas lähettää pyynnön ja saa siihen palvelimelta vastauksen. [8]

Etäproseduurikutsu voidaan luokitella myös IPC-kutsuksi (engl. Inter-Process Communication), koska sitä voidaan käyttää kahden prosessin väliseen tiedon jakamiseen [8]. Spix ajaa käskyjä testattavan ohjelmiston sisällä AnyRPC-nimisen etäproseduurihallintasysteemin avulla [7].

5. ROBOT FRAMEWORK

Robot Framework on Pythonilla tehty käyttöliittymäautomaatio-ohjelmisto. Robot Frameworkin on kehittänyt Pekka Klärck osana diplomityötään Nokia Networksille vuonna 2005. Haluttiin kehittää testiautomaatiokehys, joka varmistaisi helposti ylläpidettävät testitapaukset ja -sarjat, joiden ylläpidosta ja kirjoittamisesta voisivat vastata jopa ohjelmointia taitamattomat. [9] Vuonna 2008 Robot Frameworkista tehtiin avoimen lähdekoodin ohjelmisto, ja se onkin sen jälkeen noussut varsinkin Suomessa yhdeksi suosituimmista GUI-automaatiotyökaluista [10].

5.1 Testidata

UI-testit kirjoitetaan Robotissa selkokielisten *avainsanojen* avulla. Avainsanoja on kahdenlaisia: alimman tason avainsanat vaikuttavat suoraan testattavaan ohjelmistoon, ja ylemmän tason avainsanat kokoavat alemman tason avainsanoja yhteen. Avainsanat kootaan taulukkomaiseen muotoon testitapauksiksi, kuten alla olevassa koodinpätkässä. Robot-tiedostojen syntaksissa on jonkin verran valinnan varaa, mutta tyypillisin, ja myös tässä työssä käytettävä tyyli on kirjoittaa avainsanat taulukkoon, taulukon rivit välilyönneillä ja sarkaimilla eroteltuina. [11, *Test data syntax*]

```
***Settings***
Library     SpixLibrary.py
Variables   componentPaths.py

***Test Cases***
Example case
    Click    ${HOME_MENU_HISTORY}
    Scroll   down   ${HISTORY_LIST_VIEW}  10
    Move mouse  ${HISTORY_SEARCH_FIELD}
    Input text  Apple     ${HISTORY_SEARCH_FIELD}
```

Esimerkki yksinkertaisesta testitapauksesta

Yllä olevassa koodissa dollari-merkin ja kaarisulkeiden sisään kirjoitetut *muuttujat* määritellään erillisessä Variables-tiedostossa, joka sisällytetään Variables-otsikon avulla Settings-kohdassa testitiedostoon.

Testitiedostot koostuvat *testidatasta*, joka koostuu otsikoilla eritellyistä osioista, joita kutsutaan taulukoiksi. Otsikot sisällytetään kahden kolmen tähden (*) sarjan sisään. Ensimmäinen otsikko testidatassa on `***Settings***`, jossa määritellään asetuksia testejä varten, kuten testeissä käytettävät avainsanakirjastot Library-määreellä. [11]

Varsinaiset testitapaukset kirjoitetaan otsikon `***Test cases***` alle. Tähän kirjoitetaan testitapauksen nimi ja avainsanat taulukkona. Korkeamman tason avainsanoja voidaan määritellä `***Keywords***`-otsikon alle, mutta tätä varten on kuitenkin suotavaa luoda oma tiedosto. [11]

Robot Frameworkia voidaan käyttää myös ohjelmistorobotiikkaan. Tällöin ei kirjoiteta testitapauksia vaan tehtäviä (engl. tasks). Tehtävät kirjoitetaan `***Tasks***`-otsikon alle. Samaan tiedostoon ei voi kirjoittaa testitapauksia ja tehtäviä, sillä tämä johtaa virhetilanteeseen. Käyttäjä voi lisätä kommentteja `***Comments***`-otsikon alle. [11]

5.2 Testitiedostot

Testiautomaation tehokkaaseen ja ylläpidettävään toteutukseen tarvitaan monenlaisia tiedostoja. Näihin kuuluvat Robot Frameworkissa testitapaustiedostojen lisäksi ainakin ympäristötiedostot, muuttujatiedostot ja korkeamman tason avainsanat.

Testidatatiedostot ovat ensimmäinen tiedostotyyppi, jonka kanssa käyttäjä on todennäköisesti tekemisissä. Testidatatiedostot sisältävät varsinaiset testitapaukset. Yhteen tiedostoon voidaan kirjoittaa useita testitapauksia, joista Robot muodostaa automaattisesti testisarjan. Yksi testidatatiedosto on siis testisarja, ja näitä tiedostoja voidaan taas koota kansioon suuremmaksi testisarjaksi. Testidatan syntaksia käsitellään kappaleessa 5.1. [11]

Tilanteessa, jossa halutaan luoda korkeamman tason testisarja, on mahdollista, että samalla tavalla kuin testidatatiedostoissa, halutaan määritellä asetuksia ja muuttujia korkeamman tason testisarjaa varten. Kansio ei voi suoraan sisältää tällaista alustuskoodia, vaan sitä varten kansioon tulee lisätä *alustustiedosto*. Alustustiedosto on syntaksiltaan täysin samanlainen kuin testidatatiedosto, mutta siihen ei voida kirjoittaa testitapauksia. [11, *2.4.2*]

Resurssitiedostot (resource file) ovat syntaksiltaan ja tiedostopäätteiltään normaaleja Robot-tiedostoja, mutta niitä käytetään mm. korkeamman tason avainsanojen määrittämiseen.

```
*** Keywords ***
Open Login Page
    Open Browser    http://host/login.html
    Title Should Be    Login Page
```
Koodi: Esimerkki korkeamman tason avainsanojen määrittelystä [11]

Avainsanat määritellään yllä olevan koodipätkän mukaisesti. Korkeamman tason avainsanoja voidaan määritellä testidata-, resurssi- tai alustustiedostoissa. Testauksessa käytettäviä muuttujia voidaan määritellä *muuttujatiedostossa* (variable file), joka voidaan sisällyttää `Variables`-avainsanan avulla. [11]

5.3 Kirjastot

Robot Framework ei itsessään riitä testien luomiseen, koska se ei pääse käsiksi testattavaan ohjelmistoon ilman toisen testaustyökalun apua. Robotin käyttämät alimman tason avainsanat määritellään avainsanakirjastojen avulla. Joitain valmiita kirjastoja tarjotaan Robotin puolesta, kuten verkkosivuilla käytettävään UI-testityökaluun Selenium yhdistettävä SeleniumLibrary tai Squishin kanssa yhdistettävä SquishLibrary. [11]

Kirjastoja voi määritellä myös itse Robotin library-rajapinnan avulla, ja niitä voi kirjoittaa mm. Javalla ja Pythonilla. Pythonilla avainsanat määritellään funktioina joko luokassa tai moduulissa. [11]

5.4 Robotin asennus

Yksinkertaisin tapa asentaa Robot on käyttämällä pip-paketinhallintatyökalua.

`pip install robot`

Jos pip ei ole asennettuna, Python sisältää valmiin paketin `ensurepip` pip:in asentamiseen.

`py -m ensurepip –upgrade` (Windows)

`python -m ensurepip –upgrade` (Linux, Mac)

Asennus onnistuu myös Robot Frameworkin Githubista, tai asennuksen voi tehdä manuaalisesti. [11] Mainittakoon, että tämän työn tapauksessa vakioasennus on riittävä.

5.5 Testien tulokset ja virhetilanteet

Robot luo automaattisesti testisarjasta lokin ja kaksi tulostiedostoa, joista toinen on XML- ja toinen HTML-muodossa. Tulostiedostossa luetellaan testitapaukset ja niiden status (hyväksytty tai hylätty). Tulokset voi myös lukea ajon jälkeen komentoriviltä. Robot kirjaa testitapauksen epäonnistuneeksi, jos sen ajon aikana jokin avainsana epäonnistuu.

Tämä voi tapahtua, jos avainsanaa käytetään väärin, tai jos ajon aikana koodissa nostetaan virhe. [11]

6. TYÖKALUJEN YHDISTÄMINEN

Tässä osiossa kuvataan Spixin yhdistämisprosessia Robot Frameworkiin. Tämä koostuu Spixin asennuksesta, Spixin toiminnallisuuden laajentamisesta omalla Python-moduulilla ja Robot-avainsanakirjaston luomisesta.

6.1 Spixin asennus

Spixin riippuvuudet Visual Studio ja Cmake tulee hankkia ja asentaa ensimmäisenä. Kun ne ovat asennettuina, AnyRPC, GoogleTest ja Spix voidaan kloonata GitHubista, ajaa CMakella, ja sitten kääntää Visual Studiolla. Jotta CMake-komento toimii, pitää komennon yhteydessä määritellä DCMAKE_PREFIX_PATH-muuttuja.

Spixin asennusprosessi oli melko vaikeaa varsinkin ensimmäisellä kerralla, koska jokaiseen eri riippuvuuteen on omat asennusohjeensa, ja CMaken ympäristömuuttujien konfigurointiin ei oikeastaan ollut ohjeita, ne piti päätellä itse. Asennusprosessin yhteydessä kirjoitettiin kuitenkin ylös asennusohjeet, joista voi tulevaisuudessa tehdä automaattisen asennusskriptin, joka helpottaa asennusta huomattavasti.

Spix liitetään osaksi testattavaa ohjelmistoa lisäämällä alla oleva koodinpätkä ohjelmiston ylimmälle tasolle.

```
spix::AnyRpcServer server;
auto bot = new spix::QtQmlBot();
bot->runTestServer(server);
```

Koodinpätkä mahdollistaa etäproseduuriyhteyden luomisen Spixin ja testattavan ohjelmiston välille. Kun nämä toimenpiteet ovat tehty, Spix-testejä voidaan alkaa kirjoittaa Pythonilla.

6.2 Oma Spix-moduuli ja avainsanakirjasto Spixillä

Työssä tehtiin oma Python-moduuli, joka hyödyntää Spixiä. Moduulia käytetään Robot-avainsanakirjaston luomisessa. Spixiä voitaisiin periaatteessa käyttää suoraan Robot-kirjastossa, mutta sen valmis toiminnallisuus ei työn kirjoitushetkellä riitä kaikkiin tarpeellisiin toimintoihin. Spixin kanssa käytettiin pyautogui-kirjastoa. Python-tiedostoon `utils.py` koottiin funktioita, jotka toteuttavat Spixin perustoimintojen lisäksi seuraavat toiminnot:

- hiiren rullaaminen,
- pikanäppäinten painaminen,
- yksittäisen napin painallus,
- hiiren liikuttaminen,
- dropdown-menun painaminen ja
- komponentin odottaminen.

Seuraavaksi luotiin Robot-avainsanakirjasto alimman tason avainsanojen toteuttamiseen Spixillä. Avainsanakirjasto luotiin Robot-luvussa esitellyllä tavalla, jossa Python-funktioita merkitään luokan metodeina.

```
@library
class SpixLibrary:

    @keyword
    def Click(self, obj):
        u.mouseClick(cp.MAIN+obj)

    @keyword("Wait For")
    def waitFor(self, obj):
        u.waitFor(cp.MAIN+obj)

    @keyword
    def Scroll(self, dirString, obj, scrolls):
        # dir => "up" or "down"
        dirInt = 0
        upper = dirString.upper()
        if upper == "UP":
            dirInt = 1
        elif upper == "DOWN":
            dirInt = -1
        u.scrollFor(cp.MAIN+obj, dirInt, int(scrolls))
```

Koodi: Ote kolmesta ensimmäisestä avainsanasta Spix-avainsanakirjastossa

Yllä olevan otteen lisäksi kirjasto toteuttaa muutamia muita avainsanoja, ja avainsanoja voidaan lisätä tulevaisuudessa tarpeen mukaan.

7. TULOS

Spix pystyy paikantamaan melkein kaikki käyttöliittymäkomponentit, mutta huomattavaa on, että jos komponentille ei ole määritelty `objectName`- tai `id`-kenttää, ei Spix sitä pysty paikantamaan. Kaikille komponenteille testattavassa ohjelmassa pitäisi siis olla määriteltynä joko `id` tai `objectName`. Jos niitä ei ole määriteltynä, komponenttien paikantamiseen voi käyttää myös koordinaatteja, mutta tämä ei yleisesti ottaen ole hyvää suunnittelua. Tätä työtä varten kehitetyssä `utils`-kirjastossa on `moveAndClickOffset`-funktio, jolla voi painaa jonkin komponentin viereen komponentin mittojen mukaan kertoimella, jota voi käyttää esim. listakomponenttien painamiseen, jos listakomponenteilla ei erikseen ole määriteltynä lokaattoria.

Spix sisältää erittäin hyödyllisen funktion `getStringProperty`, jolla voidaan hakea mikä tahansa *merkkijono*-ominaisuus QML-komponentista. Ongelmana kuitenkin on se, että muutkin ominaisuudet kuin ne, jotka sattuvat olemaan merkkijonoja, olisi hyvä tietää testejä kirjoittaessa. Esimerkiksi komponentin alikomponentteihin (lapsiin) olisi hyvä päästä käsiksi.

Eräs ominaisuus, josta voisi tulevaisuudessa olla hyötyä, olisi testien nauhoittaminen. Tällaista ominaisuutta ei Spixistä löydy. Nauhoittamalla pystyisi luomaan testejä nopeasti ilman minkäänlaista ohjelmointia.

Viimeisenä mainittakoon, että Spixiä pystyy laajentamaan itse. Monet tässä luvussa mainituista ominaisuuksista voi itse kehittää, mutta niiden luominen, ylläpitäminen ja debuggaaminen veisi paljon aikaa. On myös mahdollista, että Spixin kehittäjät lisäävät ominaisuuksia tulevaisuudessa.

8. YHTEENVETO

Spix-Robot -yhdistelmä riittää todennäköisesti testien tekemiseen aluksi, mutta pidemmällä aikavälillä tulee todennäköisesti ongelmia eteen, kun jotkin tärkeät ominaisuudet puuttuvat. Spixin voi sanoa olevan täysin toimiva työkalu toistaiseksi. Jos tulevaisuudessa todetaan, että halutaan käyttää jotain muuta työkalua testeihin, ja jos uusi työkalu yhdistyy myös Robot Frameworkiin, ei Spix-kirjastoa käyttävät aiemmin tehdyt testitapaukset ole missään nimessä turhia. Aiemmat testitapaukset voidaan ajaa Robotilla yhdessä uusien kanssa.

Tuloksena saatiin lupaava alku testiautomaatiolle, jota kuitenkin tulee laajentaa. Spixin ja pyautogui-kirjaston avulla pystytään tekemään käytännössä kaikki syötteiden annot testattavalle ohjelmalle, kuten hiiren painallukset ja tekstin kirjoittaminen. Robotin yhdistäminen näiden päälle tekee suurtenkin testikokonaisuuksien hallitsemisesta ja ylläpitämisestä huomattavasti helpompaa.

LÄHTEET

[1] Malm, A. *UI-testiautomaation aloitus Robot Frameworkia hyväksi käyttäen*, 2020

[2] Mäkinen, S. *Käyttöliittymän testaus Robot Frameworkilla*, 2018

[3] Axelrod, A. *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*, Apress, 2018

[4] Vocke, H. *The practical test pyramid*, URL: https://martinfowler.com/articles/practical-test-pyramid.html, viitattu 27.10.2021

[5] King, J., *Test Automation Best Practice #4: Use Reliable Locators*, Ranorex, 11.5.2018

[6] *Qt Documentation*, Qt 5.15, The Qt Company Ltd, viitattu 1.11.2021

[7] Spix Githubissa, URL: https://github.com/faaxm/spix , viitattu 25.10.2021

[8] Nelson, B. *Remote Procedure Call*. Xerox Palo Alto Research Center. 5/1981

[9] Knowit AB, *Robot Framework Tutorial 1 – Johdanto*, URL: https://www.youtube.com/watch?v=H9YVlFKdOeM

[10] Robot Frameworkin verkkosivu, URL: https://robotframework.org , 10/2021

[11] *Robot Framework User Guide*, Robot Framework Foundation, viitattu versioon 4.1.2