TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatika instituut

IDK40LT
Gert Valdek 120947IAPB

AUTOMAATTESTIMISE PLATVORMI ARENDUS TAXIFY MOBIILIRAKENDUSELE

Bakalaureusetöö

Juhendaja: Jekaterina Tšukrejeva
Magistrikraad
Õppejõu assistent

Tallinn 2016

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Gert Valdek

23.05.2016

Annotatsioon

Käesoleva bakalaureusetöö eesmärk on välja töötada automaattestimise platvorm, millega saaks Taxify *Android* kliendirakenduse funktsionaalsuse toimimist automaatselt kontrollida. Töö käigus vaadeldakse erinevaid mobiilirakenduste testimise meetodeid ja vahendeid ja selle tulemusena arendatakse valmis automaattestimise platvorm. Lisaks kaetakse testidega ära kliendirakenduse sisselogimise funktsionaalsus ning analüüsitakse valminud platvormi puudusi ning võimalikke edasiarendusi.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 35 leheküljel, 5 peatükki, 17 joonist.

Abstract

Automated Testing Platform Development for Taxify Mobile Application

The purpose of this thesis is to develop an automated testing platform which can be used to automatically test the functionality of Taxify Android client application. This work analyzes different mobile application testing methods and tools and as a result of that automated testing platform is developed. Also login functionality of Taxify application will be covered with automated tests and platform's cons and possible future developments are being analyzed.

The thesis is in Estonian language and contains 35 pages of text, 5 chapters, 17 figures.

Lühendite ja mõistete sõnastik

CI	*Continuous integration*, tarkvaraarenduse automatiseerimise süsteem ja tava
API	*Application programm interface*, protokoll, rutiin või tööriist tarkvara arendamiseks
.APK	*Android application package*, Android mobiilirakenduse faili formaat
Android	*Android*, operatsioonisüsteem mobiilsetele seadmetele
iOS	*iOS*, operatsioonisüsteem mobiilsetele seadmetele,
UI	*User interface*, rakenduse visuaalne kujundus
Keystore	*Keystore*, mobiilirakenduse sertifikaat
HTML	*HypeText markup language*, veebilehtede märgendamise keel
GPS	*Global positioning system*, asukoha määramise süsteem
Unit test	*Unit test*, tarkvara testimise meetod, millega testitakse lähtekoodi osade toimimist
Skript	*Script*, programmikood, mille ülesandeks on automatiseerida mingit tegevust
Port	*Port*, tähistab andmesidet kasutavate andmesideühenduste lõpp-punkti
Plugin	*Plugin*, tarkvara komponent, mis lisab spetsiaalse uue funktsionaalsuse olemasolevale rakendusele
Linux	*Linux*, operatsioonisüsteem
OSX	*OSX*, operatsioonisüsteem
Spring Framework	*Spring Framework*, Java rakenduste arendamiseks kasutatav abistav raamistik
Google Maps	*Google Maps*, kaardirakendus

Sisukord

Jooniste loetelu

1 Sissejuhatus

Käesoleva bakalaureusetöö raames keskendutakse Eesti idufirma Taxify OÜ (edaspidi Taxify) Android kliendirakenduse jaoks automaattestimise platvormi loomisele.

1.1 Taust ja probleem

Taxifys on reeglina ühe arendustsükli pikkuseks viis tööpäeva ning peale seda toimub uue kliendirakenduse versiooni avalikustamine. Iga arendustsükli käigus on vaja manuaalselt kontrollida rakenduse funktsionaalsuse ja äriloogika toimimist mitmeid kordi. Selline tegevus on iga nädalaselt korduv ning seetõttu väga ebaefektiivne. Ressursse tuleks ära kasutada palju efektiivsemalt ning selle probleemi lahendamiseks tuleks manuaalne testimine asendada automaatse testimisega.

Äriloogika ja põhifunktsionaalsus ei ole ajas kiiresti muutuvad, mistõttu on nende toimimise automaatne kontrollimine mõistlik tegevus. Kui valmis arendada põhjalik platvorm ning katta automaattestidega põhiosa funktsionaalsusest, tagatakse parema kvaliteediga rakendus. Lisaks võimaldab pidev automaatne testimine vigade avastamise varajases arendusfaasis, mida on vähem kulukas parandada.

1.2 Ülesande püstitus

Bakalaureuse töö põhilised eesmärgid:

Eesmärk 1: Arendada välja platvorm ning eeldus Taxify Android kliendirakenduse automaattestimiseks.

Eesmärk 2: Arendada automaattestimise platvormile paar testijuhtumit, millega saaks automaatselt testida Taxify rakenduse mingit funktsionaalsuse osa.

1.3 Metoodika

- Analüüsitakse erinevaid testimise meetodeid ning leitakse Taxify vajadusi arvestades sobivaim meetod.
- Nõutele vastavate töövahendite valik ja põhjendus.
- Luuakse ning seadistatakse tööks automaattestimise platvorm eelnevalt valitud vahendeid kasutades.
- Arendatakse valminud automaattestimise platvormile paar testjuhtumit ning analüüsitakse nende tulemusi.

1.4 Ülevaade tööst

Bakalaureusetöö sisu on jaotatud 5-e peatükki vahel. Esimeses uuritakse erinevaid testimise meetodeid ja viise ning võrreldakse erinevaid virtuaalsete ja päris mobiilsete seadmete olemust.

Teises peatükis valitakse nõutele vastavad töövahendid, mida hakatakse platvormi loomise juures kasutama. Samuti paigaldatakse ning seadistatakse kõik valitud vahendid.

Kolmandas peatükis selgitatakse, kuidas tuleks testid arhitektuuriliselt üles ehitada, et neid oleks võimalikult lihtne hiljem täiustada või hooldada. Seejärel esitatakse testjuhtumitele kriteeriumid ning kaetakse sisselogimise funktsionaalsus automaattestidega.

Neljandas peatükis pannakse arendatud automaattestid tööle ning tutvustatakse samm haaval, kuidas see toimib. Lisaks analüüsitakse testide tulemusi.

Viiendas peatükis loetletakse välja arendatud platvormi suurimad puudused ja kitsaskohad ning analüüsitakse edasiarenduse suundi ning ideid.

1.5 Osaühing Taxify

Taxify OÜ on Eesti tarkvaraettevõte, mille põhiliseks tegevuseks on taksotellimise tarkvara loomine. Ettevõte asutati 7. veebruaril 2013.

Taxifys on kaks erineva funktsiooniga mobiilirakendust: kliendi- ja juhirakendus. Kliendirakenduse tööpõhimõtteks on automaatselt positsioneerida kliendi asukoht ning kuvada talle lähimad autod. Pärast tellimist näeb klient rakenduses reaalajas kaardi peal, kuidas juht tema juurde saabub. Sõidu lõpus on võimalik kliendil jätta juhile hinnang, mille alusel tagab Taxify teenuse hea kvaliteedi. Juhirakendust kasutavad juhid, kellele saadetakse kliendirakendusest tehtud tellimus. Juhile kuvatakse kliendi asukoht kaardil. Mõlemad rakendused on saadaval nii *Android* kui ka *iOS* operatsioonisüsteemidele. Antud lõputöö raames keskendutakse *Android* kliendirakenduse jaoks automaattestimise platvormi arendamisele.

Ettevõttes kestab reeglina üks arendustsükkel viis päeva ning toote arendus jaguneb järgmiselt:

1. Planeerimine ja analüüs – koostatakse plaan äripoolega ning analüüsitakse selle mõju.

2. Nõuded ja disain – pannakse paika reeglid ja nõuded, millele arendus vastama peab. Tehakse kujundus.

3. Arendus – Toimub arendus vastavalt eelnevates punktides kokku pandud plaanidele ja tingimustele.

4. Testimine – Kontrollitakse kas funktsionaalsus vastab seatud nõutele. Lisaks toimub tarkvaravigade otsimine. Vigade korral saadetakse rakendus tagasi arendusse ning hiljem uuesti testimisse.

5. Avalikustamine – Rakendus tehakse avalikult kättesaadavaks.

2 Mobiilirakenduste testimine

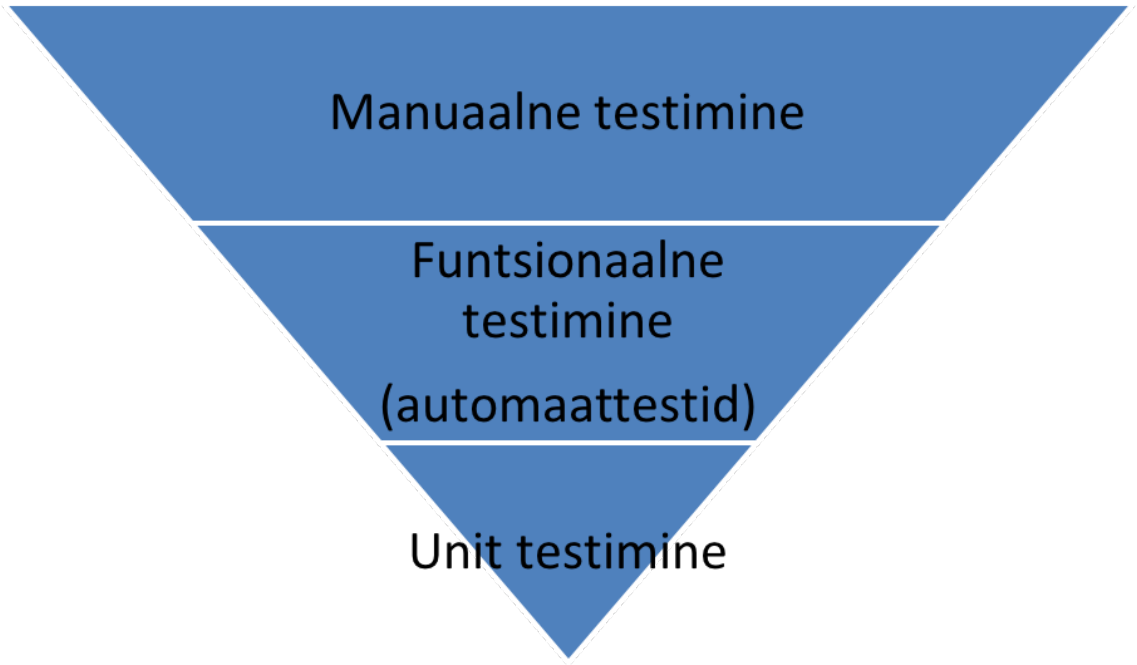
Mobiilirakenduste testimine võib erineda tavalise tarkvara testimisest oluliselt, juba selle poolest, et mobiilirakenduste suhtes on kasutajatel palju kõrgemad ootused, kuid tingimused rakenduse normaalseks toimimiseks võivad olla palju halvemad seadmete rohkuse ja erinevuste tõttu [1].

Üheks suureks katsumuseks on see, et kuna mobiilirakenduste kasutajad on enamasti pidevas liikumises ning rakenduse toimimine sõltub väga palju võrguühendusest, siis tuleb loodavat rakendust kasutada võimalikelt reaalsetes oludes. Testimisel tuleks väga palju rõhku pöörata just erinevate sensorite manipuleerimisele (näiteks GPS side) ning samuti valmistab katsumust seadmete puhul nende mobiilne võrguühendus, mille stabiilsus ning kiirus sõltub väga palju ümbritsevast keskkonnast.

Taxify kliendirakenduse puhul on suureks katsumuseks see, kuidas positsioneerida klient õigesse kohta, kui rakendus avatakse. Sellest sõltub ka edasine kasutamise kogemus, kas suudetakse tuvastada kliendi täpne asukoht. Samuti sõltub rakendus suuresti võrguühendusest ning selle ebastabiilsuse korral võib tekkida mitmeid probleeme.

Mobiilirakenduste testimise meetodite jaotus on väga palju erinev tavalise tarkvara testimisest. Juba selle tõttu, et mobiiliseadmete puhul on oluline mängida erinevate sensoritega, imiteerida ekraanil liigutusi, muuta orientatsiooni ning muid taolisi tegevusi. Praeguses olukorras ei ole veel mobiilirakenduste automatiseerimise vahendid arenenud piisavalt kaugele, et saaks (või oleks mõistlik) igat võimalikku olukorda automatiseerida. Joonisel 1 on välja toodud testimise meetodite jagunemise püramiid ning sellest kõige väiksema osa moodustab *Unit* testimine. Tihti võib *Unit* testimine osutuda võimatuks või ebaotstarbekaks (majanduslikel kui ka ajalises mõttes), sest mobiilirakendused tihti vajavad mingeid sisendeid, mida kasutajad peavad andma ning nende arendamine on keerukas. Keskmise osakaaluga on funktsionaalne (automaatne) testimine – sellega saab kontrollida kas kõik vajalik loogika toimib. Kõige suurema osakaaluga on manuaalne testimine, sest see tagab kõige parema tulemuse. Küll aga on

manuaalne testimine korduv ning ajakulukas tegevus, mis tõttu tuleks võimalikult palju tegevusest automatiseerida [1].



Joonis 1. Mobiilirakenduste testimise püramiid [1].

2.1 Testimise meetodid

Mobiilirakenduste puhul on võimalik rakendada nende spetsiifika ja olemuse tõttu väga mitmeid erinevaid testimise meetodeid: mobiili spetsiifiline testimine, funktsionaalne testimine, kasutatavuse testimine, juurdepääsetavuse testimine (puudustega inimestele), aku kasutamise testimine, jõudluse testimine, turvalisuse testimine, stabiilsuse testimine, uuendamise testimine, andmebaasi testimine, kohaliku andmebaasi testimine, vastavuse testimine [1].

Rakenduse funktsionaalne testimise meetod on iga tarkvara projekti juures kõige olulisem tegevus. Taxify kliendirakenduse puhul kasutatakse seda sama meetodit, sest ärilisest poolest on see kõige prioriteetsem ning kriitilisem. Funktsionaalne testimine tähendab seda, et veendutakse kas rakenduses tehtud funktsionaalsus toimib kõik vastavalt nõutele [1]. Funktsionaalsust saab testida erinevates tingimustes ning keskkondades. Oluline oleks testida ka kasutajate võimalikke teguviise. Näiteks kui

panna kasutajaks registreerimise protsessi ajal mobiilirakendus taustale ning uuesti avada – oodatud tulemusena peaks rakendus jätkama sealt, kus ta pooleli jäeti.

2.2 Manuaalne vs automaatne testimine

Manuaalne testimine on väga ressursimahukas töö ning pidevalt korduvad tegevused on mõistlik automatiseerida. Ainult automaattestimine ei ole lahenduseks – näiteks *GPS* andmete ning erinevate sensorite näitudega mängimine võib ebamõistlikult keeruliseks või võimatuks osutuda, kui on piiratud ressursid – seetõttu tuleks kaaluda kas on ka mõistlik seda teha. Samuti kui rakendus on olemuselt väga lihtne, väga piiratud funktsionaalsusega ning saadaval avalikult ainult piiratud aja, siis ei ole mõistlik sellele automaatteste arendada [1]. Taxify kliendirakenduse puhul on tegemist keeruka rakendusega, sest sisaldab väga palju erinevat funktsionaalsust ning automatiseerimine on mõistlik ja tasuv. Tasuvus väljendub selles, et vigu on võimalik avastada juba varajases arendusfaasis. Lisaks tasuvusele on selle tõttu võimalik tagada ka parem rakenduse kvaliteet. Ühtlasi aitab automaatne testimine vähendada ettevõtte ärilisi riske seoses tarkvara arenduse vigadega.

Enne automatiseerima asumist tuleks erinevad testjuhtumid väga detailselt kirja panna ning ka mitmeid kordi manuaalselt läbi testida erinevate seadmete peal, seejärel saab selgeks, millised osad oleks mõistlik automatiseerida [1]. Taxify kliendirakenduse puhul rakendatakse funktsionaalse testimise meetodit ning testjuhtumite loomise kriteeriumeid ja prioriseerimist käsitletakse hiljem punktis 4.2, lk 24.

2.3 Emulaator vs simulaator vs päris seadmed

Automaatteste on võimalik jooksutada kolmel erinevat seadme tüübil: emulaator, simulaator ning päris seadmed. Oluline oleks aru saada nende seadmete töö põhimõtetest, mis on testimise juures vajalik.

Mobiilseadme emulaator on rakendus, mis tõlgendab kompileeritud mobiilirakenduse lähtekoodi ülesanded nii, et selle mobiilirakenduse saab käivitada arvutis. Emulaator käitub täpselt nagu päris mobiili seadme riistvara ja operatsioonisüsteem. Kuna emulaator jookseb arvutil, siis ei ole võimalik kasutada ega rakendada kõiki päris

mobiilidele omaseid sensoreid. Näiteks *Android* rakendusi arendatakse emulaatorites [1].

Mobiilseadme simulaator on rakendus, mis on vähem keerukas kui emulaator ning simuleerib väikest osa mobiili käitumisest ning riistvarast. Simulaatoritel ei ole võimalik katsetada mobiili seadmetele omaseid riistvaralisi omadusi või sensoreid, kuid mobiili operatsioonisüsteem on reaalsusele palju lähedasem, lisaks on need palju kiiremad kui emulaatorid. Näiteks *iOS* rakendusi arendatakse simulaatorites [1].

Suurim erinevus simulaatori ja emulaatori vahel on see, et simulaator jäljendab täpselt päris mobiili seadet, samas kui emulaator proovib jäljendada mobiili seadme kogu sisemist arhitektuuri [1].

Päris seadmed on testimiseks kõige paremad, kuid võivad olla ettevõtte jaoks ka kõige kulukam lahendus. Kulukus väljendub selles, et mobiili operatsioonisüsteeme on väga palju erinevaid, millel tuleb igat rakendust testida ning lisaks mängivad tihti rolli ka seadmete füüsilised omadused (näiteks ekraani suurus). Esialgseks alternatiiviks on mõistlik kasutada *Android* kliendirakenduse puhul emulaatorit ning tulevikus osta füüsilised seadmed, kui platvorm on täielikult välja arendatud. Emulaatoreid saab luua lõpmatult ning tasuta, erinevate näitajatega. Samuti on üheks lisavõimaluseks kasutada pilvepõhiseid seadmete laboreid, kus teenuspakkuja päris seadmetel on võimalik teha automaatteste.

3 Töövahendite valimine, paigaldamine ja seadistamine

Automaattestide platvormi arendamiseks on vaja välja valida kolm vahendit. Antud töös ei käsitleta ega võrrelda sama liiki tööriistade puudusi või eeldusi, vaid esitatakse põhinõuded kolmele vahendile, millele tuginedes valik tehakse:

1. **Automatiseerimise tööriist**, mille ülesandeks on hakata teste läbi viima imiteerides liigutusi ning vajutusi kasutatavas seadmes. Nõuded:
    - *Android* ja *iOS* operatsioonisüsteemide tugi.
    - Testide programmeerimine keeles *Java*.
    - Android ja iOS automaattestide arendus samas programmeerimise keeles, mis võimaldab kahe rakenduse sarnasuse korral kiiresti duplitseerida teste ka teisele platvormile.
    - Pilvepõhine mobiilirakenduste testimise tugi *Amazonis (AWS Device Farm)*
    - *CI* tugi
2. **Testimise raamistik**, mis hakkab teste jooksutama. Nõuded:
    - Pilve põhine mobiilirakenduste testimise tugi *Amazonis (AWS Device Farm)*
    - *CI* tugi
3. **Emulaator** ehk virtuaalne seade, kus teste jooksutatakse. Nõuded:
    - *Google Play* teenuste tugi, *sh Google Maps*
    - Erinevate seadmete sensorite programmaatiline sättimine, sh *GPS* asukoha seadistamine
    - *CI* tugi

3.1 Appium

Automatiseerimise tööriist

Appium on avatud lähtekoodiga tööriist, millega saab automatiseerida mobiilirakenduse funktsionaalset testimist. Võimalik on imiteerida vajutusi, puudutusi või ekraanil lohistamist rakenduses sees. Lisaks sisestada tekste, sulgeda rakendust taustale, lukustada ekraani ning palju muid erinevaid tegevusi veel, mida üldse on võimalik tavakasutajal mobiiltelefoni kasutades teha.

Appiumi eelised [2]:

- Testimiseks saab kasutada rakenduse lõppversiooni, mis tähendab, et Appiumi raamistik ei vaja selleks eraldi testitavasse rakenduse integreerimist. Samuti ei pea lähekoodi kompileerima iga kord, kui teste teha, vaid saab kasutada otse *Android* rakenduse *.apk* faili.

- Testide programmeerimiseks on toetatud mitmed erinevad keeled: *Java, Objective-C, JavaScript, PHP, Python, Ruby, C#, Clojure* ja *Perl*.

- *Android* ja *iOS* operatsioonisüsteemide tugi.

- Tasuta ja avatud lähtekoodiga.

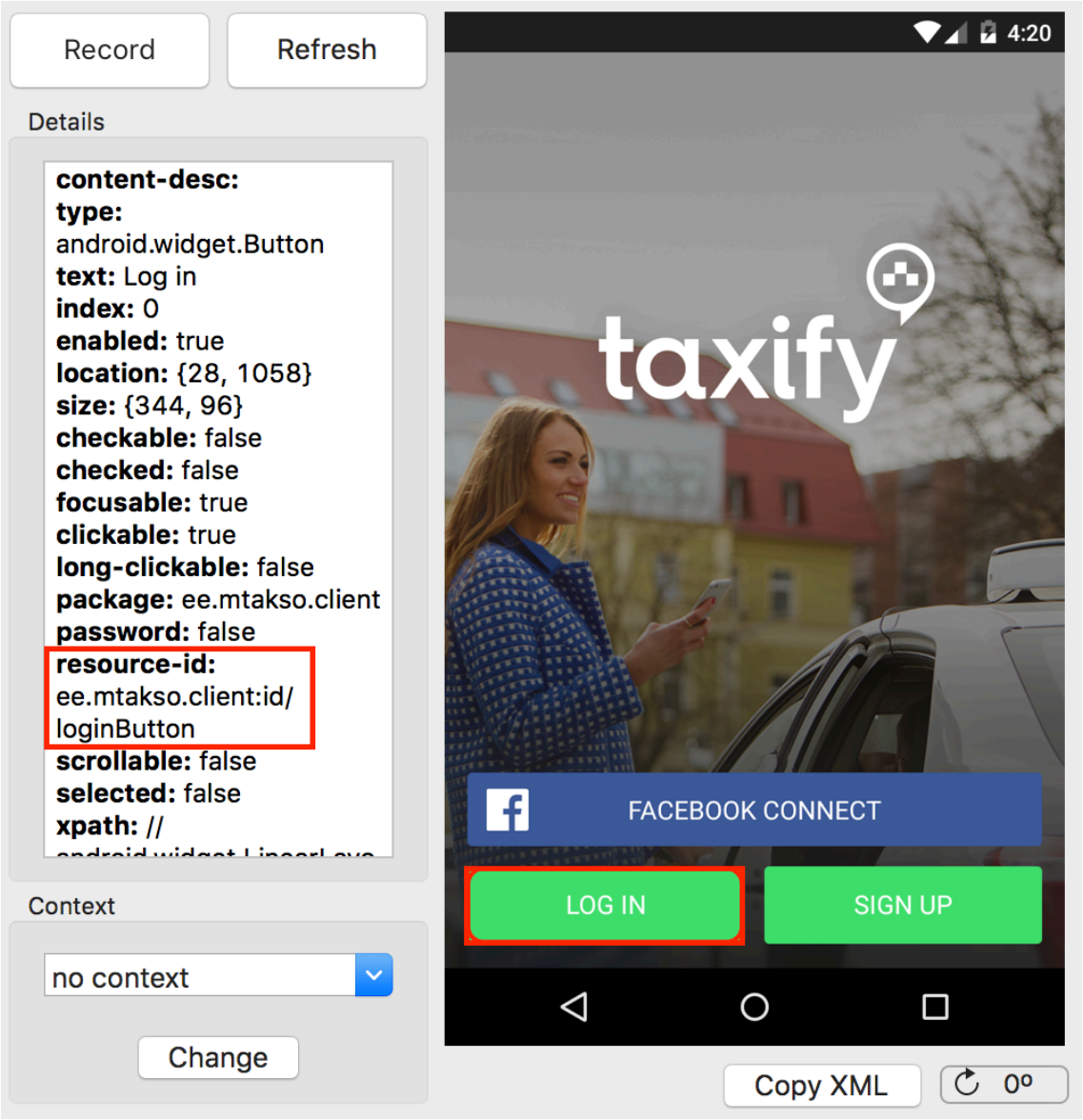
- Mitmed suuremad *CI* teenust pakkuvad ettevõtted toetavad seda.

- Võimalik käivitada ja juhtida käsurealt

Taxify puhul osutus antud tööriist valituks just selle tõttu, et toetatud on nii *Android* kui ka *iOS* operatsioonisüsteemiga seadmed. Taxify *iOS* ja *Android* kliendirakendused on funktsionaalsuselt samad ning selle tõttu ka testjuhtumid on sarnased. See võimaldab ühele operatsioonisüsteemile arendatud automaatteste taaskasutada teise süsteemiga, sest teste on võimalik arendada samas programmeerimise keeles. Lisaks on *Appium* puhul toetatud ka *Amazon Web Sevices Device Farm*, mis võimaldab tulevikus hakata kasutada pilvepõhist automaattestimist erinevatel päris seadmetel.

Rakenduse elemente (nupud, tekstiväljad) saab identifitseerida mitme meetodiga. Kõige mõistlikum oleks seda teha kasutades *Appium inspectorit*. Tegemist on tööriistaga, mis installeerib seadmele peale testitava mobiilirakenduse ning käivitab selle. Seejärel kuvatakse arvutiekraanile rakenduses kuvatav pilt ja kursoriga on võimalik valida vastav element ning vaadata selle detaile.

Taxify rakenduses on iga element identifitseeritav *resource-id* järgi. *Resource-id* on konkreetse vaate elemendi unikaalne nimi, millega saab selle tuvastada ning hiljem testides viidata selle nime kaudu õigele elemendile. Selle identifikaatori kasutamise eeliseks on see, et kui kujunduse paigutust muudetakse arenduse käigus, ei ole vajalik automaatteste uuendada. Lisaks töötab see ka iga ekraani suuruse ning orientatsiooni puhul. Joonisel 2 on toodud näide, kus punasega on märgitud *Login* nupp ja selle *resource-id*.



Joonis 2. Ekraanitõmmis. *Appium Inspector* tööriista abil elemendi identifitseerimine.

Appiumi serveri paigaldamine

Appium on saadaval eraldi rakendusena arendaja ametlikul leheküljel. Rakenduse programmaatiliseks käivitamiseks oleks mõistlik kirjutada skript, mille saab testimise projektis hiljem välja kutsuda. Joonisel 3 välja toodud skript sisaldab asukoha viidet *Appium* rakendusele kohalikus arvutis. *–p 4474* tähistab porti, mis aadressil server käivitatakse. *--use-keystore* on oluline selleks, kliendirakenduse signeerimisel *Appiumi* poolt kasutatakse lisaks rakenduse *keystoret*. Sellega on vajalik signeerida rakendus, et *Google Maps* toimiks.

```
/Applications/Appium.app/Contents/Resources/node_modules/appium/bin/appium.js
-p 4474 --use-keystore
```

Joonis 3. *Appiumi* serveri käivitamise skript.

Pärast käivitamist hakkab *Appiumi* server tööle aadressil *http://0.0.0.0:4474/* .Server töötab taustaprotsessina.

***Desired capabilities* Appiumile**

Desired capabilites on valik parameetreid ja väärtusi, mis saadetakse *Appiumi* serverile, et öelda serverile, millist automatiseerimise sessiooni täpselt tahetakse [2]. Joonisel 4 on näha kood *setUpAppium* meetodis väärtustatud parameetritest, mis *Appiumi* serverile saadetakse. *platformName* väärtuseks on *Android* ning tähistab operatsioonisüsteemi, mida testimisel kasutatakse, *deviceName* tähistab seadme nime, mille põhjal *Appiumi* server selle tuvastab, *platformVersion* on Androidi operatsioonisüsteemi versioon ning *app* näitab kliendirakenduse *.apk* faili asukohta.

```
capabilities.setCapability("platformName", "Android");
capabilities.setCapability("deviceName", "AndroidTestDevice");
capabilities.setCapability("platformVersion", "5.0");
capabilities.setCapability("app","app/taxifyClient-mock_server-debug-
CA.2.68.apk" );
```

Joonis 4. Koodinäide *Appiumi* serveri parameetrite väärtustamisest.

3.2 TestNG

Testimise raamistik

TestNG on testimise raamistik, mis on välja arendatud *JUnit* põhjal, aga sisaldab palju parandusi ning lisafunktsioone, mis teevad selle kasutamise lihtsamaks [3]. See testimise raamistik hakkab juhtima ja jooksutama kõiki automaatteste, kuvab testide tulemusi jooksvalt ning vajadusel genereerib ka testraporti.

TestNG on valitud põhjusel, et seal on toetatud lisaks kaks vajalikku annotatsiooni, mis Taxify automaattestide projekti puhul on vajalikud. *@BeforeSuite* võimaldab käivitada *mocking* serveri enne, kui üldse testid käivitatakse ning *@AfterSuite* võimaldab peatada hiljem, kui testid on täielikult lõpule viidud. Lisaks on võimalik ühe testide klassis ära grupeerida kõik meetodid. See võib olla vajalik selleks, et lisada prioriteete testidele [3]. Näiteks *CI* korral saaks valida, millised testide grupid oleks vajalik jooksutada iga päevaselt ning milliseid harvem.

3.3 Genymotion

Emulaator

Genymotion on virtuaalne *Android* operatsioonisüsteemiga seade, mis emuleerib väga ligilähedaselt või isegi paremini päris seadet [4].

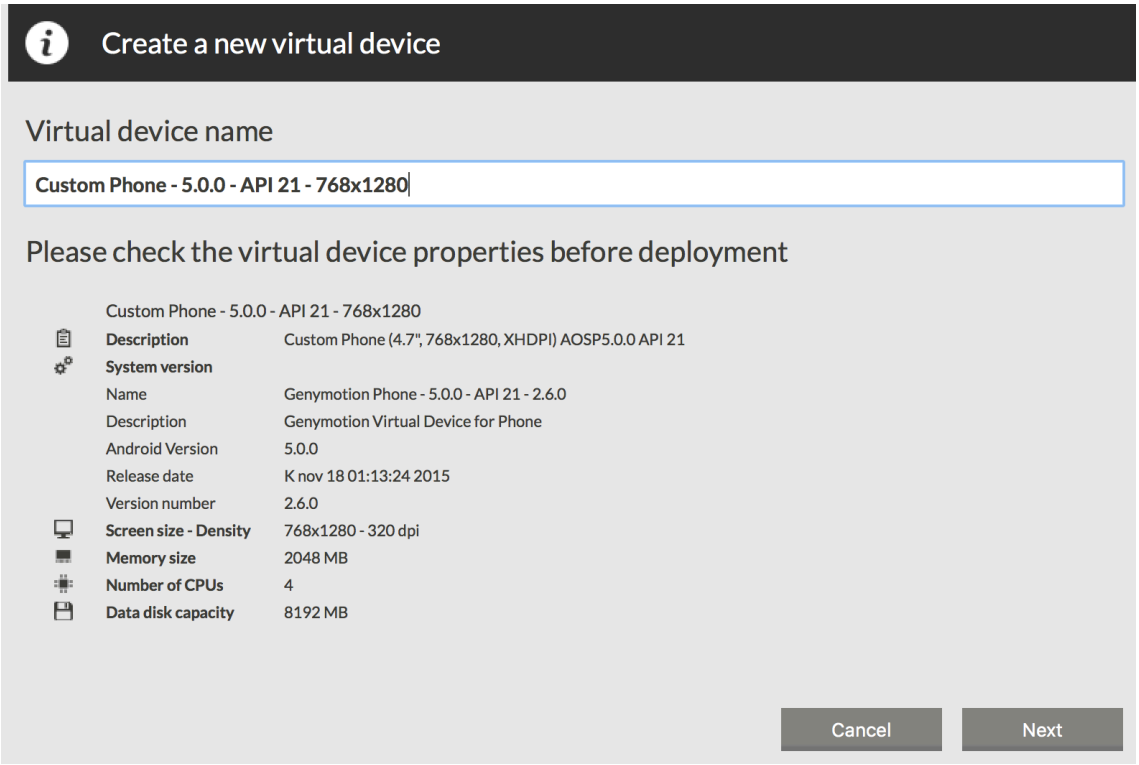
Eelised ning *Genymotioni* valimise põhjused:

- *Google Play* teenuste tugi - Taxify kliendirakendus vajab toimimiseks *Google Mapsi* tuge seadmel.
- *Genymotion Java API* - võimaldab muuta telefoni sensorite parameetreid programmaatiliselt. Hilisemas edasiarenduses on võimalik hakata sensorite andmeid (näiteks *GPS*) muutma projektis sees, mis tähendab, et testjuhtumitele on võimalik juurde lisada asukoha muutustega seotud probleemid.
- *Genymotion Gralde Plugin* – Võimaldab *Gradle* skripti kaudu kontrollida ning luua programmaatiliselt uusi virtuaalseid seadmeid.

Genymotioni kasutamine ärilisel eesmärgil on tasuline. Lõputöö raames kasutan tasuta versiooni, mis on uurimuslikul eesmärgil lubatud.

Paigaldamine

Genymotion on saadaval rakendusena tootja ametlikul leheküljel. Pärast paigaldamist tuleb luua virtuaalsed seadmed manuaalselt. Antud lõputöö raames seadmete programmaatilist loomist ei teostata, kuid seda tuleks teha edasi arendamisel tulevikus. Luuakse seade joonisel 5 kuvatud parameetritega.



Joonis 5. Ekraanitõmmis. *Genymotion* rakenduses uue virtuaalseadme loomine.

Loodud Genymotioni virtuaalseadme automaatseks käivitamiseks tuleks luua skript. Joonisel 6 on välja toodud koodirida, millega saab käivitada eelmises punktis loodud emulaatorit.

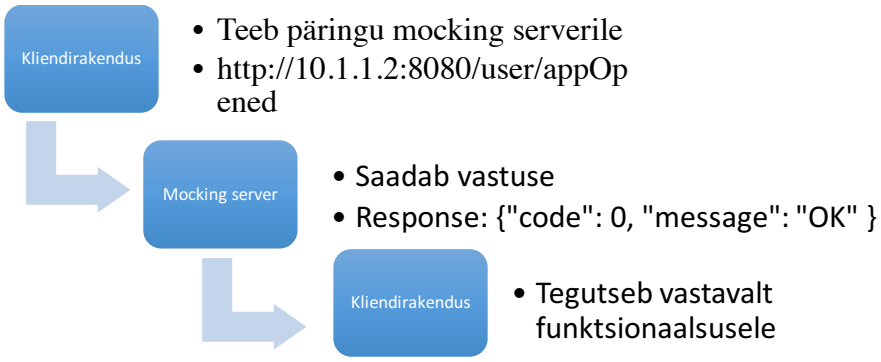
```
/Applications/Genymotion.app/Contents/MacOS/player --vm-name "Custom Phone - 5.0.0 - API 21 - 768x1280"
```

Joonis 6. *Genymotion* virtuaalseadme käivitamise skript.

3.4 *Data mocking* Taxify rakenduses

Taxify kliendirakenduse toimimine sõltub serverist, mis saadab kliendirakenduse päringu peale vajaliku info. Keeruliseks teeb automaattestimise puhul see, et iga testjuhtumi puhul on vaja simuleerida täpselt samu andmeid ning olukordi rakenduses. Selleks, et luua iga testjuhtumi puhul täpselt identsed olukorrad, on loodud *mocking*

server, mis jookseb kohalikus masinas *Spring Framework* peal ning kus on defineeritud iga testjuhtumi jaoks vajalikud andmed. Kliendirakendus käivitub ning teeb päringud *mocking* serverile. Enne igat testimist tuleb eelnevalt peale laadida konkreetne testjuhtum, mis *mocking* serveris jookseb ning saadab vastavate andmetega vastuse. Joonisel 7 on toodud näide *appOpened* päringu ja vastuse kohta, kus tehakse *appOpened* päring. Kliendirakenduse puhul tehakse see päring iga kord, kui rakendus avatakse. Server saadab vastuseks *"OK"* ning rakendus alustab tööd. Sama moodi toimub protsess ka teiste päringute puhul.



Joonis 7. Kliendirakenduse ja *mocking* serveri andmete vahetamise näide.

Mocking server on juba eelnevalt valmis loodud ning päringud ja vastused defineeritud. Selle tööle panemiseks tuleb projekt käivitada ning kasutada *REST* rakendust, millega saab testjuhtumi peale laadida. Antud lõputöö raames kasutatakse *Google Chrome* veebibrauserile tasuta saada olevat laiendust *Advanced REST Client*.

3.5 *Gradle* projekt

Loodav raamistik arendatakse välja kasutades *Gradle't*. Tegemist on avatud lähtekoodiga kompileerimise raamistikuga, mis automatiseerib tarkvara ehitamist, testimist ning avaldamist [5].

Gradle eeliseks konkurentide ees on see, et sellele platvormile arendatakse väga palju *lisapluginaid*. Kuna hilisemaks edasiarenduse eesmärgiks on täisautomatiseerimine, siis peatükis 3, lk 15 seati üheks kriteeriumiks, et kasutatavad vahendid oleksid CI toega. *Gradle* jaoks on loodud spetsiaalselt kaks *pluginat*, mida hetkel lõputöö raames ei kasutata, kuid on oluliselt põhjuseks, miks see platvorm valiti:

- *AWS Device Farm Gradle Plugin* – võimaldab *Amazon Device Farm* pilveteenust kasutades kontrollida programmaatiliselt seadmeid ning teste nende peal läbi viia [6].

- *Genymotion Gradle Plugin* – Võimaldab kontrollida loodud virtuaalseid seadmeid programaatiliselt. Lisaks saab valida, millise seadme käivitada, milliste parameetritega (näiteks sensorite parameetrid) [7].

4 Automaattestid

4.1 Arhitektuur

Automaatteste tuleb uuendada ja täiustada iga arendustsükli ajal, sest testid on sõltuvad *UI'st*, mida väga tihti uuendatakse. Selleks tuleb testid programmeerida nii, et kui rakenduse kujunduses või funktsionaalsuses toimub muudatus, saaks ka teste võimalikult kiiresti uuendada või täiustada. Kogu arhitektuuri luues tuleb silmas pidada just paindlikust ning ajalist faktorit, mis kulub testide hilisemale hooldamisele ning edasi arendusele.

Projekti lähtekood on jaotatud kolme erinevasse kausta:

- ***Cases*** - See kaust sisaldab kõiki testjuhtumeid. Testjuhtumid luuakse vastavalt seatud kriteeriumitele ja vajadusele. Iga testgruppi jaoks on loodud eraldi klassifail. Üks klassifail võib sisaldada mitmeid teste (s.t testjuhtumeid), mis on omavahel kuidagi sarnased või seotud**.** Näiteks sisselogimisega seotud testjuhtumid kuuluvad samasse klassi.

- ***Dependency*** - Seadistuse failid, mis on vajalikud testkeskkonna toimimiseks.

- ***Views*** - Iga vaate kohta on loodud eraldi klassi fail. Testides tuleb kasutada *UI* elemente korduvalt erinevates testjuhtumites. Selleks tuleb luua projektis iga erineva vaate jaoks eraldi klassifail ning defineerida kõik elemendid, mis nimetatud vaates asuvad. Selle tulemusena, kui muutub või lisandub mõni element vaatesse, saab muuta konkreetse vaate klassis definitsiooni ning testid töötavad edasi.

Joonisel 8 on välja toodud näide, kuidas *LoginView* puhul on defineeritud kaks elementi. Esimene *loginButton* on nupp, mis teostab sisselogimise. Teine on tekstisisestus väli, kuhu tuleb kasutajal sisestab mobiilinumber sisselogimiseks.

```
@AndroidFindBy(id = "ee.mtakso.client:id/loginButton")
private MobileElement loginButton;

@AndroidFindBy(id = "ee.mtakso.client:id/phoneInput")
private MobileElement phoneNumberField;
```

Joonis 8. Elementide defineerimise koodinäide.

Testjuhtumites tuleb väga tihti kasutada korduvalt samu tegevusi või vajutusi, selleks on mõistlik luua eraldi meetodid, mida saab vajadusel hilisemalt kiiresti ka uuendada. Joonisel 9 on näitena on loodud eraldi meetod *LoginView* klassis, mida saab kõikides testjuhtudes edaspidi kasutada. Meetod teostab sisselogimise nupu vajutuse.

```
public void loginButtonClick() {
    loginButton.click();
}
```

Joonis 9. Sisselogimise meetodi koodinäide.

Testjuhtumite puhul on oluline oodatud tulemuse kontrollimine. Ühe testjuhtumi puhul võib toimuda mitmeid kontrolle, et mis vaates parasjagu viibitakse. Joonisel 10 on välja toodud näide, mis asub *LoginView* vaate klassis. Selleks on loodud eraldi meetod kontrollimaks, kas asutakse selles konkreetses vaates. Lahendusena vaadatakse kas just need kaks elementi, mis on *LoginView* vaate puhul unikaalsed, on hetkel nähtavad. Seda meetodit saab kasutada erinevate testjuhtumite puhul korduvalt ning kui peaks toimuma selles vaates muudatus, siis saab ka meetodi uuendada kiiresti nii, et kõik testid endiselt veel toimivad.

```
public boolean checkIfLoginViewOpen() {
    if (loginButton.isDisplayed() && phoneNumberField.isDisplayed()) {
        return true;
    }
    return false;
}
```

Joonis 10. Elemendi olemasolu kontrollimise meetodi näide.

4.2 Testjuhtude arendamine

Testjuhtude loomisel tuleks seada prioriteedid, millises järjekorras neid arendama hakata ning mis on kõige olulisem Taxify kliendirakenduse puhul. Testjuhtude automatiseerimise kriteeriumid olulisuse järjekorras:

1. Ärilisest poolest väga kriitiline põhifunktsionaalsus. Näiteks auto tellimine.

2. Muu funktsionaalsus, mille nõutele vastavust kontrollitakse rohkem kui üks kord, kuid ei avalda lühikese perioodi jooksul äriliselt väga suurt halba mõju probleemide korral. Näiteks sõidu lõpp-punkti sisestamine rakendusse.

3. Funktsionaalsuses harva esinevad või keeruliselt tekitatavad olukorrad. Näiteks teatud liiki veateated.

4. Arendustöö käigus tekkivad tarkvaravead, mille põhjuse leidmisel saab järeldada, et see viga võib uuesti korduda. Näiteks tõlgete sünkroniseerimisel või uuendamisel tekkiv viga.

Sisselogimise funktsionaalsus on kriitilise tähtsusega osa Taxify kliendirakenduse juures ning lisas 1 on toodud näide, kus on arendatud kõik testjuhtumid selle funktsionaalsuse juures. Oluline oleks ära märkida, et enne igat testjuhtumit taaskäivitatakse ning puhastatakse rakendus ära, et eelnevatest testidest ei jääks alles tekkinud infot.

loginSuccessfully testjuhtumi puhul on arendatud sisselogimise meetod, mis kutsutakse alguses välja. Sisestatakse olemas oleva kasutaja telefoni number ning vajutatakse sisselogimise nuppu. Seejärel kontrollitakse kasutades *Assert* meetodit kas SMS kinnituskoodi sisestamise vaade on nähtaval. Selle vaate olemasolu järgi saab teha otsuse, et sisselogimine õnnestus. Lisaks sisestatakse järgmise sammuna SMS kinnituskood.

phoneNumberInvalid testjuhtumi puhul sisestatakse rakendusse mitte standardile vastav telefoni number ning kontrollitakse kas kuvatakse õige kirjega veateade.

userNotFound testjuhtumi puhul sisestatakse rakendusse telefoni number, millega varasemalt pole kontot loodud ning kontrollitakse kas kuvatakse õige kirjega veateade.

userBlocked testjuhtumi puhul sisestatakse rakendusse blokeeritud kasutaja telefoni number ning kontrollitakse kas kuvatakse õige kirjega veateade.

loginViewToBackground testjuhtumise puhul avatakse sisselogimise vaade, seejärel suletakse rakendus taustale 8 sekundiks ning avatakse uuesti. Kontrollitavaks ning oodatud tulemuseks on see, et rakendus taastab sama vaate, kust see enne taustale suleti.

5 Testide jooksutamine

TestNG on automaattestide raamistik, mis kontrollib testide käivitamist ning lõpuni viimist. Raamistik võimaldab defineerida kõik testid, mida oleks vaja käivitada selleks eraldi failis *testng.xml*. Joonisel 11 on toodud näide, kuhu on lisatud *LoginTest* testjuhtumite klass. Lisaks on lisatud testidele ka kuulaja - *dependency.ScreenshotFactory*, mis vastutab ekraanitõmmiste tegemiste eest (vt punkt 5.1 lk 28)

```
<suite name="Android Automated tests">
    <listeners>
        <listener class-name="dependency.ScreenshotFactory" />
    </listeners>
    <test name="AcceptanceTests">
        <classes>
             <class name="cases.LoginTest"/>
         </classes>
    </test>
</suite>
```

Joonis 11. Koodinäide *testng.xml* failist.

Automaatteste peab saama jooksutada iga inimene, kes käivitab projekti. Eesmärgiks oleks see, kui arendaja tõmbab alla projekti ja paigaldab vajalikud lisaprogrammid endale arvutisse, peaks olema tal võimalik teste jooksutada võimalikult kiiresti. Lisaks on eesmärgiks, et tulevikus *CI* välja töötades ei peaks manuaalset tööd olema.

Testide käivitamiseks kirjutatakse käsureale *gradle clean test*. Selle järel hakkab *Gradle* jooksutama teste kasutades selleks *TestNG* raamistikku. Eelduseks on, et *mocking* server on juba käivitatud ning testjuhtum peale laetud kasutades *REST* rakendust *http://localhost:8080/loadTest?name=AcceptanceTest*. Testimise protsess toimub järgmiselt:

1. *@BeforeSuite* annotatsiooniga *setUpAppium* meetod kutsutakse välja.
    a. Joonisel 12 toodud koodireaga käivitatakse *Appiumi* server joonisel 3 kuvatud skriptiga. Server töötab taustal ning väljundit ei anna. Kui peaks tekkima tõrge, siis prinditakse veateade automaattestide projekti. Enamus juhtudel ei ole *Appium* serveri logide nägemine oluline, sellepärast ei kuvata neid ka reaalajas testide jooksutamise hetkel.

```
Process appium = Runtime.getRuntime().exec(APPIUM_PATH_GERT);
```

Joonis 12. *Appiumi* programmaatiline käivitamine skriptist.

b. Joonisel 13 toodud koodireaga käivitatakse *Genymotioni* emulaator. Selleks kasutatakse eelnevalt loodud joonisel 6 kuvatud skripti. Võimaliku edasiarendusena tuleks jätkata nii, et kasutada *Gradle Pluginit*, et luua virtuaalseid seadmeid programmaatiliselt. Praegune lahendus ei ole väga paindlik ning sobib ajutiseks kasutamiseks ainult, kui projekt jookseb arendaja kohalikus arvutis.

```
Process geny = Runtime.getRuntime().exec(GENYMOTION_PATH_GERT);
```

Joonis 13. *Genymotion* emulaatori käivitamine skriptist.

c. *Appium* serverile saadetakse *desired capabilites* ehk siis parameetrid, milliste seadistustega soovime serverit käivitada.

d. Toimub Taxify kliendirakenduse installatsioon emulaatorile. Kliendirakenduses on tehtud vastavad seadistused, mille tõttu ühendub see automaatselt *mocking* serveri külge ning hakkab päringuid sinna saatma.

2. *@AfterMethod* annotatsiooniga *closeApp* meetod kutsutakse välja iga testjuhtumi järel. Sellega kindlustatakse, et enne testjuhtumi käivitamist puhastatakse kliendirakenduse lokaalsed andmed ning tehakse täiesti uus sessioon.

3. Genereeritakse *HTML* vormingus tulemuste raport kohaliku arvuti kettale.

Joonisel 14 on kujutatud testide jooksutamise lõpptulemus, kus on näha, et viiest testjuhtumist neli õnnestusid ning üks ebaõnnestus.

```
Total time: 1 mins 44.672 secs
Gerts-MacBook-Pro:android Gert$ gradle clean test
:clean
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava
warning: [options] bootstrap class path not set in conjunction with -source 1.7
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 warning
:processTestResources UP-TO-DATE
:testClasses
:test

Android Automated tests > AcceptanceTests > cases.LoginTest.loginViewToBackground FAILED
    org.openqa.selenium.WebDriverException at LoginTest.java:85

5 tests completed, 1 failed
:test FAILED
```

Joonis 14. Ekraanitõmmis käsurealt. Testide käivitamine ja tulemused.

5.1 Tulemused ja analüüsimine

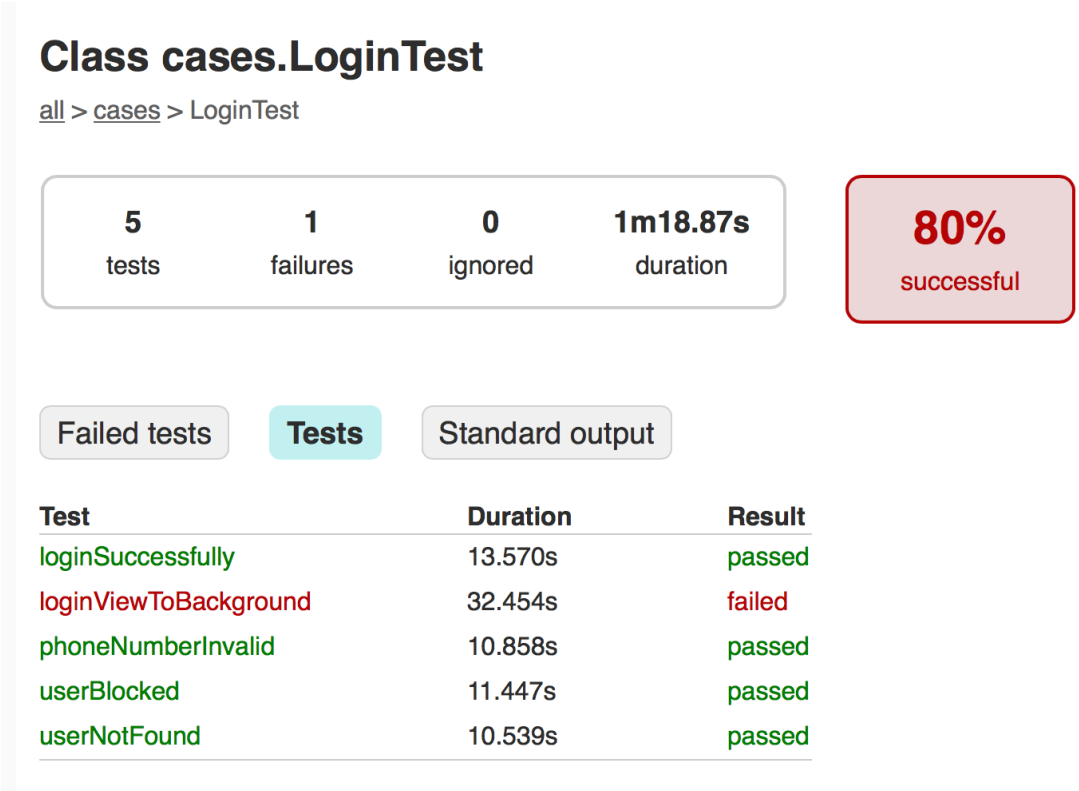
Ekraanitõmmiste tegemine

Kui testjuhtum peaks ebaõnnestuma on tagantjärgi keeruline kindlaks teha, mis põhjusel või kus kohas see täpselt põrus. Lahenduseks oleks ekraanitõmmiste tegemine testitavast kliendirakendusest juhul, kui test ebaõnnestub ning salvestada see kohaliku arvuti kettale.

Lisas 2 on toodud välja *ScreenshotFactory.java* faili lähtekood, mis teeb ekraanitõmmiseid ning kasutab testide jooksvaks jälgimiseks *TestNG* raamistikus defineeritud kuulajaid. *onTestFailure* vastutab selle eest, et kutsutakse välja õige meetod. Seejärel tehakse ekraanitõmmis rakendusest sees, salvestatakse see kohaliku arvuti kettale ning iga pildi nimeks pannakse testi klassi nimi, meetodi nimi ning kuupäev ja kellaaeg sekundi täpsusega.

HTML testraport

Pärast igat testtsükli lõppu genereerib *Gradle* automaatselt *HTML* vormingus testraporti, kus on testide tulemused näha. Joonisel 15 on näha ekraanitõmmis tulemustest.



Class cases.LoginTest

all > cases > LoginTest

5	1	0	1m18.87s
tests	failures	ignored	duration

80% successful

Failed tests | Tests | Standard output

Test	Duration	Result
loginSuccessfully	13.570s	passed
loginViewToBackground	32.454s	failed
phoneNumberInvalid	10.858s	passed
userBlocked	11.447s	passed
userNotFound	10.539s	passed

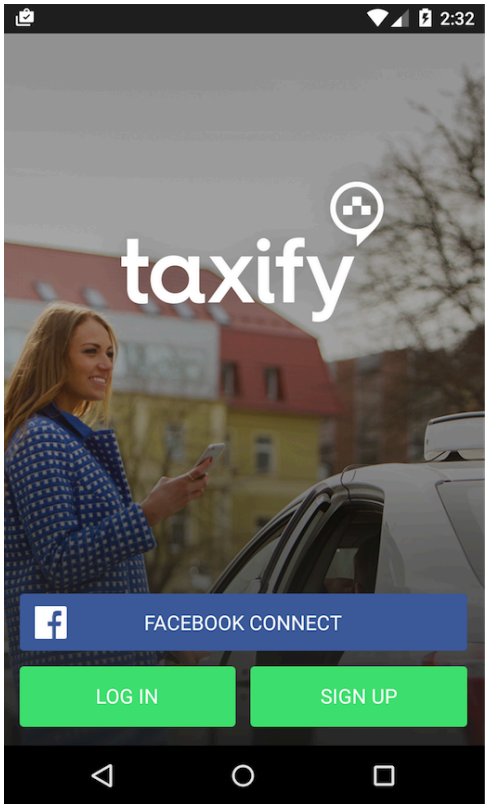
Joonis 15. Ekraanitõmmis. *Gradle* genereeritud *HTML* testide tulemuste raport.

Tulemustest on näha, et viiest testjuhtumist neli õnnestusid. Selleks, et selgeks saada, miks *loginViewToBackground* testjuhtum ebaõnnestus, tuleks vaadata kõige pealt konkreetse testjuhtumi ebaõnnestumise logisid, mis on näha testraportis. Joonisel 16 on välja toodud selle testjuhtumi ebaõnnestumise logid. Sealt saab järeldada, et mingil põhjusel rakendus ei jätkanud sisselogimise vaatest (testjuhtumite nõuded esitati punktis 4.2 lk 24).

```
Original error: ee.mtakso.client/.activity.LoginActivity never started.
Current: ee.mtakso.client/.activity.HomeActivity
```

Joonis 16. *loginViewToBackground* ebaõnnestumise logid.

Teise sammuna kontrollitakse üle ekraanitõmmis, mis tehti testi ebaõnnestumisel rakendusest seest. Joonisel 17 on näha, kuidas rakendus jätkas taustalt uuesti avanedes hoopis vales vaates, millest võime järeldada, et tegemist on rakenduses tarkvaralise veaga.



Joonis 17. Ekraanitõmmis rakendusest. *loginViewToBackground* testjuhtum.

6 Lõpptulemus ja edasiarendus

Lõputöö tulemusena välja arendatud automaattestimise süsteem on väga paindlik ning arendatud välja nii, et sellel oleks võimekus ka teisi Taxify rakendusi testida. Selleks oleks vaja muuta ainult *Appium* serverile saadetavaid parameetreid ning *iOS* rakenduse puhul lisaks ka simulaatorit. Testide arendus, meetod ning analüüsimine toimib täpselt samamoodi kõikide Taxify *Android* ja *iOS* rakenduste puhul.

Suurimad puudused:

- *Mocking* server testjuhtumite laadimine peaks toimuma automaatselt – üks testimise tsükkel koosneb väga paljudest testidest ning iga test vajab erinevaid seadistusi ning olukordi, mis luuakse kasutades *mocking* serverit. *Appiumi* projekti tuleks integreerida *mocking* serveri käivitamine, peatamine ning testjuhtumite automaatne peale laadimine.

- *Genymotion* emulaatoreid on võimalik luua programmaatiliselt kasutades *Genymotion Gradle pluginit*. Hetkel on need loodud kohalikus arvutis ning projekti jagades ei ole võimalik samu virtuaalseid seadmeid kaasa anda.

- Testide tulemuste jagamine peaks olema automaatne ning kõigile nähtav - selle probleemi saaks lahendada kasutades *CI* süsteemi. Lisaks ei panda kaasa testi ebaõnnestumisel rakendusest tehtud ekraanitõmmist, kuid see oleks mõistlik raportile külge panna.

Kõige suurema edasiarenduse lisaväärtuse annaks see, kui platvorm arendada täisautomaatseks. Selle eelduseks oleks kogu platvorm ühildada mõne *CI* süsteemiga. *CI* on tarkvaraarenduse automatiseerimise süsteem ja tava, mille puhul arendajad jagavad lähtekoodi mitu korda päevas ühises pilvepõhises hoidlas. Iga koodi jagamise järel paneb see süsteem lähtekoodi ehitama, samuti on võimalik jooksutada ka automaatteste iga korda, kui keegi jagab uut koodi [8]. Kokkuvõttes aitab see parandada tarkvara kvaliteeti, sest arenduse käigus tehtavad vead tulevad varakult välja. Lisaks puudub igasugune vajadus inimlikuks sekkumiseks töövahendite käivitamiseks. Taxifys kasutatakse juba *Jenkins CI* süsteemi, mis jookseb *Linuxi* virtuaalmasinas. Kui Taxify *Android* rakendusi saaks seal edukalt testida, siis *iOS* rakenduste jaoks on vaja eraldi

OSX operatsioonisüsteemiga masinat. Esialgu aga piisaks *Android* kliendirakenduse *CI* süsteemile üle viimine.

Kogu välja arendatud automaattestimise projekt on kättesaadav lisas 3.

Kokkuvõte

Antud lõputöö eesmärgiks oli arendada välja platvorm Taxify Android kliendirakenduse automaattestimiseks ning luua sellele platvormile paar testjuhtumit, millega kaetaks ära mingi funktsionaalne osa rakendusest.

Lõpptulemusena arendati töö käigus välja:

- Automaattestimise platvorm, millel on valmidus Taxify Android kliendirakenduse funktsionaalsuse automaattestimiseks
- Arendati viis automaattesti kliendirakenduse sisselogimise funktsionaalsuse toimimise kontrollimiseks ning analüüsiti nende tulemusi.

Antud lõputöö tingis probleem, et manuaalsele testimisele kulub iga arendustsükli käigus liiga palju aega ning need testid tuleks automatiseerida. See probleem lahendati ainult osaliselt, kuid selleks loodi suurem eeldus. Edasiarendusena tuleks arendada automaatteste ka rakenduse teistele funktsionaalsetele osadele. See on pikk ja väga mahukas protsess. Töö käigus sai paika pandud, kuidas tuleks teste arhitektuuriliselt ehitada, mille põhjal otsustada testjuhtumite olulisust ning kuidas tulemusi analüüsida.

Summary

The aim of the thesis was to develop automated testing platform for Taxify Android client application and create couple of test cases which would cover up some part of the functionality for this application.

As a result, the following was developed:

- Automated testing platform which can be used to develop automated tests for Taxify Android client application.
- Five test cases that covered up login functionality of this client application.

The problem was, that during each development cycle manual testing is taking too much time to complete and these tests should be automated. This problem was solved partially. As a future development every part of the functionality should be covered with automated tests, but this is long and very capacious process. During work it was explained how the architecture of the tests should be and how prioritize test cases and analyze the results.

Kasutatud kirjandus

[1] Hands-On Mobile App Testing: A Guide for Mobile Testers and Anyone Involved in the Mobile App Business. Knott, Daniel. Pearson Education : Crawfordsville, 2015.

[2] Appiumi: Getting Started. [WWW] http://appium.io/slate/en/master (23.04.2016)

[3] TestNG: Documentation. [WWW] http://testng.org/doc/documentation-main.html (19.04.2016)

[4] Genymotion: User Guide. [WWW] https://docs.genymotion.com/pdf/PDF_User_Guide/Genymotion-2.6.0-User-Guide.pdf (21.04.2016)

[5] Amazon AWS: Gradle – Build Automation Evolved. [WWW] https://s3.amazonaws.com/summit2013.gradleware.com/gradle-workshop/gradle_workshop.pdf (8.05.2016)

[6] AWS Device Farm Gradle Plugin. [WWW] http://docs.aws.amazon.com/devicefarm/latest/developerguide/aws-device-farm-android-gradle-plugin.html (8.05.2016)

[7] Gradle Plugin for Genymotion. [WWW] https://www.genymotion.com/release-notes/%23!/developers/gradle-plugin (8.05.2016)

[8] Toughworks: Continuous Integration. [WWW] https://www.thoughtworks.com/continuous-integration (9.05.2016)

Lisa 1 – Sisselogimise testjuhtumid

```
    @Test
    public void loginSuccessfully() {
        signUpLoginView.loginButtonClick();
        loginView.logIn(SUCCESSFUL_LOGIN_NUMBER);
        Assert.assertTrue(codeConfirmView.checkIfConfirmCodeFieldExists());
        codeConfirmView.insertConfirmCode("1357");
    }
    @Test
    public void phoneNumberInvalid() {
        signUpLoginView.loginButtonClick();
        loginView.logIn(INVALID_PHONE);
        Assert.assertEquals(common.getMessage(), "The phone number is
incorrect, country code is required");
    }
    @Test
    public void userNotFound() {
        signUpLoginView.loginButtonClick();
        loginView.logIn(USER_NOT_FOUND_PHONE);
        Assert.assertEquals(common.getMessage(), "There is no user with this
number");
    }
    @Test
    public void userBlocked() {
        signUpLoginView.loginButtonClick();
        loginView.logIn(USER_BLOCKED_PHONE);
        Assert.assertEquals(common.getMessage(), "Your account has been
blocked");
    }

    @Test
    public void loginViewToBackground() {
        signUpLoginView.loginButtonClick();
        common.closeAppToBackground();
        Assert.assertTrue(loginView.checkIfLoginViewOpen());
    }
```

Lisa 2 – Ekraanitõmmiste tegemise näide

```
public class ScreenshotFactory extends Dependency implements ITestListener {

    public void onTestStart(ITestResult result) {
    }

    public void onTestSuccess(ITestResult result) {
    }

    public void onTestFailure(ITestResult result) {
        captureScreenShot(result);
    }

    public void onTestSkipped(ITestResult result) {
    }

    public void onTestFailedButWithinSuccessPercentage(ITestResult result) {
    }

    public void onStart(ITestContext context) {
        System.out.println();
    }

    public void onFinish(ITestContext context) {
    }

    public void captureScreenShot(ITestResult name) {

        String destDir = "";
        String testName = name.getMethod().getRealClass().getSimpleName() +
"." + name.getMethod().getMethodName();

        File scrFile = driver.getScreenshotAs(OutputType.FILE);
        DateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy_HH-mm-ss");

        destDir = "FailedScreenshots";

        new File(destDir).mkdirs();
        String destFile = testName + " - " + dateFormat.format(new Date()) +
".png";

        try {
            FileUtils.copyFile(scrFile, new File(destDir + "/" + destFile));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Lisa 3 – Automaattestimise projekti lähtekood

Bakalaureuse töö käigus loodud automaattestimise projekti lähtekood koos lisadega on kättesaadav aadressil: https://www.dropbox.com/s/kvwke9icty9tk9d/Gert%20Valdek%20120947%20IAPB%20l%C3%B5put%C3%B6%C3%B6%20LISA%203.zip?dl=0