jamk.fi

Automatisoitu Android hyväksymistestaus

Sovelluksessa Figi.fm

Mika Lehtinen

Opinnäytetyö
Huhtikuu 2017
Tekniikan ja liikenteen ala
Insinööri (AMK), ohjelmistotekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Lehtinen Mika	Opinnäytetyö, AMK	4.2017
	Sivumäärä 37	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Automatisoitu Android hyväksymistestaus
Sovelluksessa FigiFm

Tutkinto-ohjelma
Ohjelmistotekniikka

Työn ohjaaja(t)
SALMIKANGAS, Esa

Toimeksiantaja(t)
Lookit Design osk

Tiivistelmä

Opinnäytetyön tavoitteena oli toteuttaa Figi.fm Android-sovellukselle järjestelmä, jolla sovelluksen hyväksymistestaus onnistuisi automatisoidusti. Tutkimusten perusteella arvioitiin eri työkalujen soveltuvuutta hyväksymistestauksen toteutukseen.

Projektin aikana tutkittiin erilaisten testausympäristöjen soveltuvuutta testattavan sovelluksen testaus tarpeisiin. Projekti toteutettiin valitsemalla parhaiten soveltuvat työkalut testaustiimin tarpeisiin.

Projektin lopputuloksena luotiin järjestelmä, joka mahdollistaa sovelluksen toiminnallisuuden testauksen ilman käyttäjää. Testauksen automatisointia pyrittiin viemään pidemmälle integroimalla Jenkins automatisointi palvelin osaksi kehitysympäristöä.

Opinnäytetyön tuloksena saatiin myös kattava kuva Android-sovellusten testaukseen käytettävistä työkaluista. Työ antaa kuvan millaisia vahvuuksia ja heikkouksia kullakin työkalulla on, ja mitä tulee ottaa huomioon yleisesti sovelluksen testausta suunniteltaessa. Työssä käsitellään myös lyhyesti sovellustestauksen eri menetelmiä ja teoriaa.

Projektin lopputuloksena oli sovelluksen lähes kokonaisuudessaan kattava testausjärjestelmä, joten suurin osa tavoitteista saavutettiin. Kesken jääneet osat ovat suunnitelmissa viedä loppuun jatkokehityksen aikana.

Avainsanat (asiasanat)

Android, Espresso, Spoon, hyväksymistestaus, testausautomaatio

Muut tiedot

jamk.fi

Description

Author(s)	Type of publication	Date
LEHTINEN, Mika	Bachelor's thesis	4.2017
	Number of pages 37	Language of publication: Finnish
		Permission for web publication: x

Title of publication
Automatic Android acceptance testing
Software FigiFm

Degree programme
Software Engineering

Supervisor(s)
SALMIKANGAS, Esa

Assigned by
Lookit Design osk

Abstract

The aim of the thesis was to implement to Figi.fm Android app a system that would allow automation of the application's acceptance testing. Based on the research, the suitability of different tools for the acceptance testing was evaluated.

During the project, the suitability of different testing environments was evaluated for testing the application to be tested. The project was implemented by selecting the most suitable tools for the testing team's needs.

As the result of the project, a system allowing testing of the application's functionality without user was created. The aim of the testing automation was to go further by integrating the automation server Jenkins into the development environment.

The thesis resulted in a comprehensive picture of the tools used to test Android applications was also obtained. The thesis gives an idea of the strengths and weaknesses of different tools, and what should be taken into consideration when planning software testing. The paper also discusses shortly different methods and theory of software testing.

As the end result of the project, a testing system capable of testing most of the application was created and as such, most of the goals were achieved. The remaining parts are planned to be completed during further development of Figi.fm.

Keywords (subjects)

Android, Espresso, Spoon, acceptance testing, testing automation

Miscellanous

SISÄLTÖ

TAULUKOT

KUVIOT

KÄSITTEET

Aktiviteetti – Käyttäjälle luotu kehys, jossa sovelluksen käyttöliittymä sijaitsee.

Android – Googlen kehittämä käyttöjärjestelmä mobiililaitteille.

Android Studio – Googlen kehittämä kehitysympäristö Android-sovellusten rakentamiseen

Android Support Repository – Kokoelma kirjastoja jotka lisäävät ominaisuuksia kehitysympäristöön.

APK – Android application package, eli Android-sovellusten tiedostotyyppi.

Back-end – Sovellus infrastruktuurissa käytetty termi datan käsittely kerroksesta, kuten esimerkiksi tietokanta.

Debugger – Virheenjäljittäjä, käytetään ohjelmointivirheiden löytämiseen ja korjaamiseen.

Emulaattori – Virtuaalinen ympäristö, joka mahdollistaa ohjelman käytön laitteella jolle sitä ei ole suunniteltu.

Git – Linux pohjainen ohjelma, joka mahdollistaa helpon versionhallinnan käyttäjälle

Hyväksymistestaus – Testauksen vaihe, jossa tutkitaan täyttääkö järjestelmä asiakkaan vaatimukset.

JAR – Java Archive, eli Java tiedostojen pakkausmuoto.

Jenkins – Automatisointi palvelin joka mahdollistaa sovelluksen jatkuvan integraation

Natiivisovellus – Sovellus, joka on kehitetty toimimaan vain yhdellä alustalla.

REST-API – Rajapinta, joka mahdollistaa sovelluksen kommunikaation esimerkiksi tietokannan kanssa.

SDK manager – Android Studion työkalu, jonka kautta on mahdollista asentaa kehitysympäristöön lisäosia, kuten Android Support Repository

Scripti – Ohjelma, tai sarja komentoja jotka käyttöjärjestelmä tulkitsee, ja toimii sen mukaisesti.

1. Työn lähtökohdat

1.1 Tavoitteet

Opinnäytetyön tavoitteena oli tutkia automatisoidun sovellustestauksen toteutusta Figi.fm Android-sovellukselle. Tarkoituksena oli tutkia erilaisia vaihtoehtoja Android-sovellusten testaukseen ja toteuttaa se parhaaksi nähdyllä tavalla toimeksiantajan käyttötarkoituksia varten. Läpikäydyistä vaihtoehdoista arvioidaan niiden soveltuvuutta ja sitä, kuinka hyvin niitä voidaan käyttää päivittäisessä sovelluskehityksessä. Lopullisena päämääränä oli tutkia mikä vaihtoehdoista olisi paras, ja toteuttaa koko sovelluksen kattava testausjärjestelmä.

1.2 Työn taustat

Opinnäytetyön aihe ja taustat liittyvät Jyväskylän ammattikorkeakoulussa 2013 alkaneeseen Ohjelmistoprojekti-kurssin asiakasprojektiin. Kurssilla valmistettiin pienissä ryhmissä koulun ulkopuolisille asiakkaille ohjelmistoprojekti. Oman ryhmäni asiakkaana oli LookIT osk, jonka toimeksiantona aloitettiin tekemään Android-sovellusta Figi.fm. Projekti jatkui edellä mainitun kurssin jälkeenkin, kun osa ryhmän jäsenistä jatkoi sovelluksen kehittämistä työharjoittelun merkeissä. Opinnäytetyön aihe tulikin aika luonnollisesti, kun sovellus alkoi lähestyä ominaisuuksiltaan valmista. Tässä vaiheessa hyväksymistestauksen merkitys nousi esille vahvasti, jotta loppukäyttäjälle päätyvä sovellus olisi paras mahdollinen.

1.3 Figi.fm

Figi.fm on idealtaan live-musiikin ystäville ennestään tuttu sovellus, jonka tarkoituksena on tehdä uusien live-kokemusten löytämisestä entistä helpompaa. Lisäksi tarkoituksena on helpottaa artistien ja tapahtumanjärjestäjien kommunikaatiota fanien kanssa.

Android-sovellus

Projektin aikana valmistettu Android-sovellus on toiminnallisuudeltaan mobiilisovellukseksi melko laaja. Laajuuden vuoksi testauksen tarpeellisuus korostuu, sillä yksikin virhe voi hajottaa koko järjestelmän. Sovelluksen toimintoihin kuuluu muun muassa:

- Lähellä olevien tapahtumien haku GPS:n avulla
- Uusien tapahtumien suosittelu puhelimella kuunnellun musiikin perusteella
- Helppo navigointi tapahtumiin Google Maps -integraation avulla
- Muistutukset tapahtumista sekä kalenterin että sovelluksen kautta lähetettävien ilmoituksien avulla
- YouTube-integraatio tapahtumiin artisteihin tutustumisen helpottamiseksi

Back-end

Luonnollisesti edellä mainittua laajuutta ei pystytä toteuttamaan pelkästään Android-sovelluksessa järkevällä tavalla. Sovelluksen selkärankana toimiikin REST-API, jonka kautta syötetään tietoja Android-sovellukselle. Tässä tapauksessa palvelimella pyörivällä järjestelmällä oli kuitenkin pelkän tietokannan sijaan tärkeä osa myöskin tiedonkeräyksen suhteen. Koska tapahtumanjärjestäjille ja artisteille ei ole olemassa yhtenäistä paikkaa josta tietoa voitaisiin hakea, sitä joudutaan keräilemään useasta paikasta järjestelmään. Tärkeässä osassa tässä osuudessa oli tunnettu internetyhteisö Last.fm, jonka avulla tapahtumista ja artisteista saatiin tietoa kerättyä tehokkaasti.

2. Ohjelmistotestaus

2.1 Johdanto

Ohjelmiston testauksen pääasiallisena tarkoituksena on tutkia ohjelman virheettömyyttä ja varmistaa yleisesti sovelluksen laatu ennen loppukäyttäjälle päätymistä. Testaaminen siis tarjoaa sekä kehittäjälle että käyttäjälle arvokasta tietoa ohjelmiston valmiudesta ja yleisestä laadusta.

Ohjelmistotestaus yksinkertaisuudessaan tarkoittaa sitä, että sovellus (tai sen osa), suoritetaan ja sitä verrataan ennalta odotettuun tulokseen. Ohjelmistotestausta käsittelevässä wikipedia-artikkelissa (Software testing wiki, n.d) mainitaan muun muassa seuraavat asiat, joita kannattaa ottaa huomioon testattaessa:

- Yleinen käytettävyys
- Nopeus, esimerkiksi verkkosivun vasteaika
- Eri käyttöalustojen toimivuus
- Vastaa käyttäjän tai muun sidosryhmän odotuksia toiminnallisuudesta

Yleisenä käytäntönä ohjelmistoprojekteissa on luoda jo hyvin alkuvaiheessa selkeä suunnitelma siitä, kuinka sovelluksen testaus toteutetaan. Suunnitelma sisältää yleisesti sen, mitä tavoitteita testauksella on ja kuinka testaus käytännössä toteutetaan. Tärkeitä huomioitavia asioita suunnitelman luonnissa ovat muun muassa käyttäjien oletetut vaatimukset, käytettävissä olevat työkalut sekä se, kuinka paljon projektissa on resursseja testaukseen annettavana.

2.2 Testauksen tavoitteet

Sovellustestauksen tarpeellisuus on peräisin kaikille ihmisille tutusta lausahduksesta: jokainen tekee virheitä. Virheiden vaikutukset vaihtelevat tapauskohtaisesti harmittomista kirjoitusvirheistä aina vakaviin tietoturvaongelmiin asti. Tästä syystä sovelluskehityksessä onkin erittäin tärkeää aina olettaa, että kehityksen aikana on sattunut virheitä.

Sovellustestaus on viime vuosien aikana siirtynyt vahvasti kohti automatisointia, jotta kehittäjän tekemää manuaalista testausta saataisiin vähennettyä. Ohjelmoijan tekemä käytännön testaus on hyvin tehoton tapa testata sovellusta jonka toimintamalli ei muutu projektin edetessä. Tästä syystä automatisoitu testaus onkin tärkeä toimenpide, jolla karsitaan kehityksen aikaisia kustannuksia. Se on myös erittäin hyvä tapa seurata projektin edetessä sovelluksen yleistä tilaa. Jos uusi ominaisuus aiheuttaa ongelmia jo olemassa oleviin osiin, automatisoitu järjestelmä auttaa ongelman löytämisessä ja ratkaisemisessa.

Testauksesta onkin edellä mainitusta syystä tullut yksi tärkeimmistä osista puhuttaessa projektin aikataulusta sekä budjetista. Yrityksen näkökulmasta puutteellinen testaus voi aiheuttaa ylimääräisiä kustannuksia tai mahdollisia tulon menetyksiä monella eri tavalla. Kustannuksia aiheuttaa luonnollisesti läpi päässeen virheen korjaus, sekä ylimääräiset henkilöstökustannusket esimerkiksi asiakaspalvelun vuoksi. Toisaalta, sovelluksessa oleva virhe voi aiheuttaa myytävälle tuotemerkille tai yritykselle ylipäätään suuria tulonmenetyksiä heikentyneen maineen vuoksi. Jos sovelluksen käytöstä jää asiakkaalle huono maku suuhun, on todennäköistä, ettei käyttäjä enää palaa käyttämään sovellusta.

Otetaan testauksen heikkouden aiheuttamista ongelmista käytännön esimerkki: Niantic Labsin kehittämä Pokemon Go. Kyseinen mobiilipeli saavutti erittäin nopeasti valtavan suosion maailmanlaajuisesti ja parhaimmillaan pelillä oli noin puoli miljardia pelaajaa. Yrityksen kannalta valitettavasti joitakin ominaisuuksia pelissä ei oltu testattu tarvittavassa määrin, joka johti suureen määrään negatiivista julkisuutta. Ongelmista johtunut julkisuus johtikin hyvin nopeasti suureen määrään menetettyjä käyttäjiä, joka aiheutti tuntuvan loven yrityksen tulokseen.

2.3 Testauksen kohteet

Ihanteellisessa tilanteessa kaikki mahdollinen testataan. Mitä suurempi osa ohjelman toiminnallisuudesta on testattu, sitä varmempaa sovelluksen oikeanlainen toiminta on. Valitettavasti suuressa osassa tilanteista kaiken testaaminen ei ole mahdollista, vaan keskitytään sovellukselle tärkeisiin ominaisuuksiin. Kun tarkastellaan mitä testataan, Android-sovelluksien testaus eroaa merkittävästi esimerkiksi perinteisten Web-sivujen testauksesta.

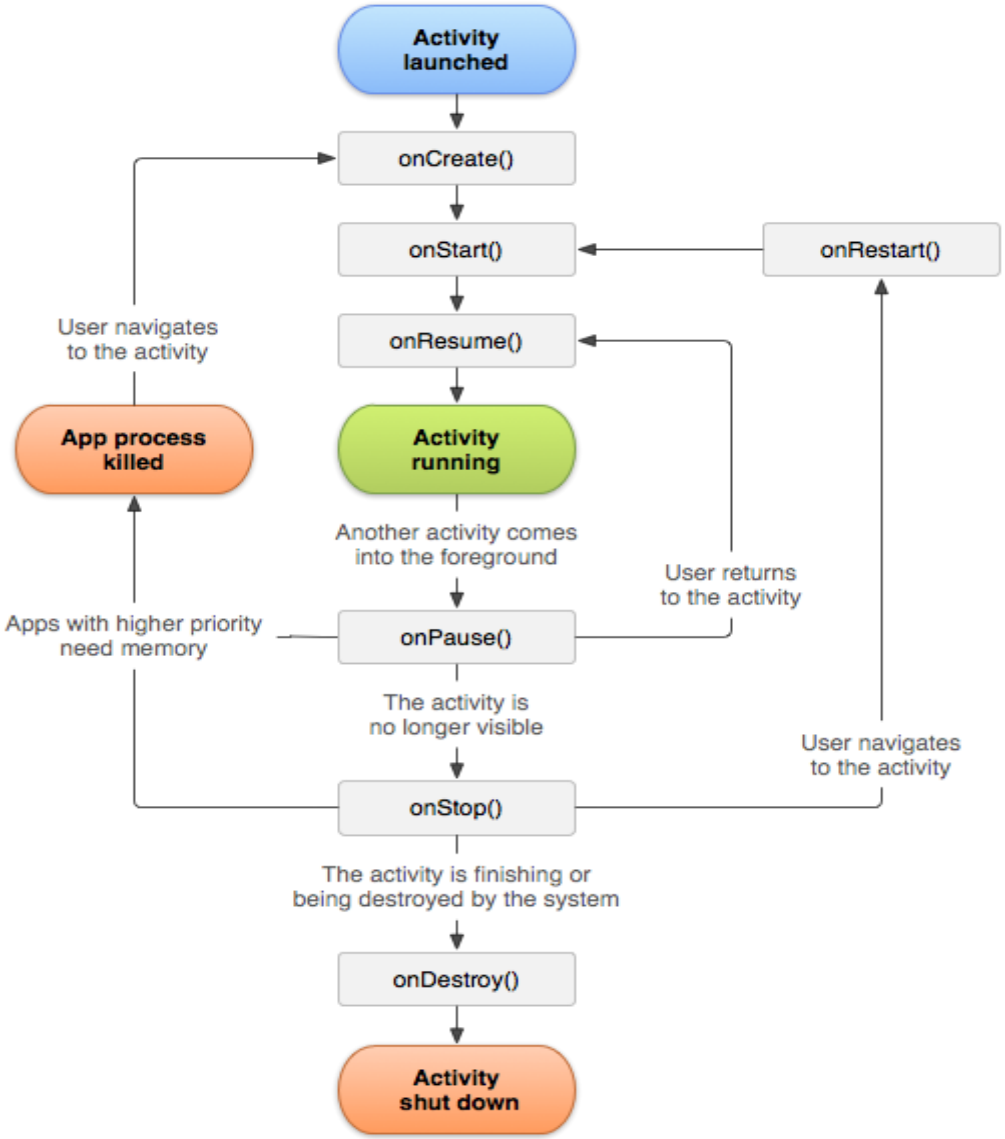
Aktiviteetit

Android-sovelluksesta puhuttaessa käyttäjän kannalta tärkein asia on aktiviteetit ja niiden elinkaaren metodit. Aktiviteetit kontrolloivat kaikkea tietoa jonka käyttäjä näkee, joten niiden toiminnan takaaminen on erittäin tärkeää. Aktiviteetin tila voidaan jakaa neljään eri tilaan: (Android developer manual, activity n.d.)

- Aktiivinen tai käynnissä oleva tila jossa sovellus on päällimmäisenä sekä käyttäjän ruudulla, että aktiviteettipinossa.
- Aktiviteetti saattaa menettää huomion (focus), mutta olla edelleen näkyvissä. Kyseinen tilanne tulee vastaan, jos jokin osittain ruudun peittävä tai läpinäkyvä aktiviteetti on edellä mainitun aktiviteetin päällä. Aktiviteetin tila on tässä vaiheessa keskeytynyt (paused). Se on edelleen käynnissä, säilyttää kaiken tietonsa ja on yhteydessä ikkunakäsittelyyn (window manager). Äärimmäisessä tilanteessa tällainen aktiviteetti saatetaan kuitenkin tuhota, kuten muistin ollessa vähissä.
- Aktiviteetti saattaa olla myös täysin pysähtyneessä tilassa (stopped). Tämä tila saavutetaan tilanteessa jossa aktiviteetti ei ole lainkaan näkyvissä käyttäjälle. Aktiviteetti säilyttää yhä kaikki tietonsa, mutta on erittäin todennäköisestä, että se tuhoutuu järjestelmän toimesta muistin vapauttamiseksi
- Kun aktiviteetti on joko keskeytynyt tai pysähtynyt, järjestelmä voi tällöin vapauttaa muistia tuhoamalla aktiviteetin. Kun tuhottuun aktiviteettiin

palataan, se joudutaan luomaan uudestaan ja palauttamaan aiempaan tilaan kaikkine tietoineen.

Kuviosta 1 nähdään yhden aktiviteetin elinkaari. Ovaalin muotoiset kuviot edustavat edellä mainittuja erilaisia tiloja, kun taas neliön muotoiset kuviot ovat erilaisia metodeja, joita käytetään aktiviteetin tilojen tutkimiseen ja käsittelyyn eri siirtymävaiheissa.



Kuvio 1. Android-aktiviteetin elinkaari ja siirtymien metodit (Android developer manual, activity. n.d.)

Kuviosta nähdään myös, kuinka metodit voidaan jakaa kolmeen pääryhmään joita olisi hyvä tarkkailla. (Android developer manual, activity n.d.)

- Yksittäisen aktiviteetin koko elinkaari sijoittuu onCreate() ja onDestroy()-metodien väliin. Aktiviteetin onCreate() metodia kutsutaan ainoastaan kerran, luonnollisesti kyseistä aktiviteettiä luotaessa. Metodi alustaa kaikki globaalit muuttujat ja resurssit, joita aktiviteetin muut osat tulevat tarvitsemaan. Kyseistä metodia käytetään myös, jos aktiviteetin tila halutaan palauttaa ennen lopetusta olleeseen tilaan. Aktiviteetin tuhoutuessa kutsutaan onDestroy() metodia, joka vapauttaa aktiviteetin varaamat resurssit muuhun käyttöön.
- Aktiviteetin näkyvä elinkaari sijoittuu metodien onStart() ja onStop() välille. Tämän aikana aktiviteetti on varmasti näkyvissä, vaikkei välttämättä päällimmäisenä. Molempia metodeja voidaan kutsua useasti, jos aktiviteetin näkyvyys muuttuu. Aktiviteetin palatessa näkyviin kutsutaan onRestart() metodia, jota seuraa aina onStart()-metodi.

- Etualan elinkaari sisältää ajan, jolloin sovellus on päällimmäisenä ja käyttäjä on aktiivisesti yhteydessä siihen. Kyseisestä tilasta poistutaan ja palataan takaisin erittäin usein, joten on tärkeää pitää onResume()- ja onPause()-metodit mahdollisimman keveinä.

Fyysiset ominaisuudet

Mobiilisovellusta kehittäessä on otettava huomioon valtava määrä erilaisia laitteita, joita käyttäjillä mahdollisesti on. Sovellusta kehittäessä on pidettävä mielessä mahdolliset laitteelta vaadittavat ominaisuudet joita sovellus käyttää. Tästä syystä sovellusta tulisi testata mahdollisimman laajalla määrällä laitteita, jotta varmistetaan ohjelman toiminta ilman vaadittuja ominaisuuksia tai vähintään käsitellään niiden puute oikein. Jatkuva uusien älylaitteiden kuten älykellojen ja televisioiden tuonti markkinoille tuo tähän osioon ison kasan uusia haasteita. Sovellukselle on mahdollista määritellä, mihin laitteisiin se voidaan asentaa, mutta sen varaan ei ole suositeltava laittaa sovelluksen toimivuutta.

Android-laitteille yleisesti kannattaa ottaa huomioon ainakin seuraavat fyysiset ominaisuudet:

- Verkkoyhteys
- Erilaiset näyttökoot ja -resoluutiot
- Näppäimistön eroavuudet laitteiden välillä
- Muistikortti

Listassa mainittuja ominaisuuksia voidaan onneksi mallintaa jo erittäin hyvin emulaattoreita käyttämällä. Luonnollisesti virtuaalisella ympäristöllä ei saada kuitenkaan täysin aukotonta kuvaa, vaan lopullinen kuva sovelluksen toimivuudesta eri laitteilla saadaan sovelluksen beetatestauksesta käyttäjien omilla laitteilla.

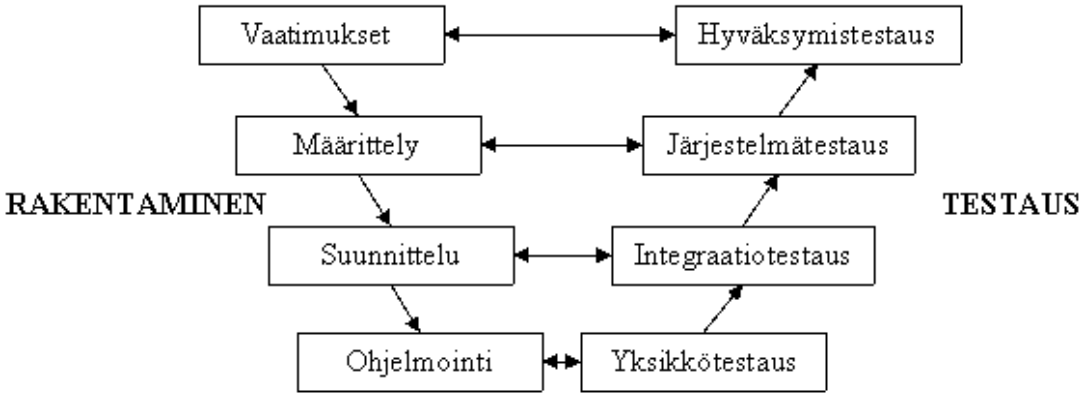
3. Testauksen teoria

3.1 Yleistä

Ohjelmistotestauksen päämääräisenä tavoitteena on selvittää, vastaako sovellus ennestään annettuja vaatimuksia. Kyseiset vaatimukset voivat olla muun muassa, vastaako toiminnaltaan tilaajan toiveita, sovelluksen toimintanopeus, yleinen käytettävyys. Koska jokaiselle sovellukselle on lähes ääretön määrä testejä, joita voidaan suorittaa, on syytä muodostaa suunnitelma, jossa käydään läpi, mitkä ovat jokaisen testin hyväksytyt ja hylätyt lopputulokset.

3.2 Testauksen eri tasot

Testausta käsiteltäessä on tärkeää ottaa huomioon ohjelmistotestauksen eri tasot, jotka jaetaan yleensä kolmeen eri kategoriaan. Kuviosta 2 voidaankin nähdä sovelluskehityksen yleinen V-malli ja kuinka mikäkin testauksen taso siihen sijoittuu.



Kuvio 2. Ohjelmistoprojektin V-malli havainnollistettuna. (Mikko Mäki-Rahkola testaussuunnitelma. 2003.)

Yksikkötestaus

Yksikkötestauksella tarkoitetaan testaustapaa, jossa tutkitaan yksittäisten metodien tai muiden vastaavien pienten osien toimintaa. Tällaisten testien tarkoitus on varmistaa, että sovelluksen pienet palaset käsittelevät saamaansa tietoa oikein ja vastaavat suunniteltua toiminallisuutta. Realistisesti nämä testit ovat käytössä vain varhaisessa kehitysvaiheessa helpottamassa ohjelmoijan virheiden löytymistä. Sovelluksen valmistumista lähentyessä yksikkötestien skaalaus on usein liian työlästä niiden tuomiin etuihin verrattaessa. (Unit testing wikipedia, n.d.)

Integraatiotestaus

Integraatiotestauksella tarkoitetaan testauksen vaihetta, jossa tutkitaan suurempien kokonaisuuksien toiminnallisuutta sekä sovelluksen toiminnan tehokkuutta. Tähän tasoon päästäessä sovelluksen yksittäisten metodien tai muiden pienten osien ei enää pitäisi aiheuttaa virheitä, sillä ne on tarkoitus eliminoida yksikkötestauksen puitteissa. Integraatiotestauksessa virheitä pitäisi tulla vastaan vain tilanteissa, joissa useiden metodien yhteistoiminta ei jostain syystä toimi. (Integration testing wikipedia, n.d.)

Tämän tyyppiseen testaukseen on monia erilaisia lähestymistapoja riippuen siitä, kuinka se halutaan toteuttaa. Yleisenä tapana on suorittaa ns. Big Bang -testaus, jossa ohjelmisto lyödään kokonaisuudessaan yhteen ja toivotaan, ettei kovin moni asia hajoaisi. Tämän ratkaisun haittana on luonnollisesti se, ettei testauksen tekemiselle ja sen aikana löytyneiden virheiden korjaamiselle jää kovin paljoa aikaa. Suurien

ohjelmistojen tapauksessa on myös haastavaa löytää, minkä osan aiheuttama mahdollinen virhe on, kun kaikki osat laitetaan yhteen. Tämän tyylinen toteutus käy helposti jatkokehityksen kannalta huomattavasti kalliimmaksi, kuin oli alustavasti suunniteltu.

Suositellumpana testausmenetelmänä integraatiotestaukseen on niin sanottu jatkuvan integroinnin testausmenetelmä. Jatkuvan integraation perusideana on se, että sovelluksen eri palat kootaan yhteen jatkuvasti. Tällä tavoin varmistetaan se, ettei sovelluksen eri osien yhteen kokoaminen tuota ongelmia. Testauksen kannalta tällainen toimintamalli on joissakin tapauksissa parempi, sillä testaus voidaan implementoida yksittäisten osien lisäksi myös eri osien väliseen toimintaan. Suurissa ohjelmistoprojekteissa jatkuva integraatio saattaa aiheuttaa testaukseen myös ongelmia, sillä siihen tarvittavan järjestelmän rakentaminen vaatii myös paljon resursseja.

Hyväksymistestaus

Ylimpänä testauksen tasona on hyväksymistestaus. Hyväksymistestauksen tarkoituksena on tutkia sitä, saavuttaako sovellus tai järjestelmä loppukäyttäjän sille asettamia vaatimuksia. Koska kyseessä on testauksen viimeinen taso, voidaan siihen sisällyttää myös käyttöohjeiden ja muun dokumentaation hyväksyntä, jos se on aiemmissa vaiheissa jätetty tekemättä.

Käytännössä järjestelmätestaus ja hyväksymistestaus ovat suurelta osin samoja asioita, sillä molemmissa tutkitaan sovelluksen toimivuutta sen ollessa hyvin lähellä valmista lopputuotetta. Suurin ero näiden testausvaiheiden välillä on se, että hyväksymistestaus mielletään yleisesti ottaen asiakkaan suorittamaksi testaukseksi. Lähteistä riippuen alpha- ja beetatestaus ovat tai eivät ole osa hyväksymistestausta, tai sitten ne voivat tulla myös hyväksymistestauksen jälkeen.

Hyväksymistestaus voidaan suorittaa myös yrityksen sisäisesti, ja sen tyyppisestä testauksesta on tässä opinnäytetyössäkin kysymys. Tällaisessa tapauksessa arvioidaan sovellusta käyttäjän näkökulmasta, joten testien ennalta arvattavuus ja tulosten tallentaminen ovat erittäin tärkeää. Yksinkertaisesti muotoiltuna hyväksymistestaus on juurikin sitä miltä se kuulosta, tapa varmistaa hyväksyttävä laatu ennen loppuasiakkaalle toimitusta.

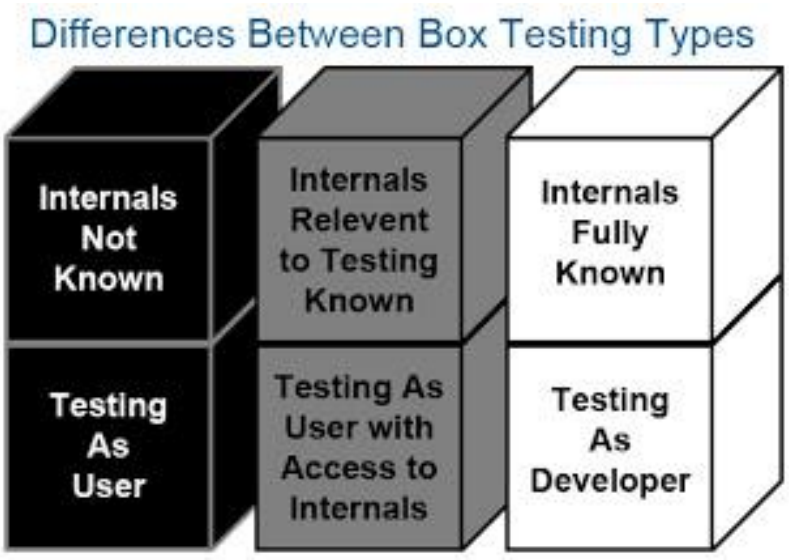
3.3 Testausmallit

Samoin kuin testauksessa on monia eri tasoja, on myös erilaisia malleja, jonka pohjalta voidaan toimia. Yleisesti ne jaetaan kahteen pääryhmään, staattisiin ja dynaamisiin menetelmiin. Staattinen malli on usein sovelluksen kehityksessä tapahtuvaa automaattista testausta, kuten esimerkiksi lähdekoodin syntaksin tarkistus, jonka kehitysympäristön kääntäjä tekee luodessaan valmiin paketin. Staattisessa testauksessa ohjelmaa ei käynnistetä.

Dynaaminen testaus taas on testausta, joka tapahtuu vain ohjelman ollessa käynnissä. Tarkoituksena on käydä läpi sovellusta erilaisilla testitapauksilla tai voidaan käyttää kehitysympäristön debugger-työkalua. Sovellusta käydään läpi systemaattisesti kehittäjän toimesta, jotta virheet löydetään ajoissa.

Laatikkomalli

Toiminnallista testausta lähestytään yleensä ns. laatikkomallilla (ks. kuvio 3). Testaus jaetaan tässä tapauksessa kolmeen eri laatikkoon eli musta-, lasi- ja harmaalaatikkotestaukseen.



Kuvio 3. Laatikkomallien sisältö havaillistettuna. (IT Audit Academy n.d.).

Mustalaatikkotestaus

Kun puhutaan mustalaatikkotestauksesta, puhutaan prosessista jossa ohjelman sisäisestä toiminnasta ei ole testaajalla tietoa. Testaaja on tietoinen siitä mitä sovelluksen olisi tarkoitus tehdä, mutta ei sitä, kuinka se tehdään. Tarkoituksena tällaisessa testauksessa on havaita suurempia loogisia virheitä, joita ohjelmoija on saattanut kehityksen aikana tehdä. Otetaan esimerkiksi metodi, jonka toiminnallisuutena on tulostaa käyttäjän tekstikenttään kirjoittama teksti, kun näytöllä olevaa nappulaa painetaan. Tämän tapaisen metodin testaus onnistuu hyvin ilman tietoa taustalla olevasta koodista, ja on myös hyvin helppo huomata, jos lopputulos ei vastaa odotuksia. (Black-box testing wiki. n.d.)

Harmaalaatikkotestaus

Harmaalaatikkotestauksella tarkoitetaan testausta, jossa testaajalla on rajallinen määrä tietoa sovelluksen toiminnallisuudesta kooditasolla. Yleisesti ottaen harmaalle alueelle päätyvät testit jotka ovat hieman käyttäjän toimintoja teknisesti korkeammalla tasolla, kuten esimerkiksi tietokantoihin liittyvät testit. Tavoitteena on saada tietoa mahdollisista virheistä tiedonkulussa tai järjestelmien yhteensopivuudesta. Käytännön esimerkkinä voidaan ottaa Web-sovellus, joka hakee käyttäjän profiilitietoja tietokannasta. Testaaja tietää tällaisessa tapauksessa mitä tietoja pitäisi saada, ja miltä sen tulisi loppukäyttäjän ruudulla näyttää. (Gray-box testing wiki. n.d.)

Lasilaatikkotestaus

Lasilaatikkotestaus taas on puhtaasti sovelluksen kehittäjälle tarkoitettu testausmalli. Tässä tapauksessa oletetaan, että testaajalla on tieto kaikesta, mitä sovelluksen sisällä tapahtuu. Testauksen tarkoituksena ei enää tässä vaiheessa ole tarkoitus vain testata antaako sovellus oikean tuloksen, vaan testataan sitä, että antaako sovellus halutun tuloksen oikealla tavalla. Esimerkkinä tällaisesta testauksesta voisi olla metodi, joka ottaa vastaan minkä tahansa kokonaisluvun, mutta tulostaa vain kokonaisluvut jotka ovat parillisia. Ilman tietoa metodin tarkasta toiminnallisuudesta, virheen havaitseminen parittomilla luvuilla saattaisi jäädä helposti huomaamatta. (White-box testing wiki. n.d.)

4. Android-testauksen työkalut

4.1 Android-käyttöjärjestelmä

Android on Googlen omistama Linux-pohjainen mobiilikäyttöjärjestelmä. Android on tällä hetkellä ylivoimaisesti suosituin käyttöjärjestelmä, kuten taulukosta 1 voidaan nähdä. Suurista puhelinvalmistajista mm. Samsung, Sony ja HTC käyttävät kyseistä käyttöjärjestelmää, joka on viime vuosien aikana vahvistanut sen asemaa.

Taulukko 1. Mobiililaitteiden käyttöjärjestelmien suosio (IDC markkinatutkimus, 2016)

Period	Android	iOS	Windows Phone	Others
2015Q4	79.6%	18.7%	1.2%	0.5%
2016Q1	83.5%	15.4%	0.8%	0.4%
2016Q2	87.6%	11.7%	0.4%	0.3%
2016Q3	86.8%	12.5%	0.3%	0.4%

Source: IDC, Nov 2016

Androidin pääasiallisena ohjelmointikielenä on Java, mutta kehitys on mahdollista myös C- ja c++ kielillä. C pohjaisten kielien käyttö vaatii tosin Native Developement Kit:in, jonka käyttö on huomattavasti kankeampaa kuin normaali kehitysympäristö. Googlen kehittäjille tarjoama Android Studio -kehitysympäristö ohjaa kehittäjiä vahvasti Javan puolelle olemalla sen ainut tuettu kieli.

4.2 Työkalujen valinta

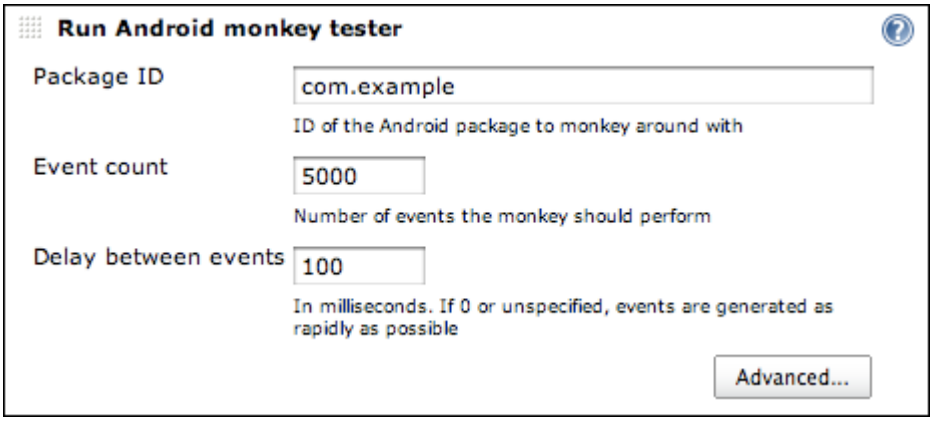
Luvussa 3 käsiteltiin testauksen perusajatuksia ja erilaisia lähtökohtia. Nämä luovat hyvän pohjan sille, kuinka testaus Android-sovelluksissa voitaisiin toteuttaa. Luvuissa 4.2-4.8 tullaan käsittelemään erilaisia työkaluja, joilla testaus voidaan suorittaa, ja arvioidaan niiden hyviä ja huonoja puolia kehittäjän näkökulmasta.

Tutkitut työkalut valittiin sen perusteella, ovatko ne edelleen aktiivisessa kehityksessä, tai ovatko ne laajassa käytössä kehittäjien parissa. Kyseiset valintakriteerit helpottavat huomattavasti ongelmatilanteiden ratkaisussa, koska voi helposti hakea apua olemassa olevalta käyttäjäyhteisöltä.

4.3 Monkey

Monkey on yksi suosituimmista, ja myös yksinkertaisimmista, Android-sovellusten testaukseen käytettävistä työkaluista. Sen tärkeimpänä käyttötarkoituksena on yleisesti sovelluksen suorituskyvyn testaus. Monkey:ta käytetäänkin usein yksikkötestauksen tapaiseen toteutukseen, jossa etsitään funktioiden toiminnallisia virheitä. (Android developer manual, Monkey n.d.)

Käytöltään Monkey on yksi yksinkertaisimmista työkaluista käyttää, sillä se luo testinsä käytännössä automaattisesti. Koska testit luodaan työkalun puolesta, kehittäjän tehtäväksi jää vain työkalun käynnistäminen komentorivin kautta. Testauksen automatisointiin suunnitellut ohjelmat, kuten Jenkins, usein tarjoavat mahdollisuuden ajaa Monkey testit kehittäjän tekemän päivityksen yhteydessä. (Ks. kuvio 4).



Kuvio 4. Monkey testien luonti Jenkinsin kautta. (Jenkins android-plugin dokumentaatio n.d)

Täysin satunnaisesti luodut toiminnot sovelluksessa toimivat hyvin, kun etsitään huonosti optimoituja ratkaisuja tai yllättäviä virheitä sovelluksessa. Monkey on loistava testaustyökalu tehtäessä mustalaatikkotestausta, sillä ohjelman sisällön tietämisestä

ei ole sen käytössä mitään hyötyä. Pääasiassa Monkey on kuitenkin tarkoitettu sovelluksen stressitesteihin, sillä 500 satunnaisen toiminnon tekeminen sovelluksessa ei tarjoa käyttäjäkokemusta vastaavaa palautetta kehittäjälle.

4.4 Robotium

Robotium on toiminnallisuudeltaan erittäin samankaltainen kuin monet tietokoneohjelmille tarkoitetuista testaustyökaluista. Sen käyttötarkoitukset ovat huomattavasti laajemmat kuin esimerkiksi Monkey, sillä Robotiumia voidaan käyttää kaikkeen aina yksikkötestauksesta hyväksymistestaukseen asti. (Robotium-kirjaston dokumentaatio. n.d.)

Android-kehittäjän näkökulmasta Robotium on yksi helpoimmista työkaluista käyttää, sillä sen testit kirjoitetaan samalla ohjelmointikielellä kuin Android-sovellukset-kin, Javalla. Ainut asia, jonka kehittäjä joutuu tekemään, on lisätä Robotiumin JAR-kirjasto kehitysympäristöön.

Robotiumin käyttö vaatii erillisen testiprojektin luonnin jo olemassa olevan ohjelmistopaketin sisälle, sillä se vaatii alkuperäisen sovelluksen toimiakseen. Testien käynnistyessä kehitysympäristö asentaa uusimman version sovelluksesta joko puhelimelle tai emulaattorille kehittäjän valintojen mukaisesti. Kääntämisen seurauksena laitteeseen asennetaan uusin versio apk-tiedostosta, jota vasten testit ajetaan. Robotiumin testien toimintaa on helpoin selittää yksinkertaisen esimerkin kautta (Ks. kuvio 5).

```
solo.clickOnText("More");
solo.clickOnText("Preferences");
solo.clickOnText("Edit File Extensions");
Assert.assertTrue(solo.searchText("rtf"));
```

Kuvio 5. Robotium koodi esimerkki. (Robotium-kirjaston dokumentaatio n.d)

Kuviossa 5 nähdään, kuinka Robotiumia käyttämällä voidaan navigoida testattavan ohjelman sisällä. Solo-objektille annetaan erilaisia käskyjä kuten clickOnText(), clickOnButton(), enterText() yms., joilla voidaan tehdä erilaisia asioita ohjelman sisällä. Yllä oleva testi navigoi läpi ohjelman klikkaamalla sille annettuja tekstipätkiä, ja

lopulta assertTrue()-metodille annetaan ehto, minkä mukaan päätetään, onko testi suoritettu hyväksytysti vai ei. Tässä tapauksessa, jos navigoinnin päätteeksi ohjelmasta löytyy näkyviltä teksti ”rtf”, testi todetaan onnistuneeksi.

Kuten voidaan huomata, Robotium vaatii huomattavan määrän tietoa siitä, mitä ohjelma pitää sisällään. Se onkin kehittäjien parissa hyvin suosittu työkalu hyväksymistestauksen toteutukseen, sillä sen joustavuus mahdollistaa erittäin kattavan testauksen. Koska Robotium luottaa paljolti ohjelmassa näkyvissä oleviin elementteihin, se voi olla arvokas työkalu myös ulkopuoliselle testaajalle, joka tutkii sovelluksen toimivuutta.

4.5 UIAutomator

UIAutomator ei enää varsinaisesti ole oma työkalunsa, vaan kuuluu Espresso -testauspakettiin. Sitä voidaan kuitenkin käyttää myös omillaan, joten sen toiminnasta on hyvä tietää hieman perusasioita.

UIAutomator on erittäin hyödyllinen työväline, kun testataan laajan sovelluksen toiminnallisuutta, joka vaikuttaa asioihin varsinaisen sovelluksen sisällä. Mitä tällä tarkoitetaan, on esimerkiksi sovellukset, jotka tarvitsevat kalenteria, galleriaa, yhteystietoja yms. sovelluksen ulkopuolisia osia, joihin käyttäjä voi kuitenkin ohjelman puitteissa navigoida. Hyvänä esimerkkinä tällaisesta toiminnasta on kuvien lisääminen esimerkiksi Facebook tai WhatsApp sovelluksiin. Sovellukset ohjaavat sinut hakemaan kuvan omasta galleriasta ja palaavat sen jälkeen takaisin sovelluksen sisälle. Tämänkaltaisen käyttäytymisen testaus ei kuitenkaan ole mahdollista perinteisen sovellustestauksen työkaluilla, sillä ne eivät pääse käsiksi testattavan sovelluksen ulkopuolisiin elementteihin. Kuviosta 6 voidaan nähdä se, kuinka sovelluksen sisällä olevan Facebook kirjautumisen testaus voidaan toteuttaa UIAutomatoria käyttäen. (Android developer manual, UIAutomator. n.d.)

```
onWebView(hasFocus()).perform(webClick());
UiObject input = mDevice.findObject(new UiSelector().instance(0).className(EditText.class));
input.setText(getFacebookLoginEmail());
SystemClock.sleep(2000);

UiObject input2 = mDevice.findObject(new UiSelector().instance(1).className(EditText.class));
input2.setText(getFacebookLoginPassword());
SystemClock.sleep(2000);


UiObject buttonConfirm = mDevice.findObject(new UiSelector().instance(0).className(Button.class));
buttonConfirm.click();
SystemClock.sleep(2000);
UiObject confirm = mDevice.findObject(new UiSelector().instance(1).className(Button.class));
confirm.click();
```

Kuvio 6. Esimerkki Facebook-login ominaisuudesta sovelluksessa Figi.fm

UiObject nimisille objekteille annetaan esimerkiksi EditText -elementti ja kyseiselle objektille voidaan sen jälkeen suorittaa erilaisia toimintoja kuten setText(), jolla muokataan olemassa olevaa tekstiä kentässä, tai click(), joka yksinkertaisuudessaan on objektin elementin klikkaus.

UIAutomatorin yksi suuri heikkous ovat sen vaatimat järjestelmän ”pysäytykset” jotta ohjelma pysyy testien mukana. Jos järjestelmää ei jokaisen elementin kanssa tehdyn toiminnon jälkeen pysäytetä on riskinä se, että testi epäonnistuu, koska haetaan elementtiä joka ei vielä ole ehtinyt ilmestymään. Toisin sanottuna on mahdollista, että testi tekee toimintoja nopeammin kuin asiat tapahtuvat laitteen näytöllä, mikä johtaa ongelmiin.

4.6 Appium

Appium on yksi monipuolisimmista testaustyökaluista, kun puhutaan yleisesti kaikista mobiililaitteista. Sen avulla on mahdollista suorittaa testejä kaikkiin kolmeen yleisimpään mobiilikäyttöjärjestelmään (Android, iOS, Windows). Sen monipuolisuutta korostaa myös mahdollisuus käyttää useita eri ohjelmointikieliä testien toteuttamiseen (mm. Java, C++, PHP, Python), joka lisää sen käyttömahdollisuuksia kehittäjien parissa. Joustavuus kielten suhteen johtuu sen pohjautumisesta tietokonesovelluksissa käytettyyn Selenium-testausympäristöön. (Appium-kirjaston dokumentaatio. n.d.)

Vaikka kaikkien laitteiden natiivisovellusten testaaminen onkin työkalulla mahdollista, se ei suinkaan ole sen vahvimpia ominaisuuksia. Sen sijaan niin sanottujen hybridi-sovellusten testaukseen Appium on erittäin arvokas työkalu. Samoja testejä voidaan nimittäin käyttää kaikille edellä mainituille alustoille, ilman suuria muutoksia. Vastaavaa joustavuutta ei muista suosituista työkaluista löydy. Kuviosta 7 voidaan nähdä, kuinka Appiumin testit käytännössä rakennetaan. Tässä Javalla tehdyssä esimerkissä testataan yksinkertaista painikkeen toiminnallisuutta. Painike elementin klikkauksen seurauksena näytölle pitäisi ilmestyä tietty tekstin pätkä, jonka täsmäävyyden testi tarkistaa. Jos oikeaa tekstiä ei löytynyt, testi hylätään.

```
// find an element by tag name
WebElement button = driver.findElement(By.tagName("button"));
button.click();

// get the value of the element
WebElement texts = driver.findElement(By.tagName("staticText"));
assertEquals(texts.getText(), "some expected value");
```

Kuvio 7. Appium testi esimerkki Javalla kirjoitettuna (Appium-kirjaston esimerkki n.d)

4.7 Espresso

Espresso on Android-sovelluksien käyttöliittymän testaukseen tarkoitetuista työkaluista uusin tulokas suosittujen työkalujen joukkoon. Suuren edun muihin työkaluihin verrattuna antaa se, että työkalu on Googlen kehittämä. Käyttöjärjestelmän kehittäjänä Googlella onkin luultavasti eniten motivaatiota tarjota laadukkaita työkaluja kehittäjille. Espresson ensimmäiset versiot ilmestyivät jo vuonna 2014, mutta suurempaan käyttöön se otettiin vasta 2015 vuoden loppupuolella. (Espresso-kirjaston dokumentaatio n.d.)

Espresso on loistava esimerkki työkalusta joka nojaa voimakkaasti lasilaatikkotestaukseen. Sen toiminta on suhteellisen samanlainen kuin Robotiumin, mutta hieman

vielä enemmän kehittäjille suunnattua. Espressoa voisikin kuvailla yhdistelmänä muiden aiempien työkalujen hyviä puolia, samalla huonoja puolia karsien.

Espresson käyttöönotto on tehty hyvin helpoksi kehittäjille, sillä sen asentaminen kehitysympäristöön onnistuu Android Studion kautta automaattisesti. Asentamalla Android Support Repositoryn kehitysympäristöön saa koko Espresso paketin käyttöönsä. Tämä paketti kuten luvussa 4.4 mainittiin sisältää myös UIAutomator-paketin. Espresson toiminnalisuutta on helppo käydä läpi lyhyen esimerkin kautta. (Ks. kuvio 8)

```
onView(withId(R.id.username_field)).perform((typeText(loginInfo.get(0))));
onView(withId(R.id.password_field)).perform(typeText(loginInfo.get(1)));
onView(withId(R.id.login_login_button)).perform(click());
```

Kuvio 8. Esimerkki Espresson koodista sovelluksessa Figi.fm

Kuviosta 8 nähdään kuinka samankaltainen toiminnaltaan Espresso ja Robotium todellisuudessa ovat. Espressossa testiä kirjoittaessa täsmennetään mille elementille tietty toiminto halutaan suorittaa, samoin kuin Robotiumissa. Ainoat suuret erot kyseisten työkalujen välillä ovatkin syntaksi, ja se kuinka laajasti Android-laitteiden elementtejä voidaan löytää työkalua käyttämällä. Espresson suurin etu onkin se, että sama firma tekee sekä käyttöjärjestelmän, että testaustyökalun sille tehtäviin ohjelmiin. Tästä syystä on erittäin harvinaista, ettei jotakin Android-sovelluksen osaa pystytä testaamaan Espressoa käyttämällä. (Espresso cheatsheet. n.d.)

Espresson ominaisuuksilta lisää kerrotaan luvuissa 5.2 ja 5.3 sillä se oli tutkimuksen päätteeksi työkalu, jolla käytännön toteutus tehtiin.

4.8 Jenkins

Jenkins on avoimen lähdekoodin automatisointi palvelin, jota käytetään usein projektien automaattiseen kokoamiseen. Vuonna 2007 alkaneesta Hudson projektista kehittynyt Jenkins on nykyään käytetyn työkalu jatkuvaan integraatioon ohjelmisto projekteissa. (Jenkins wiki n.d)

Sovelluskehittäjille Jenkins on hyvin arvokas työkalu, varsinkin jos projektissa on useita kehittäjiä. Esimerkiksi versionhallinta ohjelmaa Git käyttämällä on mahdollista asettaa Jenkins tarkistamaan tietyin aikavälein lähdekoodiin tulleet muutokset ja koota ohjelma varmistaen, ettei syntaksissa ole virheitä. Samalla varmistetaan, että ohjelman uusin versio on aina helposti saatavilla. Jenkinsiin viimeisimpänä lisätty uudistus, Pipeline vie tämän ominaisuuden vielä pidemmälle. Esimerkiksi Androidin tapauksessa on mahdollista viedä sovellus suoraan kehittäjältä Google Play-kauppaan Jenkinsin kautta, ilman että kehittäjä tekee muuta kuin lähettää lähdekoodinsa versionhallintaan. (Jenkins kotisivut n.d)

Testaajan kannalta Jenkins on myös erittäin hyödyllinen työkalu, sillä sovelluksen automaattisen kokoamisen yhteydessä on mahdollista myös ajaa ennalta määritettyjä testejä. Tällä tavoin voidaan automaattisesti varmistaa, että sovellus täyttää sille annetut kriteerit jokaisen kasauksen yhteydessä. Testien tulokset tulevat automaattisesti Jenkinsin hallintapaneeliin muiden kokoamiseen liittyvien tietojen kanssa.

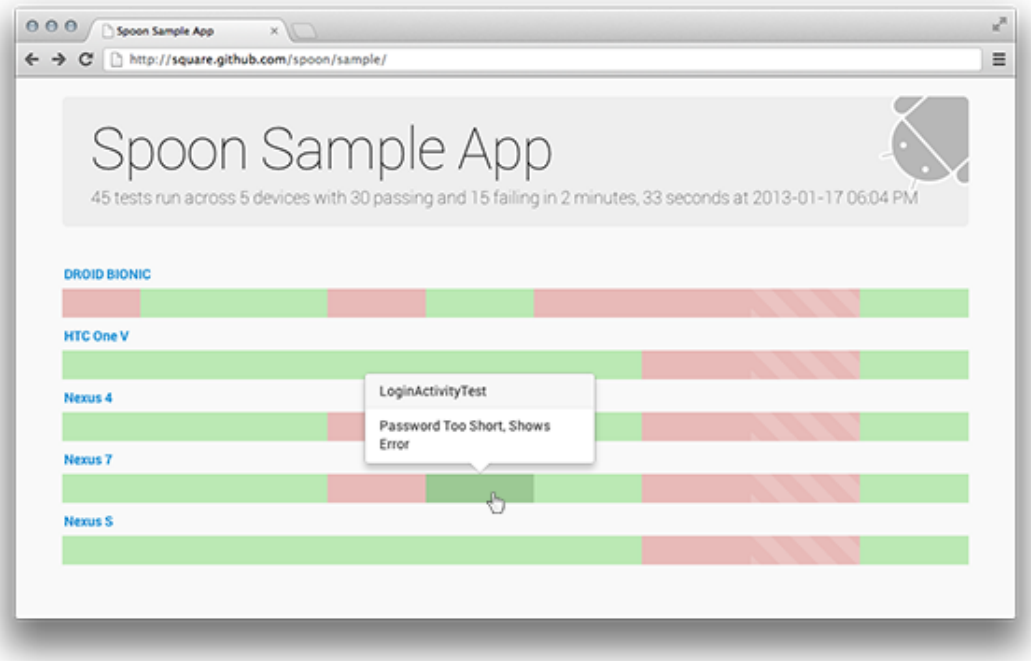
4.9 Spoon framework

Vuonna 2013 Square, Inc:in julkaisema Spoon on yksi harvoista Android-laitteille suunnatuista testauskehyksistä. Luotettu kehittäjä projektin taustalla, sekä avoin lähdekoodi, ovatkin nostaneet sen nopeasti tärkeäksi osaksi Android-testaajien päivittäistä työkalupakkia. Se ei ole oikeastaan testaustyökalu sen kirjaimellisessa merkityksessä, vaan on enemmänkin työkalu joka mahdollistaa testien tulosten tarkemman tutkimisen. Sillä ei ole mahdollista tehdä uusia testejä, vaan sen päämääräinen tarkoitus on ajaa olemassa olevat testit tehokkaammin ja esittää niistä saatua dataa merkitsevällä tavalla. Sitä voidaankin verrata jossain määrin toiminnalisuudeltaan aiemmin mainittuun Jenkinsiin.

Spoonin käyttö on tehty kehittäjän näkökulmasta erittäin yksinkertaiseksi. Se hyödyntää Androidille ominaisia APK tiedostoja, joiden avulla se voi ajaa olemassa olevat testit joko emulaattorilla, tai oikealla laitteella. APK tiedostot kopioidaan samaan kansioon Spoonin kanssa, jonka jälkeen muutaman rivin komentoriville kirjoittamalla

testit suoritetaan, ja niistä muodostetaan automaattisesti HTML-muodossa oleva testausrapotti. (Ks. kuvio 9)

Yksi tärkeimmistä ominaisuuksista jonka Spoon mahdollistaa, on testien ajaminen usealla laitteella samanaikaisesti. Sen ansiosta sovellusta on erittäin helppo vertailla monien laitteiden välillä, ja etsiä löytyykö syy epäonnistuneisiin testeihin laitteesta, vai virheellisestä koodista. Tämä on erityisen tärkeää Android-laitteilla, sillä erilaisia laitteita on tuhansia ja olemassa olevien käyttöjärjestelmien ominaisuudet vaihtelevat suuresti. (Spoon-kirjaston dokumentaatio. n.d.)



Kuvio 9. Esimerkki Spoonin luomasta raportista (Spoon-kirjaston dokumentaatio n.d)

Selkeä vahvuus muihin vastaaviin työkaluihin verrattaessa on Spoonin luoma testausraportti. Yllä olevasta kuviosta nähdään esimerkki siitä, kuinka testien tulokset eri laitteilla näytetään kehittäjälle. Sivulta on sen jälkeen mahdollista tutkia tarkemmin eri laitteilla, tai eri testeissä tapahtuneita virheitä, ja niistä luotuja virheviestejä. Tämän kaltainen raportti on erittäin tehokas kehittäjän kannalta, sillä kaikki tarvittava tieto on helposti saatavilla yhdeltä verkkosivulta.

5. Käytännön toteutus

5.1 Työkalujen valinta

Aiemman luvun aikana käsitellyistä työkaluista testaustiimi valitsi testien toteutusta varten Espresson sekä raportointia varten Spoonin. Valittuihin työkaluihin päädyttiin niiden käyttöönoton helppouden ja erittäin luotettavien kehittäjien vuoksi. Tarkoituksena oli rakentaa testausprosessi jota ei tarvitse muuttaa sovelluksen loppukehityksen aikana työkalujen hajoamisen vuoksi.

5.2 Työkalujen asennus

Espresson asennus Android Studioon

Kuten luvussa 4.6 jo hieman avattiinkin, Espresson asennusprosessi onnistuu kokonaisuudessaan yleisesti käytetyn kehitysympäristön, Android Studion kautta. Kehittäjän tulee lisätä SDK managerin kautta Android Support Repository, jonka jälkeen kaikki testaukseen tarvittava ohjelmointi on mahdollista.

Seuraavaksi täytyy muokata sovelluksen build.gradle tiedostoa hieman, jotta se osaa ottaa kokoamiseen mukaan edellä lisätyt testauskirjaston tiedostot. Samalla määritetään mitä test runneria käytetään, Espresson tapauksessa se on aina yksikkötestauksessa käytetty JUnitRunner.

```
testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
androidTestCompile 'com.android.support.test:runner:0.5'
androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'
```

Kuvio 10. Koodi lisäykset build.gradle tiedostoon

Seuraavana askeleena ennen testauksen aloitusta täytyy vielä luoda uusi konfiguraatio, jotta koottaessa ajetaan testejä, ei varsinaista sovellusta. Tämä onnistuu Android Studiossa seuraavasti: Run menu -> Edit Configurations, josta sen jälkeen voidaan luoda uusi Android Tests konfiguraatio. Tärkeää tässä osassa on huomata lisätä test runneriksi sama JUnitRunner kuin mikä lisättiin build.gradle tiedostoon aikaisemmin.

Näiden vaiheiden jälkeen voidaan aloittaa testien kirjoitus ja ajaminen Android Studion kautta. Testaustiimi havaitsi kuitenkin, että vielä yksi pieni asia saattaa aiheuttaa ongelmia testien suorituksessa, joten sen ratkaiseminen on suositeltavaa. Kyseessä on käytettävän testauslaitteen (puhelin/emulaattori) animaatioiden poistaminen käytöstä. Se onnistuu puhelimen asetuksista kehittäjäasetusten alta, ja poistettavat asetukset ovat:

- Window animation scale
- Transition animation scale
- Animator duration scale

Tällä tavoin poistetaan ongelma, joka johtuu siitä, että testi koittaa mennä eteenpäin, mutta puhelimessa on vielä animaatio käynnissä. Animaatioilla ei ole mitään tekemistä ohjelman lähdekoodin toimivuuden kannalta, joten niiden käyttämättömyys ei tuota uusia ongelmia. Sen sijaan niiden puuttuminen estää ylimääräisten virheellisten testituloksien ilmestymistä, joka on kehittäjän ajankäytön kannalta positiivinen asia. (Espresso-kirjaston dokumentaatio, n.d.)

Spoon

Spoonin käyttöönotto on hieman toisenlaista, sillä se on kokonaisuudessaan komentoriviltä toimiva kokonaisuus, joten varsinaista asennusta ei ole tarpeellista tehdä. Yksinkertaisimmillaan käyttöönotto onnistuu lataamalla heidän verkkosivultaan sekä runner että client .jar tiedostot, ja sen jälkeen voidaankin siirtää APK tiedostot samaan kansioon.

```
java -jar spoon-runner-1.7.1-jar-with-dependencies.jar \
    --apk example-app.apk \
    --test-apk example-tests.apk
```

Kuvio 11. Esimerkki Spoonin ajamisesta komentoriviltä (Spoon-kirjaston dokumentaatio n.d)

Testaustiimi kuitenkin totesi, että tiedostojen siirtäminen manuaalisesti joka kerta muutosten jälkeen on melko kankea tapa suorittaa testausta. Tästä syystä päädyttiin luomaan .bat scripti, jolla tiedostot siirtyvät aina oikeaan paikkaan, ja Spoon suorittaa testit kaikilla käytettävissä oleville emulaattoreilla tai puhelimilla samanaikaisesti.

Samalla olisi mahdollista viedä sovelluksen testauksen automatisointia pidemmälle, ajamalla esimerkiksi Jenkinsillä tämä kyseinen scripti aina kun lähdekoodiin tulisi muutoksia. Projektin pienen kehitystiimin vuoksi todettiin kuitenkin, ettei tällainen toteutus ole tarpeellista, vaan testit voidaan ajaa kehittäjän toimesta ennen kuin lähdekoodi lähetetään versionhallintaan.

5.3 Onnistumiset ja vaikeudet

Projektin lähtökohtana oli luoda hyväksymistestit, jotka kattaisivat mahdollisimman suuren osan sovelluksen toiminnallisuudesta. Testaustiimin harmiksi havaittiin kuitenkin jo suhteellisen varhaisessa vaiheessa, ettei kaikkea testausta ole mahdollista automatisoida. Käyttöjärjestelmien ja laitteiden väliset erot tekevät aukottomasta testauksesta ilman käyttäjiä käytännössä mahdotonta.

5.3.1 Vaikeudet

Kuten kaikissa ohjelmistoprojekteissa, yhteensopivuus ongelmat olivat yksi testaustiimin suurimmista hidasteista toteutuksessa. Tutkitaan esimerkkinä Facebook-loginin tuomia haasteita. Facebook on tullut viime vuosien saatossa erittäin tutuksi monille ohjelmoijille sen tarjoaman helpon käyttäjän tunnistuksen vuoksi. Lisäämällä pienen ulkoisen järjestelmän osaksi omaa sovellusta, ei tarvitse kehittäjän huolehtia käyttäjätiedoista lainkaan sillä Facebook hoitaa sen puolen. Tämä palvelu on kätevä sekä käyttäjän että kehittäjän kannalta, sillä kehittäjän ei tarvitse huolehtia käyttäjätietojen tietoturvasta, ja käyttäjä voi kirjautua samalla profiililla moniin eri sovelluksiin.

Testaajan kannalta tämä kuitenkin on melko ongelmallinen yhtälö, varsinkin Espresson tapauksessa. Ongelman syynä on se, ettei testaaja pääse Espressoa käyttämällä mitenkään käsiksi ohjelman osana toimivaan Facebook-login osioon. Kyseinen toteutus on mitä luultavimmin tarkoituksellista Facebookin puolelta. Jos sen Android-laitteissa hyödyntämään WebView rajapintaan ei pääse testaustyökalulla kiinni, on hyvin todennäköistä, että myös erilaisilla haittaohjelmilla ja muilla vastaavilla on vaikeuksia.

Testaustiimin lukuisten yritysten tuloksena onnistuttiin lopulta luomaan testi, joka toimii vain tietyssä versiossa (SDK 23, tuttavallisemmin Marshmallow) Android-käyttöjärjestelmää. Koska testauksen alla oleva sovellus on tarkoitettu toimimaan huomattavasti tätä vanhemmissa versioissa, testin luotettavuus on melko kyseenalaista. Tästä syystä kyseinen testi jouduttiinkin kommentoimaan pois koodista, ja sitä käytetään vain harvoin.

Toiseksi automatisoidun Android-testauksen heikoksi kohdaksi testaustiimi joutui toteamaan emulaattorit. Niiden välillä havaittiin huimia eroja lähinnä suorituskyvyn puolella. Väärän laitteen emuloiminen saattoi johtaa jopa 10 minuuttia kestäviin testeihin, kun taas hyvässä tapauksessa ne onnistuttiin ajamaan läpi alle kahdessa minuutissa. Testaustiimi päätyikin tästä syystä käyttämään jokaisesta SDK versiosta vain yhtä laitetta, sillä kaikkien vaihtoehtojen tutkiminen olisi ollut liian aikaa vievää puuhaa. Kun hyvä laite löydettiin, siinä myös pysyttiin.

Viimeisimpänä suurena hankaluutena oli se, että osa ohjelman toiminnalisuudesta vaatii Google-tilin. Esimerkiksi Figi.fm:stä löytyvä tapahtumien kalenteriin lisääminen osoittautui hyvin hankalaksi testattavaksi emulaattoreita käyttämällä, sillä niihin Google-tilin lisääminen on työlästä. Samaa tiliä ei voi käyttää useassa laitteessa samanaikaisesti, joten testaustiimi joutui toteamaan, ettei se ole vaivan arvoista. Ongelma kierrettiin testaamalla kalenteriin lisäys metodi ohjelmassa ilman että katsotaan mitä kalenteriin tulee. Jos sovellus ei testin aikana kaadu, voidaan kohtalaisella varmuudella olettaa, että kalenteriinkin tietoa lisääntyi. Tiimi totesi, ettei pienen osan testaaminen ole tärkeämpää kuin kokonaisuuden testaaminen useilla laitteilla samanaikaisesti.

5.3.2 Onnistumiset

Edellä mainituista vastoinkäymisistä huolimatta testaustiimi onnistui rakentamaan testausprosessin, jolla saadaan varmistettua, että valtaosa ohjelmasta toimii siltä odotetulla tavalla. Tiimi oli erityisen tyytyväinen siihen, kuinka hyvin Espressoon liitetty UIAutomator mahdollisti joitakin varsinaisen ohjelman ulkopuolelle ulottuvia testejä. Esimerkkinä tällaisesta toiminnalisuudesta oli salasanan vaihtamisen testaus.

Tässä tapauksessa käyttäjä vaihtaa salasanansa ja kirjautuu ulos ohjelmasta. Kirjauduttaessa ulos, ohjelma sammuu ja siihen ei pääse käsiksi ilman UIAutomatoria. Sen avulla sovellus oli mahdollista käynnistää uudestaan valikosta ja kokeilla onnistuiko käyttäjän salasanan vaihto.

Samalla työkalulla rakennettiin myös ainut toimiva versio aiemmin mainitusta Facebook-login testauksesta. Espresson perustoiminnoilla siihen on erittäin hankala päästä käsiksi, mutta UIAutomatorin kautta voidaan tehdä toimintoja WebView-elementeille.

Vaikka edellisen luvun lukemisesta saattaakin jäädä hieman negatiivinen kuva Espresson käytöstä, se ei kuitenkaan vastaa tiimille jäänyttä kokonaiskuvaa. Vaikeuksista huolimatta kaikki Android-sovelluksille ominaiset piirteet saatiin testattua erittäin onnistuneesti ja työkalu osoittautui erittäin käyttäjäystävälliseksi kehittäjän näkökulmasta. Kaikista onnistumisista on haastava keksiä hyviä esimerkkejä, lähinnä sen vuoksi että onnistuneita asioita on niin monta.

5.4 Espresso testi esimerkki

```
@Test
public void ForgotPwTest() {

    onView(withId(R.id.forgot_pw)).perform(click());
    //Check that we are on right page
    onView(withId(R.id.reset_pw_text)).check(matches(withText("Fill your e-
mail address below")));
    //Lets do a failed one and see the error handling works

onView(withId(R.id.reset_pw_email)).perform(typeText("eitaatoimi@asdasdasd.
com"));
    closeSoftKeyboard();
    onView(withId(R.id.reset_pw_send)).perform(click());
    //We assume error dialog popped, lets click ok and do a correct email
one
    onView(withText("OK")).perform(click());

onView(withId(R.id.reset_pw_email)).perform(clearText(),typeText("mika.leht
inen92@gmail.com"));
    closeSoftKeyboard();
    onView(withId(R.id.reset_pw_send)).perform(click());
    try {
        //Try to catch the toast message, in some emulators this fails so
we check that we returned to launch activity in the catch part

onView(withText(R.string.reset_pw_email_sent)).inRoot(withDecorView(not(mAc
tivityRule.getActivity().getWindow().getDecorView()))).check(matches(isDisp
layed()));
    } catch (NoMatchingRootException a) {

onView(withId(R.id.login_intro_text)).check(matches(isDisplayed()));
    }
}
```

Yllä nähdään esimerkki kokonaisesta testistä, joka on toteutettu pelkästään Espressoa käyttämällä. Seuraavaksi käydään vaiheittain läpi mitä testi tekee, ja miten se käytännössä tapahtuu.

- Jokainen testi alkaa ohjelman ”etusivulta”, kyseisessä sovelluksessa se on sisäänkirjautumiseen tarkoitettu sivu. Tämän testin tarkoitus on testata käyttäjän salasanan resetointia.
- onView(withId) on yksi espresson yleisimmistä tavoista navigoida elementtien välillä. Ensimmäinen (R.id.forgot_pw) johdattaa klikkauksen kautta meidät salasanan resetointi näkymään, kun taas seuraavaa (R.id:reset_pw_text) käytetään tarkistamaan, että onko klikkauksen seurauksena päästy oikealla sivulle. Tarkistus tapahtuu vertaamalla

näkymässä olevaa tekstiä kehittäjien tietämään oikeaan tekstin pätkään.

- Seuraavaksi suoritetaan virheviestin tarkistus syöttämällä sähköposti osoite jollaista ei löydy tietokannasta, käyttämällä typeText() funktiota.
- onView(withText(”OK”)) viittaa tämän testin puitteissa ennalta määriteltyyn virheviestiin, jonka kuuluu näkyä virheellisen sähköpostin tapauksessa.
- Seuraavana syötetään oikea sähköposti osoite ja testataan, lähteekö ohjelman toimesta viesti käyttäjälle
- Tarkistus perustuu tässä tapauksessa pitkälti taustalla toimivaan REST-APIin, sillä jos sähköpostin lähetys ei onnistu, sovellukseen tulee virheilmoitus. Testin viimeisenä tarkistuksena on se, että pääsikö käyttäjä takaisin sisäänkirjautumiseen.
- Jos käyttäjä pääsi sisäänkirjautumiseen asti takaisin, testi on onnistunut.
- Jos käyttäjä ei pääse takaisin sisäänkirjautumiseen, tai jos näytöllä on sisäänkirjautumis-sivulla virheviesti, testi on epäonnistunut.

6. Pohdinta

6.1 Testausympäristöjen arviointi

Työkalujen soveltuvuus projektiin on luonnollisesti paljolti kiinni siitä, minkä suuruinen projekti on kyseessä. Vaikka projekti olisi vain muutaman henkilön sivutyö, sovelluksen testaus on silti syytä vähintään jollain tasolla toteuttaa. Kuten jo luvussa 4.2 mainittiin, Monkey on vähiten kehittäjän resursseja vaativa testaustyökalu. En kuitenkaan henkilökohtaisesti pidä sillä suoritettavia testejä riittävänä, sillä sen tulokset eivät ole ennalta arvattavia. Tästä päästääkin käsittelemään testausympäristöjä joilla hyväksyntätestauksen toteuttaminen on mahdollista sekä suurissa että jopa yhden hengen kehitysprojekteissa.

Robotium

Hyväksymistestauksen näkökulmasta Robotium on vähintäänkin varteenotettava vaihtoehto. Sen kehitysympäristöön lisääminen on erittäin vaivatonta, sillä pelkän jar-kirjaston lisääminen Android kehitysympäristöön on kehittäjille jo ennestään tuttua puuhaa. Helpon käyttöönoton luoma kätevyys madaltaa huomattavasti kynnystä testauksen aloittamiseen.

Hyviin puoliin voidaan laskea myös Robotiumin suhteellisen pitkä ikäisyys, sen 6.0.1 versio esimerkiksi julkaistiin yli kolme vuotta sitten tammikuussa 2014. Nykyään mennään versiossa 5.63 joten merkkejä hidastumisesta ei ainakaan vielä ole huomattavissa.

Appium

Appium on varsin samaan tapaan kuin Robotium erittäin monipuolinen testaustyökalu, jonka kanssa on mahdollista testata Android-sovellusten lisäksi iOS- ja Windows-käyttöjärjestelmien sovelluksia. Tästä syystä pidänkin sitä erittäin tärkeänä ympäristönä opeteltavaksi, jos työskentelee esimerkiksi firmassa joka kehittää molemmilla iOS- ja Android-alustoilla toimivia sovelluksia. Pelkkiin Android-sovelluksiin en kuitenkaan Appiumia näe hyvänä valintana, yksinkertaisesti koska parempiakin vaihtoehtoja on tarjolla.

Espresso

Espresso on omasta mielestäni testauksen aloittamiseen kaikista varteenotettavin työkalu. Samaan tapaan kuin Robotium, Espresson kehitysympäristöön lisääminen on erittäin helppoa. Android Studiossa SDK-managerin kautta asennettava valmis paketti mahdollistaa testauksen aloituksen.

Espresson selkeästi parhaaksi puoleksi lasken kuitenkin sen monipuolisuuden. Android-sovelluksissa on erittäin paljon erilaisia käyttöliittymä elementtejä joiden testaaminen voi osoittautua joillekin työkaluille olla haastavaa. Googlen kehittämälle testausympäristölle tämä ei luonnollisesti ole ongelma, koska kyllähän käyttöjärjestelmän kehittäjä tietää mitä sillä voidaan tehdä.

Toiseksi hyväksi puoleksi on pakko mainita UIAutomator. Sen mahdollistama sovelluksen ulkopuolella navigointi tuo testaukseen uutta syvyyttä, jota en muiden työkalujen toimesta nähnyt mahdolliseksi. Samalla UIAutomator mahdollistaa myös joidenkin ulkoisten kirjastojen lisäämien elementtien testauksen, joihin perinteisesti tarvitsisi lisätä ylimääräisiä kirjastoja muita työkaluja käyttäessä.

Jenkins

Testien automatisointi Jenkinsiä käyttämällä on erittäin suosittu käytäntö, varsinkin web-sivujen testauksen parissa. Valitettavasti ainakaan tämän tyylisen Android-projektin puitteissa Jenkins jätti paljon toivomisen varaa. Android-sovellusten testaus luottaa edelleen erittäin paljon emulaattoreihin, ja niiden yhdistäminen automatisoituun palvelimeen kuten Jenkins, ei tuntunut toimivan kovinkaan saumattomasti. Virtuaaliympäristön luominen automatisoidulle palvelimelle tuntui aiheuttavan enemmän ongelmia kuin siitä oli hyötyä. Epävakaana ympäristönä se ei luonnollisesti soveltunut tällaisen sovelluksen testaukseen.

Tutkimuksen aikana kuulin kuitenkin muilta sovelluskehittäjiltä paljon positiivisia puolia Jenkinsin automatisoinnista. Sen toiminnallisuus tuntuu toimivan hyvin, kun puhutaan pienemmistä sovelluksista joissa toimintoja ei ole kovinkaan paljoa. Äskettäin lisätty Pipeline ominaisuus, jolla sovelluksia voidaan automaattisesti päivittää Jenkinsin kautta suoraan Google Play- kauppaan, on taatusti arvokas työkalu kehittäjille jotka haluavat nopeuttaa sovelluksen päätymistä kehittäjältä käyttäjälle.

6.2 Jatkokehitys

Opinnäytetyön tavoitteena oli kartoittaa mahdollisia tapoja toteuttaa automatisoitu hyväksymistestaus sovellukselle, ja lopulta käytännössä toteuttaa kyseinen järjestelmä. Tavoitteeseen päästiinkin omasta mielestäni mallikkaasti, vaikkakin eri työkalujen tutkimiseen menikin enemmän aikaa kuin odotin. Samoin varsinaiseen käytännön toteutukseen kulunut aika pääsi yllättämään. Lopputuloksena oli kuitenkin järjestelmä, jolla pystytään varmistamaan, ettei käyttäjälle päädy sovellusta jossa on suuria ongelmia. Luonnollisena jatkokehityksen kohteena on tietysti parantaa kehitetyn testauksen kattavuutta.

Toisena jatkokehityksen kohteena olisi automatisoinnin pidemmälle vieminen esimerkiksi jo paljon puhutun Jenkinsin avulla. Vaikka se jouduttiinkin projektin aikataulullisten syiden vuoksi hylkäämään, sen integroiminen järjestelmään myöhemmässä ajankohdassa olisi erittäin mielenkiintoinen tutkimuskohde. Automatisoinnin pidemmälle vieminen on viime vuosina ollut jatkuvassa kasvussa yritystoiminnankin puolelta, joten sen tutkiminen olisi melko ajankohtainen aihe.

Viimeisimpänä kehityskohteena on suhteellisen luonnollisena uusien testausympäristöjen tutkiminen. Varsinkin Android-sovelluksille tuntuu uusia työkaluja tulevan varsin mukavaa tahtia ja niiden vertailu voisi olla arvokas tutkimuskohde tulevaisuutta ajatellen.

6.3 Tutkimus kokonaisuutena

Opinnäytetyön aihe oli itselleni varsin mielenkiintoinen ja ajoittain haastava. Android-sovellukset ovat olleet opintojen loppupuolella suuressa osassa erilaisten kouluprojektien ja harjoittelun kautta, mutta niiden testaus ei ole ollut missään vaiheessa suuressa osassa. Testaus ylipäätään on opintojen aikana jäänyt vähäiseen osaan, joten tällaisen projektin toteutus oli mukavaa vaihtelua.

Koko opinnäytetyön teko kesti lähes vuoden verran, hyvin vaihtelevalla aktiivisuudella. Tässä ajassa työkalujen kehityksen nopeus tuli erittäin hyvin tutuksi. Jos jokin työkaluista vaikutti projektia aloittaessa melko pelkistetyltä, eikä sillä pystynyt oikein

mitään tekemään, yhtäkkiä se saattoikin olla yksi paremmista työkaluista. Se avasi silmät hyvin sille, kuinka sekä avoimen lähdekoodin että suurien yritysten tuottamat suljetun lähdekoodin työkalut saattavat kehittyä valtavasti lyhyessä ajassa.

Tutkimuksen tulokset eivät lopulta olleet niin selkeitä kuin olisin itse toivonut. Tutkitut testaustyökalut ovat sen verran erilaisia, ettei niiden toimintaa voi suoraan edes vertailla. Lisäksi opinnäytetyön suhteellisen lyhyt aika-ikkuna ei antanut mahdollisuutta lähteä toteuttamaan testausta monella työkalulla, joten niiden vertailu perustuu pitkälti muiden käyttäjien kokemuksiin. On kuitenkin selkeää, että testaukseen ei ole sitä yhtä oikeaa työkalua, vaan niitä on syytä vertailla projektin vaatimukset mielessä pitäen.

Lähteet

Android developer manual, activity. Viitattu 2.4.2017. https://developer.android.com/reference/android/app/Activity.html

Android developer manual, Monkey. Viitattu 2.4.2017. https://developer.android.com/studio/test/monkey.html

Android developer manual, UIAutomator. Viitattu 2.4.2017. https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html

Appium-kirjaston dokumentaatio. Viitattu 2.4.2017. https://github.com/appium/appium

Appium-kirjaston esimerkki. Viitattu 2.4.2017 https://github.com/appium/sample-code/blob/master/sample-code/examples/java/junit/src/test/java/com/saucelabs/appium/SauceTest.java

Black-box testing wiki. Viitattu 2.4.2017. https://en.wikipedia.org/wiki/Black-box

Espresso cheatsheet. Viitattu 2.4.2017. https://google.github.io/android-testing-support-library/docs/espresso/cheatsheet/

Espresso-kirjaston dokumentaatio. Viitattu 2.4.2017. https://github.com/codepath/android_guides/wiki/UI-Testing-with-Espresso

Gray-box testing wiki. Viitattu 2.4.2017. https://en.wikipedia.org/wiki/Gray_box

Integration testing wiki. Viitattu 2.4.2017. https://en.wikipedia.org/wiki/Integration_testing

IDC markkinatutkimus. Viitattu 2.4.2017. http://www.idc.com/promo/smartphone-market-share/os

IT Audit Academy. Viitattu 2.4.2017. http://www.itauditacademy.com/black-box-testing

Jenkins wiki. Viitattu 2.4.2017. https://en.wikipedia.org/wiki/Jenkins_(software)

Jenkins kotisivut. Viitattu 2.4.2017. https://jenkins.io/

Jenkinsin Android-plugin dokumentaatio. Viitattu 2.4.2017. https://wiki.jenkins-ci.org/display/JENKINS/Android+Emulator+Plugin

Mikko Mäki-Rahkola Testaussuunnitelma. 2003. Viitattu 2.4.2017. http://www.soberit.hut.fi/T-76.115/03-04/palautukset/groups/PPT/i1/testing/testplan.htm

Robotium-kirjaston dokumentaatio. Viitattu 2.4.2017.

https://github.com/robotiumtech/robotium/wiki

Spoon-kirjaston dokumentaatio. Viitattu 2.4.2017.

https://github.com/square/spoon

Software testing wiki. Viitattu 2.4.2017. https://en.wikipedia.org/wiki/Software_testing

Unit testing wiki. Viitattu 2.4.2017. https://en.wikipedia.org/wiki/Unit_testing

White-box testing wiki. Viitattu 2.4.2017. https://en.wikipedia.org/wiki/White-box_testing