Joose Virta

ANDROID-SOVELLUKSEN TESTAUKSEN AUTOMATISOINTI

Tietojenkäsittelyn koulutusohjelma
2017

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

ANDROID-SOVELLUKSEN TESTAUKSEN AUTOMATISOINTI

Virta, Joose
Satakunnan ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Maaliskuu 2017
Ohjaaja: Nieminen, Hans
Sivumäärä: 37

Asiasanat: Android, Java, testaus, testauksen automatisointi

Opinnäytetyön tavoitteena oli pyrkiä helpottamaan ja nopeuttamaan testausta ja testien tekemistä Android-kehitysympäristössä. Käytännön osuus toteutettiin MyGamez Finland Oy:lle tämän kehittämään MySDK-projektiin ja julkaistaviin peleihin, joihin MySDK on integroitu.

Sovelluksen testaus manuaalisesti on helposti hyvin aikaa vievä, vaivalloinen ja virhealtis prosessi. Automatisoimalla tätä prosessia voidaan saada aikaan merkittäviä ajansäästöjä sekä vähentää inhimillisten virheiden määrää, joka vuorostaan voi vähentää ajankäyttöä vielä enemmän vähentämällä virheidenetsimistä ja korjaamista kehitysprosessin myöhemmässä vaiheessa.

Opinnäytetyön käytännön osuudessa mahdollistetaan yksikkötestien tekeminen MySDK-projektissa sekä kehitetään testiympäristö julkaistavien pelien käyttöliittymätestien luomista ja suorittamista varten.

ANDROID APPLICATION TESTING AUTOMATION

Virta, Joose
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Business Information Technology
March 2017
Supervisor: Nieminen, Hans
Number of pages: 37

Keywords: Android, Java, testing, test automation

The purpose of this thesis was to make creating and running unit and UI tests easier and faster in an Android development environment. The practical part of this thesis was done for MyGamez Finland Oy and implemented in their MySDK project and games to be published that have integrated MySDK.

Testing an application manually can easily be a very time consuming task, and having to do it repeatedly leaves plenty of room for human error. By automating this task, time spent can be reduced greatly, and reducing the possibility for errors, which in turn can immensely reduce the time spent finding and fixing potential bugs later in the development lifecycle.

In the practical part of this thesis, the MySDK project will be configured to make creating and running unit tests possible, and for games to be published, a testing environment developed for creating and running automated UI tests.

SISÄLLYS

1 JOHDANTO 5
2 TILANNE 6
3 TESTAUS 7
3.1 Testauksesta yleisesti 7
3.2 Testaus ohjelmistokehityksessä 8
3.3 Testaustavat 10
3.3.1 Staattinen ja dynaaminen testaus 10
3.3.2 Musta laatikko - ja lasilaatikkotestaus 10
3.3.3 Regressiotestaus 11
3.4 Testaustasot 11
3.4.1 Yksikkötestaus 11
3.4.2 Integrointitestaus 12
3.4.3 Järjestelmätestaus 12
3.4.4 Hyväksymistestaus 12
4 TESTAUKSEN AUTOMATISOINTI 13
4.1 Automatisoinnista yleisesti 13
4.2 Automatisoinnin hyödyt ja ongelmat 14
4.3 Mitä kannattaa automatisoida? 15
4.4 Yksikkötestien automatisointi 16
4.5 Käyttöliittymätestauksen automatisointi 17
5 KÄYTETTÄVÄT TEKNIIKAT 18
5.1 Android 18
5.2 Android Studio 18
5.3 JUnit 19
5.4 Mockito 19
5.5 Docker 20
5.6 Node.js 20
5.7 Appium 20
5.8 Selenium Grid 21
6 MYSDK-PROJEKTIN TESTIYMPÄRISTÖ 21
6.1 Lähtökohdat 21
6.2 Yksikkötestauksen käyttöönotto Android Studio -ympäristössä 22
7 JULKAISTAVIEN PELIEN TESTIYMPÄRISTÖ 25
7.1 Paikallisen testipalvelimen toteutus 25
7.2 Käyttöliittymätestiympäristön konfigurointi 30
8 YHTEENVETO 34
LÄHTEET 36

1 JOHDANTO

Opinnäytetyön tavoitteena oli tutustua testauksen automatisointiin Android-sovelluskehityksessä ja sen avulla tehdä testien tekemisestä sekä suorittamisesta helpompaa ja vähemmän aikaa vievää. Työn käytännön osuus toteutettiin MyGamez Finland Oy:lle. Työn teoriaosuudessa selitetään aluksi testaus yleisesti, jonka jälkeen käydään läpi testauksen automatisointi, sen hyötyjä ja haittoja ja miten se voidaan toteuttaa. Käyn läpi käytännön osuudessa käytetyt työkalut ja teknologiat ja itse käytännön osuudessa otan niitä käyttöön MySDK-projektissa sekä julkaistavien pelien testauksessa.

MyGamez on suomalais-kiinalainen mobiilipelijulkaisija, jonka tavoitteena on antaa länsimaalaisille pelinkehittäjille helppo ja yksinkertainen keino päästä Kiinan mobiilipelimarkkinoille. Kiinan mobiilipelimarkkinoille voivat julkaista pelejä ainoastaan kiinalaiset julkaisijat. Länsimaalaisella julkaisijalla on joko oltava Kiinan puolella yhteistyökumppani tai oma julkaisuyritys, jolla on hallussaan pelien julkaisemiseen vaadittava lisenssi. MyGamez tarjoaa myös lokalisointiapua pelin sovittamiseen kiinalaisille markkinoille.

Kiinassa on kymmenittäin kanavia joiden, pelikauppojen kautta käyttäjät voivat ladata pelejä puhelimilleen. Monilla kanavilla on omat vaatimuksensa, jotka pelin tulee täyttää, jotta se voidaan hyväksyä ja päästää kanavan testeistä läpi. Jotkin kanavat voivat esimerkiksi vaatia oman logonsa pelin kuvakkeeseen tai käynnistysruutuun. Jotkin vaatimukset voivat olla teknisempiä, esimerkiksi pelisovelluksella ei saa olla lupaa lähettää SMS-viestejä tai sen sovellustunnisteen on oltava tietynmallinen. Jotkin suosituimmat kanavat kehittävät omaa SDK-pakettiaan, joka täytyy integroida peliin, jotta peli voidaan hyväksyä kanavan pelikauppaan. Laajaa kanavakattavuutta havittelevalle pelinkehittäjälle useiden SDK-pakettien ylläpitäminen voi olla hyvinkin työlästä, sillä ne päivittyvät suhteellisen useasti ja niiden päivittäminen on hyvin usein pakollista. Lisäksi mitä suositumpi ja suurempi kanava on, sitä laajempi ja monimutkaisempi sen tarjoama SDK-paketti usein on.

MySDK on MyGamezin kehittämä ja tarjoama SDK-paketti, joka toimii rajapintana pelin ja eri maksujärjestelmien ja kanavien omien SDK-pakettien välillä. Tämän ansiosta pelinkehittäjien ei tarvitse taistella kymmenien eri SDK-pakettien kanssa, vaan riittää että pelissä on integroituna ainoastaan MySDK, joka yhdistää ne yhden rajapinnan taakse.

2 TILANNE

Opinnäytetyön aloitushetkellä kaikki testaus tapahtui manuaalisesti. Kunnollisia yksikkötestejä ei ollut, ja käyttöliittymätestit tehtiin käsin. Ajan ja tiedon puutteen vuoksi yrityksessä ei ollut ehditty kunnolla tutustua testaukseen eikä testauksen automatisointiin, vaikka eri menetelmien ja työkalujen käyttöönotto oli ollut suunnitteilla jo jonkin aikaa. Koska testattavia pelejä ja eri kanavaversioita on monta, alkaa helposti testaamisen into ja tarkkuus hiipua ennen pitkää. Kaikkia ei edes välttämättä ehdittäisi testata kunnolla, jos julkaisulla on kiire.

Kun halutaan julkaista uusia versioita kanaville, Kiinan toimistolta lähetetään dokumentti, jossa on määritelty kanavat mille halutaan julkaista, käytettävät versionumerot ja kanavakohtaisia tietoja ja vaatimuksia. Tämän dokumentin pohjalta luodaan konfiguraatiot, joilla generoidaan kanavakohtaiset versiot eri peleistä.

Kun kanavakohtaiset versiot on generoitu, ne käydään käsin läpi ja varmistetaan niiden yleinen toimivuus, jonka jälkeen ne lähetetään Kiinan puolelle. Maksutoimintoja ei voida Suomen puolella testata, joten se jää Kiinan toimiston testausryhmän hoidettavaksi. Kiinan testaajat käyvät parhaansa mukaan vielä pelit uudelleen läpi, testaavat maksujen toimivuuden ja varmistavat että ne vastaavat kanavien asettamia vaatimuksia. Operaattoreilla ja kanavilla yleensä on myös omat testausprosessinsa, joiden läpi pelit ajetaan ennen julkaisemista.

Yhden version testaus ei vielä välttämättä ole raskas työ, mutta kun julkaistavia versioita on kuutisenkymmentä kappaletta ja niitä pitäisi testata usealla puhelimella, niitä

kaikkia ei ole mahdollista järkevässä ajassa hutiloimatta käydä läpi. Silloin tällöin joitakin asioita jää huomaamatta. Joskus mahdolliset virheet jäävät vasta kanavan testiprosessin haaviin, jolloin se hylätään ja lähetetään takaisin. Tahojen välinen kommunikointi voi joskus olla hyvinkin hidasta, joten hukkaan mennyt aika voi olla tässä tapauksessa hyvinkin merkittävä. Siksi olisi parasta, että virheet saataisiin kiinni mahdollisimman varhaisessa vaiheessa. On siis korkea aika ottaa testaustyökaluja käyttöön.

3 TESTAUS

3.1 Testauksesta yleisesti

Testaaminen voidaan määritellä olevan sovelluksen tai sen osien suorittamista virheiden löytämiseksi mahdollisimman aikaisessa vaiheessa ohjelmistokehitystä. (Haikala & Mikkonen 2011, 205; Myers, Sandler & Badgett 2011, 6) Testaamisella pyritään myös varmistamaan, että toteutettu ohjelma on suunnitelmien ja määrittelyjen mukainen, toimii kuten sen pitääkin ja se täyttää asiakkaan tarpeet. (Kasurinen 2013, 10, 13)

Koodia voidaan oikeastaan testata jo ennen kuin se on kertaakaan suoritettu, sillä jo koodia ja sen määrittelyä lukemalla voidaan analysoida koodin toiminta ja todeta sen toimivuus ja laatu. Testauksen ei edes tarvitse tapahtua ihmisen toimesta, sillä jo käytetty editori voi aktiivisesti tarkistaa koodin syntaksia mahdollisten virheiden varalta.

Testaaminen ei kuitenkaan takaa ohjelmiston virheettömyyttä. Sillä voidaan todistaa, että virheitä löytyy, mutta se ei voi yksinkertaisessakaan tapauksessa todistaa, että niitä ei olisi. Ohjelmasta on käytännössä mahdollista testata vain hyvin pieni osa kaikesta toiminnallisuudesta, joten kaikkia mahdollisia virheitä ei edes voida yrittää löytää. (Haikala & Mikkonen 2011, 205)

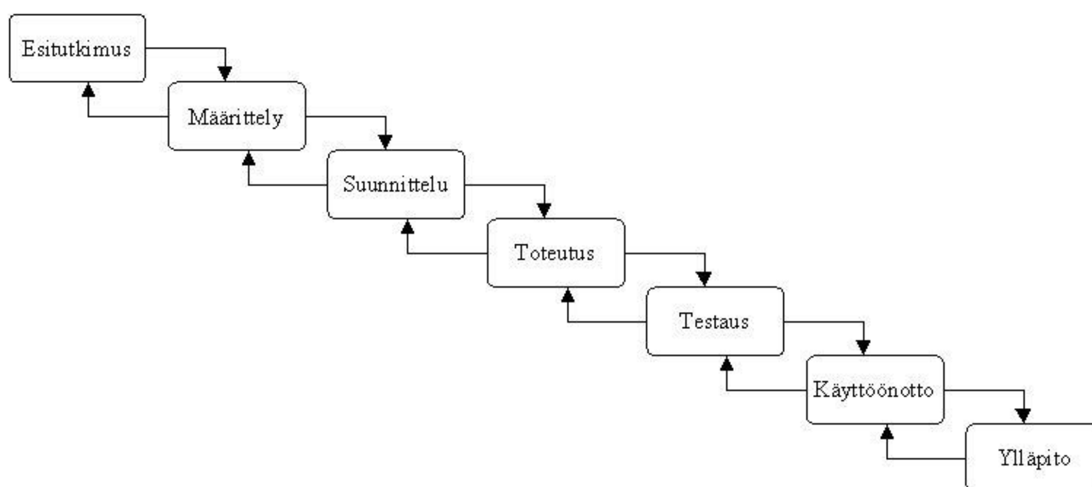
Testaamisen työvaiheita ovat testauksen suunnittelu, testiympäristön luonti, testin suoritus ja sen tuloksen tarkastaminen. Suunnitteluvaiheessa määritellään testisuunni-

telma ja käytettävät testikäytännöt ja luodaan testitapaukset vaatimusmäärittelyn pohjalta. Testauksen mahdollistamiseksi kehitetään testausympäristöä, jossa testit voidaan suorittaa. Testit suoritetaan testausympäristössä, ja niiden tuloksia verrataan odotettuihin tuloksiin. (Haikala & Mikkonen 2011, 205)

3.2 Testaus ohjelmistokehityksessä

Testaus on hyvin olennainen osa ohjelmistokehitystä. Se on ohjelmoinnin ohella vaihe, johon ohjelmistoprojektissa voi kulua hyvinkin paljon aikaa ja resursseja. Sitä kuitenkin voidaan painottaa hyvin eri tavalla riippuen siitä, millainen projektin prosessi on.

Yksi tunnetuimmista prosessimalleista on vesiputousmalli (kuva 1). Vesiputousmallissa edetään ohjelmistoprojektissa vaihe kerrallaan. Alun perin taaksepäin iterointi oli sallittua, ja jopa suotavaa, mutta jossain vaiheessa sitä alettiin välttää. Äärimmäisissä tapauksissa se jopa kielletään täysin. (Haikala & Mikkonen 2011, 37)

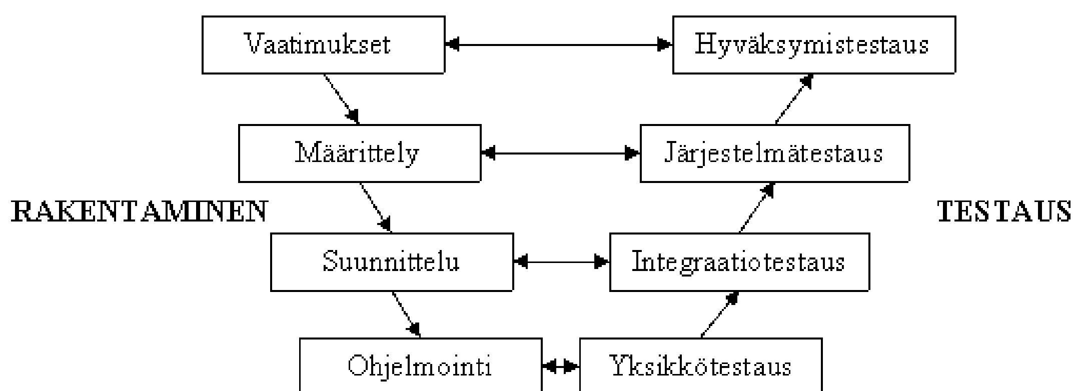


Kuva 1. Vesiputousmalli

Vesiputousmallissa testaaminen tapahtuu vasta ohjelman toteutusvaiheen jälkeen. Mikäli törmätään virheeseen joka olisi helposti voitu korjata jo määrittely- tai suunnitteluvaiheessa, korjaus tapahtuu vasta kun koko toteutus on tehty ja tämä voi johtaa merkittävään määrään korjauksia senhetkiseen toteutukseen. (Patton 2006, 34)

Vesiputousmallissa on tarkoitus, että jokainen vaihe suoritetaan kunnolla loppuun ennen seuraavaan siirtymistä. Testausvaiheessa ilmentyviä virheitä ei siis pitäisi olla, koska kaikki olisi niin tarkasti ja hyvin määritelty, että ne olisi jo huomattu määrittely- ja suunnitteluvaiheissa. On silti mahdollista, että jokin merkittävä virhe saattaa päästä läpi, ja kun se huomataan vasta testausvaiheessa, sen korjaaminen voi olla hyvinkin kallista, kun koko ohjelma on jo toteutettu. (Patton 2006, 34)

Toinen tunnettu malli on V-malli (kuva 2), jossa testaus on nostettu paremmin esille. V-mallissa eri kehityksen elämänkaaren vaiheille on niitä vastaava testaustaso. Tällöin testausta tapahtuu pitkin ohjelmistokehityksen elämänkaarta, eikä vain projektin loppupuolella. (Kasurinen 2013, 51)



Kuva 2. Testauksen V-malli

Edellä mainituissa testaus tapahtuu aina toiminnallisuuden toteutuksen jälkeen. Testivetoisessa ohjelmoinnissa (*test driven development*) tämä on kuitenkin käännetty päälaelleen. Se on testien ympärille keskittynyt toimintamalli, jossa testit luodaan ennen kuin toiminnallisuutta on tehty. Toiminnallisuus määrittyy testien kautta. Koodia tehdään niin kauan, kunnes se toteuttaa vaaditun toiminnallisuuden ja pääsee testeistä läpi. Tämän jälkeen voidaan siirtyä seuraavan toiminnallisuuden toteuttamiseen. (Kasurinen 2013, 148)

3.3 Testaustavat

3.3.1 Staattinen ja dynaaminen testaus

Staattisessa testauksessa ohjelmaa testataan ilman, että sitä suoritetaan. Tämä tapahtuu analysoimalla ja arvioimalla järjestelmän koodia, määrittelydokumentteja ja suunnitelmia. Staattinen testaus voidaan aloittaa hyvin aikaisessa vaiheessa ohjelmistokehitystä, jo ennen kuin mitään koodia on saatu aikaan. Ohjelmakoodin staattiseen testaukseen voidaan lukea esimerkiksi ohjelmointiympäristöissä yleisenä ominaisuutena oleva syntaksin tarkistaja. Myös kääntäjä voi toimia staattisena testaajana huomaten mahdollisia virheitä käännösvaiheen aikana. (Kasurinen 2013, 65)

Dynaaminen testaaminen taas on päinvastaisesti järjestelmän testaamista ja tarkastelua sen suorituksen aikana. Dynaamisessa testauksessa ohjelma tai sen osia suoritetaan mahdollisesti erilaisilla syötteillä ja tarkistetaan, toimiiko se odotetusti ja onko sen tuottama tulos oikea. (Kasurinen 2013, 65)

3.3.2 Musta laatikko - ja lasilaatikkotestaus

Musta laatikko -testauksella *(black box testing)* tarkoitetaan sovelluksen testaamista tietäen miten sovelluksen tulisi määrittelyjen ja dokumentaation mukaan toimia, mutta ilman tietoa siitä, miten ohjelma on varsinaisesti toteutettu. Dynaaminen testaus tapahtuu siis valmiilla suoritettavalla sovelluksella. Sovellusta testataan useilla eri syötteillä ja varmistetaan, että toiminta ja tulokset ovat oikeanlaisia. Sovelluksen käyttöliittymä voidaan testata joko manuaalisesti naputellen tai hyödyntäen jotakin testaustyökalua, joka tekee sen automaattisesti. Musta laatikko -testaus voidaan tehdä myös staattisesti esimerkiksi tutkimalla ohjelman käyttödokumenttia. Se kuvaa, miten ohjelman pitäisi toimia, mutta ei paljasta ohjelman sisäisestä toiminnasta mitään. (Kasurinen 2013, 65-66; Patton 2006, 56, 64)

Lasilaatikkotestauksessa *(white box testing)* hyödynnetään tietoa siitä, miten ohjelma on toteutettu. Ohjelman suoritusta seurataan ikään kuin läpinäkyvän lasilaatikon läpi,

jolloin nähdään, miten ohjelman osat ja palaset toimivat keskenään. Staattisessa lasilaatikkotestauksessa käydään läpi järjestelmän rakennetta, arkkitehtuuria sekä koodia ja etsitään niistä mahdollisia virheitä. Dynaamisessa lasilaatikkotestauksessa tarkastellaan ohjelman suorituksen aikana myös sitä, mitä järjestelmän sisällä tapahtui. (Kasurinen 2013, 67-68; Patton 2006, 92, 106)

Olemassa on myös näitä kahta yhdistelevä harmaa laatikko -testaus. Sitä voidaan hyödyntää silloin, kun koko järjestelmää ei voi testata lasilaatikkona, mutta joidenkin komponenttien koodi on tarkasteltavissa. Järjestelmää testataan musta laatikko -tyylisillä testitapauksilla, mutta siinä voidaan myös kurkistaa ohjelman sisään ja tarkastella miten se toimii. (Kasurinen 2013, 68; Patton 2006, 218)

3.3.3 Regressiotestaus

Regressiotestaus tarkoittaa testausta, jossa koodiin tehtyjen muutosten jälkeen halutaan varmistaa, että uudet muutokset eivät ole tuoneet mukanaan regressioita eli virheitä, joiden vuoksi jokin toiminnallisuus ei enää toimi oikein. Regressiotestaus ei varsinaisesti ole testausmenetelmä, vaan sillä viitataan kaikenlaiseen testaamiseen, jonka tavoitteena on tarkistaa, että sovellus toimii muutosten jälkeen edelleen oikein. (Kasurinen 2013, 68-69)

3.4 Testaustasot

3.4.1 Yksikkötestaus

Yksikkötesteillä testataan jonkin yksittäisen moduulin, kuten metodin, funktion tai luokan, toiminta. Testeillä varmistetaan, että moduuli kääntyy virheettä, toimii määritellyllä tavalla ja reagoi oikein sille annettuihin syötteisiin. (Kasurinen 2013, 51-52)

Keskeneräiseen järjestelmään tehdyssä yksikkötestauksessa voi tulla tarve käyttämään testipetejä, jossa osa järjestelmän toiminnallisuudesta on simuloitu. Puuttuvat osat kor-

vataan käyttäen tynkiä, testiajureita ja jäljitelmäolioita. Tällöin moduulia voidaan testata silloin, kun muita sen tarvitsemia osia ei ole vielä toteutettu. Tynkien ja jäljitelmäolioiden avulla voidaan testata mitä tietoa yksikkö lähettää, mitä eri syötteillä tapahtuu ja tarkkailla kuinka niitä käytetään. Näitä voi tulla tarve käyttää myös silloin, kun on tarve testata yksikköä eristyksissä muusta järjestelmästä. (Haikala & Mikkonen 2011, 207; Kasurinen 2013, 52)

Hyvät ja kattavat yksikkötestit voivat toimia hyvin myös järjestelmän dokumentaationa. Niistä voidaan kätevästi selvittää, mitä moduulin kuuluu tehdä, miten sitä kuuluu käyttää ja miten sen on tarkoitus toimia muun järjestelmän kanssa.

3.4.2 Integrointitestaus

Integrointitestauksessa testataan moduulin toimivuus yhdessä muun järjestelmän ja muiden moduulien kanssa. Pääosin tutkimisen kohteena ovat rajapinnat. Integrointitestaus tapahtuu usein yksikkötestauksen rinnalla, eikä sitä tarvitse toteuttaa erillisenä operaationa. (Haikala & Mikkonen 2011, 207-208; Kasurinen 2013, 54-56)

3.4.3 Järjestelmätestaus

Järjestelmätestauksessa testataan täysin integroitu järjestelmä ja varmistetaan, että se toimii kokonaisuutena ja täyttää sille asetetut vaatimukset ja tavoitteet. Tässä vaiheessa ei enää ole tarvetta testityngille tai sijaiskomponenteille. Testaus tapahtuu kuitenkin vielä testiympäristössä, eikä lopullisessa tuotantoympäristössä. (Haikala & Mikkonen 2011, 208-209; Kasurinen 2013, 56-57)

3.4.4 Hyväksymistestaus

Hyväksymistestaus ei ole enää niin tekninen testausvaihe kuin edeltävät vaiheet. Tässä vaiheessa varmistetaan, että järjestelmä toimii vaatimusmäärittelyn mukaisesti hyväk-

syttävällä tavalla ja että asiakas olisi siihen tyytyväinen. Sovellusta saatetaan nyt testata suoraan tuotantoympäristössä. (Haikala & Mikkonen 2011, 208-209; Kasurinen 2013, 57)

4 TESTAUKSEN AUTOMATISOINTI

4.1 Automatisoinnista yleisesti

Testauksen automatisointi tarkoittaa manuaalisen testauksen suorittamista koneellisesti hyödyntäen joitakin työkaluja tai testaussovelluksia. Sillä pyritään vapauttamaan manuaaliseen testaamiseen uppoavia resursseja ja tekemään testauksesta johdonmukaisempaa ja kattavampaa.

Automatisoinnin tavoite ei ole kokonaan eliminoida manuaalista testausta, vaan vähentää manuaalisesti suoritettavien testitapausten määrää, keventämään testaamisen työtaakkaa ja antaa testaajalle enemmän aikaa keskittyä muihin toimintoihin. Joitakin testitapauksia ei edes välttämättä olisi mahdollista automatisoida, jos testaaminen vaatii testaajalta jotakin, jota tietokone ei voi korvata tai toteuttaa. (Kasurinen 2013, 76-77)

Itse testien lisäksi myös testien tekemistä voidaan automatisoida. Joidenkin työkalujen avulla voidaan generoida testitapauksia yksiköille. Työkalu saattaa kuitenkin generoida liian paljon testejä, ja ei välttämättä osaa ottaa kaikkea huomioon. Työkalut eivät pysty täysin päättelemään, mitä asioita kannattaa testata ja mitä sopii jättää pois. Sen tarkoitus onkin lähinnä helpottaa niiden tekemistä vähentämällä tietynlaisten testien kirjoittamisen tarvetta. (Fewster & Graham 1999, 19)

4.2 Automatisoinnin hyödyt ja ongelmat

Onnistuneella automatisoinnilla voidaan saavuttaa paljon merkittäviä etuja. Sen avulla voidaan saada suoritettua useampia testejä lyhyemmässä ajassa ja suoritus on johdonmukaisempaa. (Fewster & Graham 1999, 9-10)

Kun järjestelmään tuodaan uusia ominaisuuksia tai toiminnallisuutta, regressiotestaus on helpompi suorittaa vanhojen testien avulla. Näin varmistetaan, että olemassa oleva koodi toimii edelleen kuten pitää. (Fewster & Graham 1999, 9)

Kuten aiemmin mainittiin, automatisoimalla saadaan vähennettyä manuaalisiin testeihin uppoavia resursseja ja aikaa. Projektin resursseja voidaan hyödyntää paremmin ja testaajille jää enemmän aikaa keskittymään paremmin muiden manuaalisten testien suorittamiseen. Testaukseen käytetty aika vähenee, ja tuote voidaan saada markkinoille nopeammin. (Fewster & Graham 1999, 9; Dustin, Garrett & Gauf 2009, 23)

Automatisointi mahdollistaa myös sellaisten testien suorittamisen, jotka olisivat ihmiselle vaikeita tai jopa mahdottomia tehdä. Sillä mahdollistetaan esimerkiksi kuormitustestaus, jossa kaikki testikäyttäjät voidaan simuloida. (Fewster & Graham 1999, 9; Dustin ym. 2009, 23)

Vaikka automatisointi lupaa onnistuessaan suuria hyötyjä, tulee ottaa huomioon tiettyjä ongelmia ja vaatimuksia, jotta se voidaan saada onnistumaan.

Vaikka automatisoinnin tarkoituksena on vähentää manuaaliseen testaamiseen uppoavaa aikaa ja vaivaa, se tuo kehitykseen mahdollisesti hyvinkin paljon lisää vaadittavaa työtä. Uusia testejä on tehtävä uutta toiminnallisuutta kehitettäessä ja olemassa olevia testejä on ylläpidettävä. Kun sovellukseen tulee muutoksia, joutuvat usein testitkin muutoksien alle. Kun testien päivittämiseen menee enemmän aikaa ja vaivaa kuin niiden suorittamiseen manuaalisesti, tulee testiautomaatiosta tehokkuusmielessä hyödytön tai jopa haitallinen. (Fewster & Graham 1999, 11)

Mikäli käytössä oleva testauskäytäntö on heikko ja puutteellinen, ei testiautomatisointiin kannata ryhtyä. On kannattavampaa lähteä ensin kehittämään testauskäytäntöä paremmaksi ja edistää hyvää testauskäytäntöä, kuin lähteä edistämään huonoa testauskäytäntöä. Kuten Fewster ja Graham kirjassaan sanovat, ”kaaoksen automatisointi johtaa vain nopeampaan kaaokseen”. (Fewster & Graham 1999, 11)

Vaikka kaikki testit suoriutuisivatkin onnistuneesti, se ei tarkoita sitä, että järjestelmä on virheetön. Testit eivät välttämättä ota kaikkea huomioon, tai niissä voi olla itsessään virheitä. Testien tulokseen ei siis kannata täysin sokeasti luottaa. Testien laatuun on panostettava ja on pidettävä huoli siitä, että ne eivät sisällä virheitä. (Fewster & Graham 1999, 11, 23-24)

Testiautomaatiotyökaluissa voi itsessään olla myös virheitä. Ne ovat kuitenkin myös ihmisten tekemiä ohjelmistoja, eivätkä ole immuuneja virheille ja puutteelliselle tuelle. Voi myös olla, että työkalut eivät ole yhteensopivia muiden kehityksessä käytettyjen työkalujen ja sovellusten kanssa. (Fewster & Graham 1999, 11-12)

Testiautomaatio ei siis ole asia joka voidaan vain ottaa käyttöön ja toivoa että se välittömästi korjaa kaikki ohjelmistokehityksen ongelmat ja saa kehitystyön paremmaksi. Sen eteen on aluksi nähtävä runsaasti työtä, mutta ajan kuluessa se voi maksaa itsensä takaisin ja sen hyödyt voivat nousta esiin merkittävälläkin tavalla.

4.3 Mitä kannattaa automatisoida?

Kaikkea ei kannata, eikä edes voi, automatisoida. Joidenkin testien automatisointi ei tuota tarpeeksi hyötyä, että se olisi sen arvoista. Automatisoinnissa tulee ottaa huomioon budjetit, aikataulut ja osaaminen. Aikaa ja resursseja on rajallisesti, ja niitä ei riitä kaikkien mahdollisten asioiden testaaminen ei ole mahdollista. Täytyy siis arvioida, mitkä testit ovat järkeviä ja mahdollisia automatisoida. (Dustin ym. 2009, 134)

Automatisoitavat testitapaukset voidaan valikoida riskianalyysin perusteella. Ohjelman kriittiset osat joita eniten suoritetaan ja joiden on hyvin tärkeä toimia kunnolla ovat ensisijaisia automaation kohteita. (Dustin ym. 2009, 135)

Automatisoitavia testitapauksia valittaessa tulee ottaa huomioon, kuinka usein testi suoritetaan projektin aikana ja kuinka työläs se on toteuttaa. Tietysti ensisijaisesti kannattaa automatisoida testit jotka suoritetaan usein ja eivät ole liian vaikeita automatisoida. (Dustin ym. 2009, 135-137)

Testien uudelleenkäytettävyys tulevissa ohjelmistoversioissa tulee myös ottaa huomioon. Sellaisten ohjelman osien testausta, joiden toiminnallisuuden tiedetään muuttuvan usein, ei yleensä kannata lähteä automatisoimaan. Tällöin toiminnallisuuden muuttuessa kaikki testitkin olisi tehtävä uudelleen. Tällaisten kertaluontoisten testien arvo ei todennäköisesti ole tarpeeksi suuri siihen nähtyyn vaivaan verrattuna. (Dustin ym. 2009, 136)

4.4 Yksikkötestien automatisointi

Yksikkötestit ovat yleisin automatisoinnin kohde, ja sitä varten on olemassa lukemattomia määriä työkaluja ja järjestelmiä. Testauskehysten, kuten JUnit, avulla voidaan luoda yksikkötestejä ja hallita niitä ja niiden suorittamista. Työkalujen avulla voidaan luoda raportteja testien suorituksesta sekä mitata testikattavuutta.

Yksikkötestit tulisi toteuttaa siten, että ne eivät ole riippuvaisia muista testeistä eivätkä muuta sovelluksen tilaa suorituksen aikana, ja ne voidaan testata muista erillään missä järjestyksessä tahansa. Tällöin testejä voidaan myös suorittaa useita samanaikaisesti.

Automatisoidut yksikkötestit ovat tärkeä osa jatkuvaa integrointia (*continuous integration*). Jatkuva integrointi on käytäntö, jossa kehittäjät integroivat muutokset versionhallintaan mahdollisimman usein. Tällöin mahdollisten konfliktien määrä vähenee, kun niitä ei ehdi muodostua. Tehtyjen muutoksien jälkeen järjestelmä automaattisesti rakentaa suoritettavan sovelluksen ja ajaa testit sitä vasten. (Kasurinen 2013, 175)

4.5 Käyttöliittymätestauksen automatisointi

Käyttöliittymätestauksessa varmistetaan, että käyttöliittymä toimii tarkoitetulla tavalla ja että se on määrittelyjen mukainen. Käyttöliittymän elementit testataan ja varmistetaan että siirtyminen näkymästä toiseen toimii oikein. Käyttöliittymän toiminnallisuuden testaamisen automatisointi tapahtuu yksinkertaisimmillaan ohjelmalla, joka toistaa painikkeiden painamisia ja hiiren napsautuksia ruudulla ja tarkistaa, toimiko sovellus kuten odotettiin.

Käyttöliittymätestausta ei voi kuitenkaan täysin automatisoida. On joitakin asioita, joita tietokone ei ainakaan vielä lähivuosina voi ymmärtää, kuten käyttöliittymän käytettävyys ja sen ulkonäkö. Näiden arvioimiseen tarvitaan oikeaa käyttäjää. (Fewster & Graham 1999, 23)

Yleisin käyttöliittymätestauksen automatisointityökalu on sovellus, jonka avulla voidaan nauhoittaa napinpainallukset sekä tekstinsyötöt ja generoida makro, joka voidaan sitten suorittaa uudelleen testaustyökalulla. Generoitua makroa saattaa joutua jälkeenpäin vielä säätämään, mikäli halutaan vaikka vähentää toimintojen välistä viivettä tai jos jokin toiminto riippuu jostakin eri suorituskerroilla mahdollisesti eri viiveellä tapahtuvasta asiasta, kuten esimerkiksi sovelluksen käynnistyminen tai jonkin verkon yli tapahtuneen toiminnon jälkeen suoritettava käsky. (Patton 2006, 241-242)

Käyttöliittymätestauksessa voidaan hyödyntää ns. testiapinaa, kun halutaan simuloida käyttäjien toimintaa. Testiapina ei suinkaan viittaa mihinkään henkilöön tai eläimeen, vaan työkaluun jonka avulla voidaan syöttää joko satunnaisia, pseudosatunnaisia tai harkittuja syötteitä testattavaan sovellukseen riippuen sen älykkyystasosta. Tyhmä testiapina on yksinkertaisimmillaan vain sovellus, joka voi painaa satunnaisia näppäimiä näppäimistöllä ja satunnaisia pisteitä ruudulla. Älykkäämpi testiapina taas voi olla jollakin tasolla tietoinen ohjelman toiminnasta, käyttöliittymän elementeistä, niiden sijainneista ja niiden toiminnoista, ja saattaa yrittää simuloida oikean käyttäjän järjenjuoksua. (Patton 2006, 245-250)

5 KÄYTETTÄVÄT TEKNIIKAT

5.1 Android

Android on Googlen sekä Open Handset Alliance -konsortion mobiililaitteita varten kehittämä avoimen lähdekoodin käyttöjärjestelmä. Sen ytimenä toimii modifioitu Linux-ydin. Android on saatavilla useille eri suoritinarkkitehtuureille, kuten ARM, x86 ja MIPS. Se on maailman suosituin mobiilikäyttöjärjestelmä. (Android Developers 2017a)

Androidissa sovelluksia suorittaa Dalvik-virtuaalikone, tai uudemmissa versioissa oleva Dalvikin jatkaja Android Runtime (ART). Android-sovellukset ovat pääosin Java-ohjelmointikielellä toteutettuja. Käännetty Java-tavukoodi muunnetaan Dalvik-tavukoodimuotoon, jonka Dalvik/ART osaa suorittaa.

Androidista on julkaistu vuosien varrella useita eri versioita, joista monia on edelleen aktiivisessa käytössä. Eri versioiden välillä voi olla joitakin tiettyjä eroavaisuuksia, jotka täytyy ottaa sovelluskehityksessä huomioon, jos halutaan sovelluksen toimivan mahdollisimman monella laitteella. Tämän vuoksi Android-sovelluksia on suositeltavaa testata usealla eri Android-versiolla, jotta voidaan varmistaa, että toimivuusongelmia ei ole. Eri puhelinvalmistajien malleissa voi olla myös omia muutoksiaan Android-käyttöjärjestelmään, jotka voivat aiheuttaa yhteensopivuusongelmia. Android-sovelluksia täytyy joskus siis testata myös eri versioiden lisäksi eri puhelinmalleilla, mikäli vain mahdollista. Tätä varten ei onneksi ole aina pakko hankkia itse laitetta, vaan sen voi vuokrata joltakin puhelinfarmilta ja ajaa testit verkon ylitse.

5.2 Android Studio

Android Studio on Googlen kehittämä virallinen Android-ohjelmointiympäristö. Se kehitettiin käyttäen pohjana JetBrainsin kehittämää IntelliJ IDEA -ohjelmointiympäristöä, joka on yksi käytetyimmistä ja tunnetuimmista Java-ohjelmointiympäristöistä. Siinä on mukana useita Android-kehitystyötä helpottavia ominaisuuksia ja työkaluja,

kuten Android-emulaattori, käyttöliittymäeditori, debuggeri ja natiivikoodituki. (Android Developers 2017b, Android Developers 2017c)

Android Testing Support Library on Googlen kehittämä kirjasto, jonka avulla voidaan instrumentoiduissa eli suoraan Android-järjestelmässä suoritettavissa yksikkötesteissä hyödyntää JUnitia sekä käyttöliittymätestauksessa käyttää Espresso sekä UI Automator -työkaluja. Lisäksi muita testaustyökaluja saa helposti asennettua lisäosavalikon kautta. (Android Developers 2017d, Android Developers 2017e)

5.3 JUnit

JUnit on avoimen lähdekoodin yksikkötestausohjelmistokehys Java-ohjelmointikielen yksikkötestien luomiseen. Sen ovat luoneet Erich Gamma ja Kent Beck, ja se perustuu Kent Beckin alun perin Smalltalk-kielellä kehitettyyn yksikkötestausohjelmistokehykseen, SUnitiin. Tähän malliin pohjautuvia yksikkötestausohjelmistokehyksiä on sittemmin kehitetty monille eri ohjelmointikielille, jotka ovat nimetty kollektiivisesti xUnit-ohjelmistokehyksiksi. (Gregory, Leme, Massol & Tachiev 2011, 4)

5.4 Mockito

Mockito on avoimen lähdekoodin ohjelmistokehys Java-kielelle. Sillä voidaan luoda automaattiseen yksikkötestaukseen käytettäviä jäljitelmäolioita (*mock object*), jotka imitoivat ohjelman varsinaisten luokkien toiminnallisuutta. (Mockito 2017)

On joitakin asioita, joihin Mockito ei kuitenkaan kykene. Sillä ei muun muassa ole mahdollista tehdä jäljitelmiä staattisista metodeista eikä rakentajametodeista. On olemassa työkaluja jotka kykenevät myös tähän, mutta todennäköisesti tarve tehokkaammille työkaluille on merkki siitä, että ohjelman rakenteessa on ongelmia ja koodia tulisi muokata. (Mockito 2017)

5.5 Docker

Docker on avoimen lähdekoodin sovellus, jonka avulla voidaan pakata sovellus ja kaikki sen suorittamiseen tarvittavat osat yhteen eristettyyn virtuaaliseen säiliöön (*container*). Toisin kuin virtuaalikoneet, kontit eivät sisällä kokonaista käyttöjärjestelmää, vaan ainoastaan ne osat ja moduulit mitä sovelluksen suorittamiseen vaaditaan. (Docker Inc. 2017)

5.6 Node.js

Node.js on asynkroninen tapahtumaohjattu JavaScript-ajoympäristö, jonka avulla voidaan kehittää palvelintyökaluja ja verkkosovelluksia. (Node.js Foundation 2017) Se on monialustainen ja saatavilla muun muassa Windows-, macOS- ja Linux-käyttöjärjestelmille.
Node.js:n käyttämällä npm-pakettienhallinnalla voidaan ladata ja asentaa satoja tuhansia eri JavaScript-paketteja omiin projekteihin. (npm, Inc. 2017)

5.7 Appium

Appium on monialustainen avoimen lähdekoodin työkalu Android-, iOS- ja Windows-sovelluksien testauksen automatisointiin, ja sillä on mahdollista automatisoida Android-sovelluksia, verkkosovelluksia sekä näitä yhdisteleviä hybridisovelluksia. (Appium 2017)

Appium toimii palvelin-asiakas -arkkitehtuurilla. Se on ytimeltään web-palvelin, joka tarjoaa WebDriver-protokollan mukaisen REST-rajapinnan. WebDriver on selaimilla suoritettavaa testausta varten kehitetty rajapinta, ja Appium valjasti sen käyttöön Android-sovellusten testaamiseen. Rajapinnan kautta Appium-palvelin ottaa vastaan testipuhelimeen tai emulaattoriin ohjattavia käskyjä ja palauttaa suorituksen tuloksen. Rajapinnan ansiosta testejä voidaan tehdä lähes millä tahansa ohjelmointikielellä HTTP-pyyntöjen avulla. (Appium 2017)

5.8 Selenium Grid

Selenium Grid on Java-kielellä toteutettu välityspalvelin, joka välittää komentoja muihin testejä suorittaviin instansseihin, jotka käyttävät WebDriver-rajapintaa. Selenium Gridin avulla voidaan toteuttaa hajautettu testausympäristö. Se mahdollistaa testien suorituksen eri laitteilla samanaikaisesti. Yksi Selenium Grid -instanssi toimii keskussolmuna (hub), johon muut WebDriver-asiakasinstanssit yhdistetään solmuiksi. (Selenium Project 2017) Koska Appium hyödyntää WebDriver-rajapintaa, voidaan Appium yhdistää myös Selenium Gridiin.

6 MYSDK-PROJEKTIN TESTIYMPÄRISTÖ

6.1 Lähtökohdat

MySDK:ta kehitetään käyttäen Android Studiota. Koska projekti on tehty Android Studiolla, siinä on käytössä Gradle-koontityökalu. Sen avulla voidaan konfiguroida projektin koontiprosessi sekä hallita projektin riippuvuuksia build.gradle -tiedostojen kautta.

Gradlessa on tuki projekteille, jotka koostuvat eri aliprojekteista. MySDK-projekti on jaoteltu eri moduuleihin, joista jokainen on oma projektinsa. Tärkein moduuli on projektin ydin, *core*, jossa kaikki keskeisimmät toiminnallisuudet sijaitsevat. Kaikki eri maksu- ja kanava-SDK-paketit ovat eritelty omiksi moduuleikseen. Käyttörajapinta määritellään *master*-moduulissa, jossa sijaitsevat kaikki metodit joiden kautta MySDK:ta on tarkoitus käyttää.

Androidissa yksikkötestit voidaan luokitella kahteen ryhmään: paikallisiin yksikkötesteihin sekä instrumentoituihin yksikkötesteihin. Paikalliset yksikkötestit ovat testejä, jotka voidaan suorittaa tavallisella Java-virtuaalikoneella (JVM). Instrumentoidut yksikkötestit puolestaan täytyy suorittaa Android-järjestelmässä, joko emulaattorilla tai

fyysisellä laitteella. Jos testi tai testattava koodi käyttää Android API:a, on joko tehtävä instrumentoitu testi tai hyödynnettävä tynkiä tai jäljitelmäolioita.

6.2 Yksikkötestauksen käyttöönotto Android Studio -ympäristössä

Loin projektiin uuden Android-kirjastoprojektin nimeltä *testing*. Tähän projektiin määritellään testiriippuvuudet sekä luodaan luokat, joita halutaan hyödyntää eri projektien testeissä. Sitä voidaan hyödyntää muiden projektien paikallisissa ja instrumentoiduissa testeissä lisäämällä niiden build.gradle-tiedostoihin testien aikainen riippuvuus testing-projektiin (Kuva 3).

```
testCompile project(':testing')
androidTestCompile project(':testing')
```

Kuva 3. Testing-projektin asettaminen riippuvuudeksi muihin projekteihin

Paikallisia yksikkötestejä varteen lisäsin testing-projektin build.gradle-tiedostoon myös JUnit 4- sekä Mockito-riippuvuudet. SDK Manager -valikon kautta asennettiin Android Support Repository, jonka mukana tulee Android Testing Support Library -kirjasto. Projektissa se otettiin käyttöön lisäämällä build.gradle -tiedostoon tarvittavat riippuvuudet (kuva 4).

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'junit:junit:4.12'
    compile 'org.mockito:mockito-all:1.10.19'
    compile 'com.android.support.test:runner:0.5'
    compile 'com.android.support.test:rules:0.5'
    compile 'com.android.support.test.espresso:espresso-core:2.2.2'
    compile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'
}
```

Kuva 4. Testing-projektin riippuvuudet

Näillä eväillä projekti oli aika lailla valmis yksikkötestien toteuttamiseen. Yksikkötestit toteutetaan ja suoritetaan JUnitilla, ja muun muassa Android-riippuvuuksien korvaamiseen hyödynnetään Mockitoa. En halunnut ottaa käyttöön Mockitoa tehokkaampaa jäljittelykehystä, koska mielestäni projektissa pitäisi mieluummin korjata koodi paremmin testattavaan kuntoon kuin käyttää kyseenalaisia menetelmiä sen mahdollis-

tamiseksi vaikeasti testattavan koodin kanssa. Ison projektin kanssa tilanne olisi tietysti toisin, mutta koska MySDK-projekti on vielä suhteellisen pienikokoinen, ei ole vielä täysin mahdotonta tehdä isojakin muutoksia sen eteen.

JUnitilla testit voidaan pitää projektin muusta koodista erillään. JUnitissa testit sijaitsevat luokissa, jotka sisältävät testimetodeita. Eri testiluokkia voidaan ryhmitellä testisarjoihin (*suite*). Testisarjaan voidaan ryhmitellä myös muita testisarjoja. (Gregory ym. 2011, 21-23)

JUnitissa hyödynnetään koodissa metodeihin ja luokkiin asetettavia merkintöjä, joiden avulla voidaan muun muassa määritellä testiluokan testimetodit sekä alustus- ja lopetusmetodit, jotka suoritetaan joko ennen testiluokan (`@BeforeClass`) tai yksittäisen testin (`@Before`) suoritusta tai sen jälkeen (`@AfterClass`, `@After`). (Gregory ym. 2011, 15, 31)

Oikeellisuuden tarkistaminen tapahtuu *assert*-metodien avulla. Metodeilla muun muassa verrataan saatua tulosta odotettuun tulokseen ja ilmoitetaan virheestä, jos ne eivät täsmää. (Gregory ym. 2010, 15-16)

Voidaan haluta testata myös sitä, että metodi heittää oikeanlaisen poikkeuksen virhetilanteissa. Se voidaan määritellä lisäämällä `@Test`-merkintään parametri `expected=`*`ExpectedException`*. Testi onnistuu, mikäli koodi aiheuttaa määritellyn poikkeuksen. (Gregory ym. 2010, 44-45)

Testien suorittamisen hoitaa *runner*-luokka. Testiluokalle voidaan määritellä merkinnän `@RunWith` avulla runner-luokka, joka suorittaa testiluokan testit. Myös oman runner-luokan tekeminen on mahdollista periyttämällä `org.junit.runner.Runner` -luokka. (Gregory ym. 2010, 19-20)

Yksikkötestaamisessa tulee hyvin usein tarve jäljitelmäolioille, ja Android-kehityksessä niiltä on vaikea välttyä, sillä koodissa käytetyt viittaukset Androidin metodeihin ja olioihin eivät ilman Androidin sisällä suorittamista toimi, vaan ne on korvattava.

Mockito on jäljitelmäolioiden luomiseen tehty Java-ohjelmistokehys, jota voidaan hyödyntää tähän tarkoitukseen.

Mockitossa jäljitelmäolioiden luominen tapahtuu joko `@Mock`-merkinnällä tai staattisella metodilla `mock(Class aClass)`. Tämä olio voidaan antaa testattavaan koodiin, eikä se huomaa mitään eroa aidon ja jäljitelmäolion välillä. Oletuksena jäljitelmäoliot palauttavat metodikutsuissa paluuarvon tyypistä riippuen joko nollan, totuusarvon epätosi (*false*), tyhjän kokoelman tai null-arvon. Metodille voidaan myös määritellä arvo, jonka sen halutaan palauttavan. Jos testillä halutaan varmistaa, että testattava metodi toimii oikein myös poikkeustilanteissa, jäljitelmäolion kutsuttava metodi voidaan laittaa heittämään kutsuttaessa myös poikkeus. (Mockito 2017)

Android Studiossa instrumentoidut yksikkötestit sijaitsevat kansiossa *projekti*`/src/androidTest/java/`, ja paikalliset yksikkötestit kansiossa *projekti*`/src/test/java/`. Uuden testiluokan luominen tapahtuu joko luomalla uusi Java-luokka näihin kansioihin, tai Android Studion kautta.

Aluksi suunnittelin ottavani käyttöön testiajurin, joka suorittaisi useita testejä samanaikaisesti. Sen kanssa olisi kuitenkin mahdollisesti tullut kaikenlaisia ongelmia, joiden vuoksi se ei ollut kannattavaa. Testit saattavat suorituksen aikana esimerkiksi muuttaa muiden luokkien tilaa. Jos useat testit käsittelevät näitä luokkia samanaikaisesti, se voi tuottaa virheitä tai vääristää testien tuloksia. Lisäksi se voi tehdä suoritusraportin lukemisesta hieman epäselvempää, kun testit on ajettu melko sekavassa järjestyksessä. Lisäksi yksikkötestien suoritukseen ei kuuluisi mennä niin paljon aikaa, että tarvetta samanaikaiselle suoritukselle olisi.

7 JULKAISTAVIEN PELIEN TESTIYMPÄRISTÖ

7.1 Paikallisen testipalvelimen toteutus

Toimistolla on käytössä kourallinen testipuhelimia. Lähdin tutkimaan samanaikaisen testaamisen mahdollisuutta ja päätin kehittää järjestelmän, jonka avulla saataisiin testit suorittumaan kaikilla toimiston vapailla puhelimilla samanaikaisesti.

Toimistolle tuotiin muun muassa tätä tarkoitusta varten tietokone, johon oli asennettuna Ubuntu 16.04 LTS. Tavoitteena oli saada kytkettyä puhelimet tähän koneeseen kiinni ja saada kone suorittamaan sille lähetetyt testit niillä verkon yli.

Koska halusin työskennellä järjestelmän parissa myös toimiston ulkopuolella, täytyi keksiä jokin nopea ja helppo keino asentaa ja konfiguroida se toisiin testikoneisiin. Tätä varten päätin hyödyntää Dockeria. Sen avulla voidaan koko järjestelmä paketoida yhdeksi levykuvaksi (*image*) ja automatisoida vaiheet, jotka järjestelmän asentamiseen ja käyttöönottoon vaaditaan. Tämän ansiosta se voidaan helposti asentaa ja suorittaa eri järjestelmissä joissa on Docker asennettuna.

Docker-levykuvan luominen voidaan automatisoida *Dockerfile*-tiedoston avulla. Se on tekstidokumentti, jossa määritellään pohjana käytettävä levykuva sekä vaiheet, joilla se konfiguroidaan ja joilla siihen asennetaan tarvittavat paketit. Kun Docker-image on luotu, siitä voidaan luoda suorittuva säiliö. Eroa levykuvan ja säiliön välillä voidaan verrata luokan ja olion väliseen eroon, missä olio on luokan ajonaikainen ilmentymä.

Dockerissa on tarkoitus, että kontit ovat eristettyinä isäntäjärjestelmästä. Tämä tarkoittaa myös sitä, että oletuksena konttiin ei saa yhteyttä ulkopuolelta eikä kontti pysty esimerkiksi hyödyntämään isäntäkoneen USB-laitteita. Eristäytyneisyyteen voidaan kuitenkin vaikuttaa konfiguroinnilla. Dockerfile-tiedostossa voidaan määritellä `EXPOSE`-komennon avulla portit, jotka avataan ulospäin. Tämän lisäksi täytyy kuitenkin myös käynnistyskomennon `docker run` -yhteydessä antaa joko argumentti `-p [in-port]:[out-port]` jokaiselle portille, jossa `in-port` on Dockerin sisäinen portti ja `out-port` on portti johon ulkoapäin otetaan yhteys, tai argumentti `–`

`P/--publish-all=true` joka yhdistää kaikki Dockerfilessä määritetyt portit satunnaisiin isäntäkoneen vapaisiin portteihin.

Jotta USB-laitteet saadaan ohjattua konttiin, se täytyy käynnistää argumentein:
`--privileged –v /dev/bus/usb:/dev/bus/usb`
Tämä käynnistää kontin privileged-tilassa, joka antaa sille enemmän oikeuksia ja mahdollistaa pääsyn kiinni isäntäjärjestelmään. Lisäksi se ohjaa Dockerin USB-väylän isäntäkoneen USB-väylään, jolloin sillä on pääsy kaikkiin isäntäkoneeseen kytkettyihin laitteisiin. Tämä on kieltämättä jonkin verran Dockerin idean vastaista, mutta se toimii, eikä tässä tilanteessa ole tarvetta rajoittaa sen oikeuksia.

Kun yritin ottaa Dockeria käyttöön, huomasin, että sillä on kuitenkin joitakin rajoitteita. Dockeria ei saa ollenkaan 32-bittisille alustoille, ja esimerkiksi ARM-pohjaiset kontit eivät toimi kuin muissa ARM-pohjaisissa laitteissa. Tämä rajoitti mahdollisten testikoneiden määrää. En voinut hyödyntää kotonani lojuvia vanhoja tietokoneita enkä Raspberry Pi -laitteita.

Yritin ajaa kontin Mac-työkannettavallani Docker for Mac -sovelluksen avulla vain kuullakseni, että USB-laitteiden läpiajo ei ole sillä mahdollista. Sitä tarvitaan, jotta testipuhelimet saadaan siihen yhdistettyä. VirtualBoxilla tehty virtuaalikone toimi tarkoitukseen melko hyvin, mutta USB-laitteiden kanssa tuli silloin tällöin ongelmia. Jotkin laitteet toimivat liian hitaasti tai yhteys katkeili. Löysin kuitenkin käyttööni fyysisen koneen, jolla sain kaiken toimintakuntoon.

Käyttöliittymätesteihin suunniteltiin käytettäväksi Appiumia. Appium-palvelimen asennus tapahtuu Node.js:n kautta, jonka vuoksi täytyi myös se asentaa.

Appiumille lähetetyssä datassa annetaan käytettävän laitteen tietojen lisäksi myös APK-tiedoston sijainti, jolla testit on tarkoitus suorittaa. Tämä voi olla paikallisen testikoneella sijaitsevan tiedostosijainnin lisäksi myös URL, josta APK:n saa verkon yli ladattua. Sitä ei siis tarvitse aina käsin siirtää suoraan testikoneelle.

Python tekee tästä erittäin helppoa sen mukana tulevalla SimpleHTTPServer-moduulilla, joka mahdollistaa kansioiden jakamisen verkon yli. Kuvassa 5 kuvatut komennot

käynnistävät nimen mukaisen yksinkertaisen http-palvelimen sille annettuun porttiin (tai oletusporttiin 8000, jos porttia ei ole annettu). Palvelin jakaa verkon yli kansion, jossa komento suoritettiin. Kansioon ja sen tiedostoihin pääsee käsiksi koneen IP-osoitteella ja portilla.

```
python -m SingleHTTPServer [port]
python3 -m http.server [port]
```

Kuva 5. SimpleHTTPServer-moduulin käynnistyskomennot Pythonin versioille 2 ja 3

Appiumin rajoitteena on, että yksi palvelin voi suorittaa testit vain yhdelle laitteelle kerrallaan. Jokaiselle laitteelle joudutaan luomaan oma instanssi, jos halutaan ajaa testejä samanaikaisesti usealle laitteelle. Koska instanssien käynnistäminen ja sulkeminen manuaalisesti ovat melko rasittavaa puuhaa, päätin selvittää, olisiko mahdollista tehdä skripti, joka tunnistaa kytketyt laitteet ja luo niitä varten uuden palvelininstanssin ja sulkee sen, kun laite irrotetaan. Dockerin yhteydessä Node.js on hyvin suosittu vaihtoehto, ja koska se jo asennettiin Appiumin asentamista varten, päätin hyödyntää sitä tässäkin.

Adbkit on Node.js-paketti, jonka avulla voidaan Node.js:n kautta suorittaa komentoja ADB:n kautta. Sen avulla voidaan myös tarkkailla, jos laite kytketään kiinni koneeseen tai irrotetaan. Tämä paketti sopi tarkoituksiini erittäin hyvin.

Aina, kun puhelin kytketään kiinni testikoneeseen, skripti huomaa sen ja käynnistää sille uuden oman Appium-prosessin. Kun puhelin irrotetaan, prosessi suljetaan. Jokainen Appium-instanssi vaatii oman portin, jonka kautta sitä ohjataan. Instansseja luotaessa tarvitaan siis jokin avoin portti. Node.js:ssä tämä onnistuu helposti pienellä kikalla. Ensin käynnistetään http-palvelin ja asetetaan se kuuntelemaan porttia 0. Asettamalla portin nollaan se hakee järjestelmästä jonkin satunnaisen avoinna olevan portin. Annettu portti otetaan talteen ja suljetaan palvelin. Nyt portti on taas avoin, ja sitä voidaan käyttää muualla.

Tällä hetkellä testit täytyy testikoodin konfiguraation kautta ohjata oikeaan Appium-instanssiin. Jokaiselle puhelimelle täytyy siis selvittää sen Appium-instanssin portti, ja viitata siihen konfiguraatiossa. Jos portti vaihtuu, esimerkiksi irrottamalla puhelin ja

laittamalla se takaisin kiinni, se täytyy selvittää uudelleen tai testi ei mene oikeaan paikkaan. Lisäksi tällä hetkellä ei ole kätevää keinoa selvittää, mitä laitteita on kytkettynä menemättä katsomaan palvelimelta tai suoraan fyysiseltä laitteelta.

Kun suunnittelin mahdollisen rajapinnan kehittämistä puhelimien tietojen ja niiden vastaavien Appium-palvelinten porttien kyselemiseen palvelimelta, törmäsin Selenium Gridiin. Selenium Grid mahdollistaa useiden Appium-instanssien kokoamisen yhden IP-osoitteen ja portin taakse. Appium voidaan konfiguroida rekisteröimään itsensä Gridiin uudeksi solmuksi, ja kun testit lähetetään Appiumin sijaan Gridille, Grid ohjaa sen testikoodissa määriteltyjen ominaisuuksien perusteella oikeaan solmuun. Lisäksi kaikki yhdistetyt solmut ja niiden tiedot voidaan nähdä selaimen kautta Gridin hallintapaneelista (Kuva 6).

Se Grid Console v.3.3.1
Help

DefaultRemoteProxy (version : 1.6.3)
id : http://0.0.0.0:35043, OS : ANY
Browsers Configuration
WebDriver
v:4.2.2NX503A

DefaultRemoteProxy (version : 1.6.3)
id : http://0.0.0.0:36093, OS : ANY
Browsers Configuration
browserTimeout: 0
debug: false
help: false
port: 36093
role: node
timeout: 30000
cleanUpCycle: 2000
host: 0.0.0.0
maxSession: 1
servlets:
capabilities: Capabilities [{browserName=GT-I9195, platformName=ANDROID, maxInstances=1, deviceName=077d7360, version=6.0.1, applicationName=GT-I9195}]
downPollingLimit: 2
hub: http://localhost:4444
id: http://0.0.0.0:36093
hubHost: 127.0.0.1
hubPort: 4444
nodePolling: 5000
nodeStatusCheckTimeout: 5000
proxy:
org.openqa.grid.selenium.proxy.DefaultRemoteProxy
register: true
registerCycle: 5000
remoteHost: http://0.0.0.0:36093
unregisterIfStillDownAfter: 60000

view config

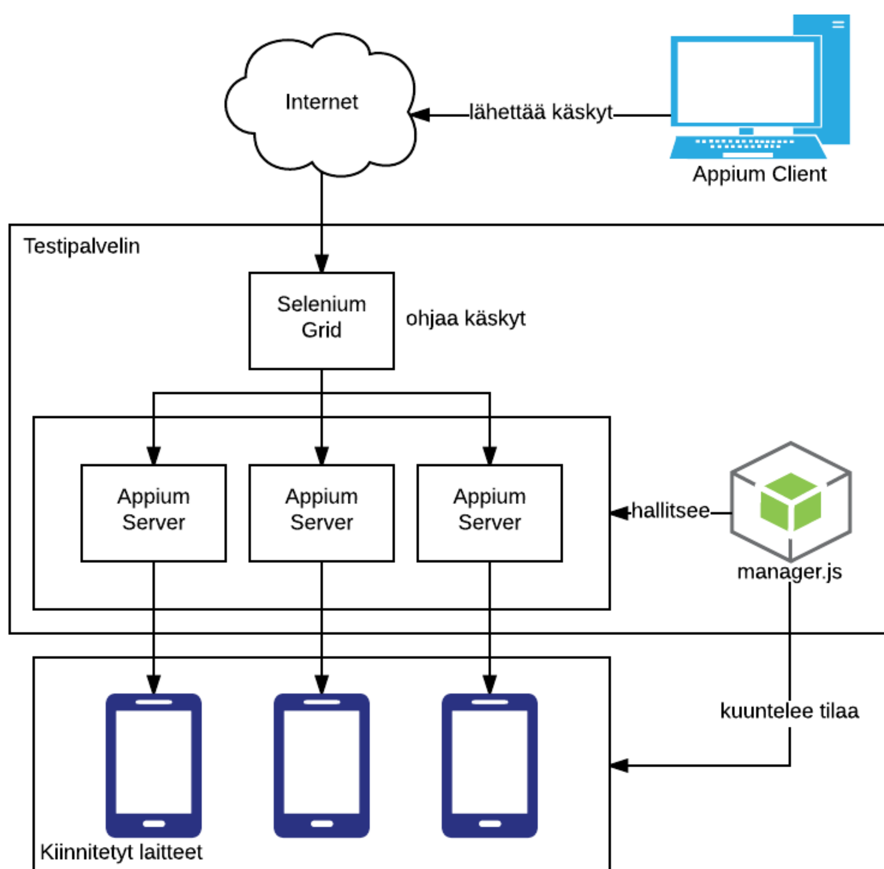
Kuva 6. Selenium Grid -hallintapaneeli kahden puhelimen ollessa kytkettynä

Selenium Gridiin rekisteröitymistä varten Appium-palvelimelle pitää antaa polku JSON-tiedostoon, joka sisältää Selenium Gridin vaatimia solmukohtaisia asetuksia,

kuten solmun laitteen tiedot. Päivitin edellä mainitun skriptin generoimaan puhelinta kytkettäessä tämän JSON-tiedoston tiedoilla, jotka saatiin laitteesta luettua Adbkitin avulla. Laitteesta otetaan sen nimi, asennettu Android-versio sekä laitekohtainen ID.

Selenium Gridin toiminnallisuutta voidaan laajentaa Java Servlettien avulla. Servletit ovat Java-ohjelmakomponentteja, joiden avulla voidaan Java-palvelimen toiminnallisuutta dynaamisesti laajentaa. Gridiin voidaan toteuttaa servlet, joka mahdollistaa vaikkapa kaikkien solmujen tietojen hakemisen palvelimelta JSON-muodossa. Niitä voidaan sitten hyödyntää testikoodissa, jos halutaan saada kaikkien laitteiden tiedot testien niillä suorittamista varten.

Lopputuloksena syntyi järjestelmä, jonka yksinkertaistettu rakenne on kuvattu kuvassa 7.



Kuva 7. Yksinkertaistettu kuvaus järjestelmästä

Testejä suorittaessa huomasin, että yksi puhelin aiheuttaa ongelmia. Vivo X9 -puhelimessa ilmestyy ennen asennusta varmistusdialogi, joka kysyy käyttäjältä, halutaanko

sovellus asentaa. En löytänyt puhelimesta asetusta josta sen olisi saanut pois päältä. Käynnistäessä testit täytyy aina käydä manuaalisesti naputtelemassa dialogi pois, jotta testattava peli saadaan asennettua ja testit pääsevät suoriutumaan.

Järjestelmää voidaan vielä kehittää. Järjestelmä ei esimerkiksi vielä tallenna suorituksen aikana puhelimesta saatua lokitietoa. Lokitiedoista olisi merkittävä apu testin epäonnistumisen syyn selvittämiseen. En kuitenkaan ehtinyt vielä tutustua tähän opinnäytetyön aikana, mutta uskon, että se toteutetaan hyvin pian, jos järjestelmää aletaan kehittää eteenpäin. Järjestelmää voidaan tulevaisuudessa mahdollisesti hyödyntää myös instrumentoitujen testien suorittamiseen MySDK-projektissa, johon Appiumista löytyy tuki.

7.2 Käyttöliittymätestiympäristön konfigurointi

Käyttöliittymätestit toteutetaan hyödyntäen Appiumin lisäksi myös JUnitia. Testejä varten tein oman Java-projektin, johon olisi tarkoitus säilöä kaikki käyttöliittymätestit ja niiden vaatimat apumetodit.

Appium-palvelimen käskyttämiseen käytetään sille tehtyä Java-kirjastoa. Aluksi aioin käyttää Pythonia tähän tarkoitukseen, mutta huomasin, että vaikka sillä tehty testikoodi olikin yksinkertaisempaa ja selkeämpää, oma Python-osaaminen ei vielä riittänyt joidenkin asioiden toteuttamiseen riittävän fiksulla tavalla. Lisäksi Javalle vaikutti olevan olemassa enemmän esimerkkejä, ja koska Java on kuitenkin jo ennestään aktiivisessa käytössä, päätin siirtyä siihen.

Appiumissa käskyjä ohjaa palvelimelle ajuriluokka (*driver*). Appiumin `AndroidDriver`-luokka on Android-laitteisiin keskittyvä ajuri, jossa on hyödyllisiä metodeita Android-laitteiden testaamiseen Appium-palvelimen kautta.
Koska halusin lisätä joitakin toiminnallisuuksia ajuriin, periytin luokasta `AndroidDriver` oman `ExtendedAndroidDriver`-luokan, johon ne lisäsin (Kuva 8). Yksi apumetodi oli esimerkiksi metodi `waitForActivity(String activity, long timeout)`, joka odottaa *timeout*-parametrissa määritellyn ajan että haluttu Activity tulee aktiiviseksi (Kuva 9).

```java
public class ExtendedAndroidDriver<T extends WebElement> extends AndroidDriver<T> {

    public ExtendedAndroidDriver(HttpCommandExecutor executor, Capabilities capabilities) {
        super(executor, capabilities);
    }

    public ExtendedAndroidDriver(URL remoteAddress, Capabilities desiredCapabilities) {
        super(remoteAddress, desiredCapabilities);
    }

    ...

    public ExtendedAndroidDriver(Capabilities desiredCapabilities) { super(desiredCapabilities); }

    public Dimension getScreenSize() { return this.manage().window().getSize(); }

    public Point getPointOnGrid(int col, int row, int cols, int rows) {...}

    public Point getPercentualPoint(double xPercent, double yPercent) {...}

    public boolean waitForActivity(String activity, long timeout) {...}
}
```

Kuva 8. Ote ExtenderAndroidDriver-luokasta

```java
public boolean waitForActivity(String activity, long timeout) {
    try {
        WebDriverWait wait = new WebDriverWait( driver: this, timeout);
        wait.until(function -> currentActivity().equals(activity));
        return true;
    } catch (TimeoutException ex) {
        return false;
    }
}
```

Kuva 9. waitForActivity-metodi

Kun Appiumin halutaan ottavan yhteys Appium-palvelimeen, sille annetaan tietysti palvelimen osoitteen lisäksi halutun testilaitteen tiedot (*desired capabilities*). Näiden tietojen avulla palvelin ohjaa suoritettavat testit oikeaan laitteeseen, mikäli mahdollisia testilaitteita on useita.

Parameterisoitujen testien suorittamista varten JUnit tarjoaa ajuriluokan `Parameterized`. Sen avulla voidaan sama testiluokka suorittaa useilla eri parametreilla. Lisäämällä luokkaan julkisen staattisen metodin joka palauttaa `Collection<Object[]>`-kokoelman ja asettamalla tälle annotaation `@Parameters`, se suorittaa testiluokan testit jokaiselle parametriparille ohjaten parametrit järjestyksessä joko testiluokan konstruktoriin tai annotaatiolla `@Parameter` merkittyihin julkisiin kenttiin.

Testien suorittamista varten tein `Parameterized`-ajurista mukautetun testiajuriluokan, joka ottaa vastaan parametreja sekä ajaa useita testejä samanaikaisesti. Parametrien kautta määritellään testikohtaisesti APK-tiedoston sijainti, palvelimen osoite sekä testilaitteen tiedot. Koska tarkoituksena olisi suorittaa testejä usealle APK-tiedostolle usealla puhelimella, täytyy generoida parametrijoukko, jolla suoritetaan jokainen testi jokaiselle APK:lle jokaisella puhelimella. Tein parametrien tarjoilemista varten *DataProvider*-luokan, johon voidaan määritellä testiajossa käytettävät asetukset ja jonka staattinen metodi `getData()` palauttaa testeissä käytettävien parametrijoukkojen kokoelman. DataProvider hakee parametreja varten testiluokalle määritetystä YAML-konfiguraatiotiedostosta käytettävän palvelimen osoitteen sekä APK-tiedostot sisältävän kansion polun.

Parametreja varten tarvittiin testikoneeseen liitettyjen puhelimien eli Selenium Gridiin rekisteröityjen solmujen tiedot. Oletuksena Selenium Gridistä voi solmujen tiedot nähdä ainoastaan hallintapaneelin kautta. Toteutin luokan, joka noutaa ja parsii hallintapaneelisivun html-lähdekoodista käytettävien solmujen tiedot. Tulevaisuudessa voin ottaa Selenium Gridiin käyttöön servletin, jolla solmujen tiedot voisi palvelimelta saada JSON-muodossa, mutta tällä hetkellä tämä ratkaisu on riittävä.

Palvelimelta tarvitaan pääsy testattaviin APK-tiedostoihin. Lisäsin projektiin luokan, jolla voidaan käynnistää tiedostopalvelimen, jolla jaetaan kansio verkon yli. Nyt palvelin voi päästä käsiksi testattavat APK-tiedostot sisältävään kansioon ja sen sisältöön, kun testit suoritetaan.

Kun testien suoritus aloitetaan, alussa ajuri hakee DataProvider-luokalta parametrijoukot millä testiluokan testit suoritetaan. Konfiguraatiotiedostosta haetaan palvelimen osoite sekä APK-tiedostojen sijainti, ja käynnistetään tiedostopalvelin jakamaan tämä kansio verkon yli. DataProvider hakee palvelimelta laitteiden tiedot sekä lukee APK-tiedostot sisältävän kansion sisällön läpi. Kun tiedot on haettu, se luo parametriyhdistelmät jokaiselle APK-tiedostolle ja laitteen tiedoille. Sen jälkeen testejä aletaan suorittaa jokaiselle näistä parametrijoukoista.

Nyt testejä voidaan suorittaa. Tein vielä testien tekemistä varten joitakin sitä helpottavia ominaisuuksia.

Joidenkin pelien kohdalla pelin valikoissa liikkuminen joudutaan tekemään painamalla tiettyjä pisteitä näytöllä. Näinhän se tietysti aina tehdään, mutta tässä tapauksessa kyse on siitä, että nappien koordinaatit joudutaan itse määrittelemään. Peli ei hyödynnä Android-käyttöliittymäelementtejä, joka vaikeuttaa valikon kanssa toimimista huomattavasti, koska niihin ei voida esimerkiksi niille annettujen resurssi-ID:iden kautta viitata taikka hakea esimerkiksi elementin tyypin mukaan.

Tein näytön näppäilyn parantamiseksi `ExtendedAndroidDriver`-luokkaan kaksi apumetodia (Kuva 10). Ensimmäinen metodi palauttaa prosenttiarvon mukaan x-y-koordinaatit ruudulla. Toisen metodin avulla voi jakaa näytön alueen ruudukkoon. Ruudukon koko määritellään sille annetuin parametrein, ja palauttaa halutun ruudun keskipisteen koordinaatit. Näiden metodien avulla voidaan osoittaa aina samaan pisteeseen näytön koosta huolimatta.

```
public Dimension getScreenSize() {
    return this.manage().window().getSize();
}

public Point getPointOnGrid(int col, int row, int cols, int rows) {
    Dimension dimensions = getScreenSize();
    double cellWidth = dimensions.getWidth() / cols;
    double cellHeight = dimensions.getHeight() / rows;
    double x = (cellWidth * col) + (cellWidth / 2);
    double y = (cellHeight * row) + (cellHeight / 2);
    int xint = (int)Math.round(x);
    int yint = (int)Math.round(y);
    return new Point(xint,yint);
}

public Point getPercentualPoint(double xPercent, double yPercent) {
    Dimension dimension = getScreenSize();
    int width = dimension.getWidth();
    int height = dimension.getHeight();
    double x = (width/100) * xPercent;
    double y = (height/100) * yPercent;
    return new Point((int)Math.round(x),(int)Math.round(y));
}
```

Kuva 10. getPointOnGrid- ja getPercentualPoint-metodit

Metodit ratkaisevat ainakin suoraan näytön koosta aiheutuvat ongelmat, mutta peli saattaa skaalata ja sijoittaa käyttöliittymäelementtejä eri tavoin resoluutiosta riippuen. Tämä voidaan ottaa pelikohtaisissa testeissä huomioon tarkistamalla laitteen resoluutio ennen tehtäviä painalluksia ja säädellä sen mukaan. Yritin tähän tarkoitukseen ottaa käyttöön kuvantunnistustekniikkaa, mutta omissa testeissäni toteutukset joita yritin

hyödyntää eivät toimineet tarpeeksi tarkasti. Päätin, että palaan aiheeseen joskus myöhemmällä ajanjaksolla opinnäytetyön ulkopuolella, jos aikaa ja tarvetta on.

8 YHTEENVETO

Alun perin opinnäytetyössä oli tarkoitus tehdä testiympäristön lisäksi myös testit. Testiympäristöä kuntoon laittaessani ja testien tekemiseen vaaditun työn määrää arvioidessani kuitenkin totesin, että se alkoi olla jo aivan liikaa. Tämän vuoksi rajoitin aiheen toteutuksen koskemaan vain automatisoitavien testien tekemisen ja suorittamisen mahdollistavien työkalujen ja ohjelmistokehyksien käyttöönottoon.

Projektia piti tehdä muiden tehtävien ohella, joita oli paljon. Alkuperäinen tavoite oli saada projekti tehtyä vuoden 2016 joulukuussa. Opinnäytetyön projekti ei ollut kuitenkaan tärkeämpien ja kiireellisempien tehtävien vuoksi tarpeeksi korkealla yrityksen prioriteettilistalla, jonka vuoksi myös sen aloittaminen venähti jonkin verran. Opinnäytetyön toteutuksen aikana pariin otteeseen vastaan tuli yllättäviä tilanteita, joiden vuoksi opinnäytetyö jouduttiin jättämään lähes kuukaudeksi sivuun.

Testipalvelimen toteuttaminen osoittautui erityisesti melko miellyttäväksi projektiksi, kun kasasi eri palasista yhteen toimivan kokonaisuuden. Pääsin tutustumaan Dockeriin ja Node.js:ään, sekä vähän kehittämään Linux-osaamistani. Kun järjestelmän sai vihdoin toimimaan, oli hienoa nähdä testejä suorittumassa kaikissa testipalvelimeen kytketyissä testipuhelimissa samanaikaisesti.

Toivon, että nyt testien tekemiselle ei olisi enää niin isoa kynnystä kuin ennen. Koska testien tekeminen olisi vaatinut ensin testiympäristön pystyyn laittamista, siihen ei ajan puutteen vuoksi ollut haluttu lähteä. Nyt kun testiympäristö on valmiina, testejä voidaan vain lähteä tekemään, kun siltä tuntuu. En usko, että vanhaan koodiin ainakaan vielä lähiaikoina luoda monia yksikkötestejä, koska ilman merkittäviä muutoksia koodiin se osoittautui hankalaksi ja monimutkaiseksi. Projektin kanssa on kuitenkin harkittu vähän isompiakin muutoksia, joiden jälkeen koodi olisi helpommin testattavaa.

Uuden koodin kanssa testausta voidaan hyödyntää, ja huomasin itsekin, että pyrkimällä tekemään koodista alusta asti testattavaa, se tuntuu tuottavan lopputuloksena myös laadukkaampaa koodia.

LÄHTEET

Android Developers 2017a. Android, the world's most popular mobile platform. Viitattu 8.2.2017. Saatavissa: https://developer.android.com/about/index.html

Android Developers 2017b. Meet Android Studio. Viitattu 13.2.2017. Saatavissa: https://developer.android.com/studio/intro/index.html

Android Developers 2017c. Android Studio Features. Viitattu 13.2.2017. Saatavissa: https://developer.android.com/studio/features.html

Android Developers 2017d. Test your app. Viitattu 13.2.2017. Saatavissa: https://developer.android.com/studio/test/index.html

Android Developers 2017e. Getting Started with Testing. Viitattu 13.2.2017. Saatavissa: https://developer.android.com/training/testing/start/index.html

Appium 2017. Introduction. Viitattu 15.2.2017. Saatavissa: http://appium.io/introduction.html

Dustin, E., Garrett, T., Gauf, B. 2009. Implementing Automated Software Testing. Stoughton, Massachusetts. Addison-Wesley.

Docker Inc. 2017. What is Docker? Viitattu 18.2.2017. Saatavissa: https://www.docker.com/what-docker

Fewster, M., Graham, D. 1999. Software Test Automation. New York. Addison-Wesley.

Gregory, G., Leme, F., Massol, V., Tahchiev, P. 2010. JUnit in Action, Second Edition. Greenwich. Manning Publications Co.

Haikala, I., Mikkonen, Y. 2011. Ohjelmistotuotannon käytännöt. 12. uud. p. Helsinki. Talentum Media Oy.

Kasurinen, J. P. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä. Docendo Finland Oy.

Mockito. 2017. Mockito FAQ. Viitattu 19.2.2017. Saatavissa: https://github.com/mockito/mockito/wiki/FAQ

Myers, J., Badgett, T., Sandler, C. 2012. The Art of Software Testing. 3. p. Hoboken, New Jersey. John Wiley & Sons, Inc.

Node.js Foundation 2017. About | Node.js. Viitattu 18.2.2017. Saatavissa: https://nodejs.org/en/about/

npm, Inc. 2017. npm:n kotisivu. Viitattu 18.2.2017. Saatavissa: https://npmjs.com

Patton, R. 2006. Software Testing, Second Edition. Indianapolis, Indiana. Sams Publishing.

Selenium Project 2017. Selenium-Grid – Selenium Documentation. Haettu 15.3.2017. Saatavilla: http://www.seleniumhq.org/docs/07_selenium_grid.jsp