View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Theseus

WWW.JAMK.FI

ANDROID-SOVELLUSTEN TESTAUS JA AUTOMATISOINTI

Toteutusmahdollisuudet FreeNest-ympäristössä

Janne Pekkarinen

Opinnäytetyö
Toukokuu 2013

Ohjelmistotekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN KUVAILULEHTI

<table>
<tr><td rowspan="3">Tekijä(t)
PEKKARINEN, Janne</td><td>Julkaisun laji
Opinnäytetyö</td><td>Päivämäärä
17.05.2013</td></tr>
<tr><td>Sivumäärä
59 + 12</td><td>Julkaisun kieli
Suomi</td></tr>
<tr><td></td><td>Verkkojulkaisulupa myönnetty
(X)</td></tr>
<tr><td colspan="3">Työn nimi
Android-sovellusten testaus ja automatisointi – Toteutusmahdollisuudet FreeNest-ympäristössä</td></tr>
<tr><td colspan="3">Koulutusohjelma
Ohjelmistotekniikan koulutusohjelma</td></tr>
<tr><td colspan="3">Työn ohjaaja(t)
PELTOMÄKI, Juha</td></tr>
<tr><td colspan="3">Toimeksiantaja(t)
RINTAMÄKI, Marko</td></tr>
</table>

Tiivistelmä

Opinnäytetyön tarkoituksena oli tutkia, miten Android-sovelluksia voidaan testata ja kuinka tätä prosessia voitaisiin automatisoida. Tutkimusten perusteella arvioitiin sitä, miten hyvin erilaisia Andoid-sovellusten testaukseen tehtyjä testauskehyksiä voidaan hyödyntää osana FreeNest-ympäristöä.

Työssä käytiin läpi ohjelmistotestauksen eri menetelmät ja testauksen teoriaa. Näiden tietojen pohjalta tutkittavaksi valittiin muutamia varteenotettavimpia vaihtoehtoja testauksen eri vaiheisiin. Tutkimuksessa käsiteltiin testausautomaatiota ja sen toimivuutta yleisesti paikallisella koneella ja kehitysympäristössä. Tämän jälkeen testattuja työkaluja sovellettiin käytettäväksi erillisellä testauspalvelimella.

Seuraavaksi aikaisempien testien tuloksia yritettiin soveltaa FreeNest-ympäristön käyttöön, jotta saataisiin jonkinlainen mielikuva siitä, miten Android sovellusten kehittäminen onnistuisi FreeNest:iä hyödyntäen.

Lopputuloksena saatiin suosituksia siitä, millaisia työkaluja Android-sovellusten testauksessa kannattaa käyttää, kun käytetään FreeNest-ympäristöä, ja mitä tulee ottaa huomioon. Tulokset pyrittiin esittämään myös yleisellä tasolla, joten niistä saattaa olla hyötyä myös muita testausympäristöjä ajatellen.

Suurin osa opinnäytteelle asetetuista tavoitteista saatiin toteutettua, mutta aiheen laajuuden vuoksi tuntuma käytettyihin mahdollisuuksiin jäi hieman pinnalliseksi.

Avainsanat (asiasanat)
Android, FreeNest, Jenkins, Robotium, Robolectric, testausautomaatio

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s) PEKKARINEN, Janne	Type of publication Bachelor´s Thesis	Date 17052013
	Pages 59 + 12	Language Finnish
		Permission for web publication (X)

Title
Android application testing and automation – Implementation possibilities in FreeNest environment

Degree Programme
Software Engineering

Tutor(s)
PELTOMÄKI, Juha

Assigned by
RINTAMÄKI, Marko

Abstract

The purpose of this thesis was to study how Android applications could be tested. Furthermore, the thesis discusses what the possibilities to automate application testing are. Based on the research results there was evaluation about how well different testing frameworks can be used as a part of FreeNest.

The thesis goes through different methods and principles (which are) used in software testing. The theoretical part of the thesis presents basic knowledge needed when evaluating the available options. The most promising tools and frameworks were studied more in detail and their possibilities were tested in different parts of software testing. The research focused first on general usability and usage on the local computer and development environment, after which the tools were re-evaluated, however, this time only with a separate testing server.

The results from previous studies were then applied into practice on FreeNest. This way some kind of conception was gained for the question: “How could Android application testing be carried out using FreeNest?.

As a result, the thesis gives recommendations about the usability of the tested tools in FreeNest context and what should be taken into account. The results were introduced also in a more general way and some of the information might be useful when automated testing is under consideration.

Most of the goals set to the thesis were fulfilled and the results are positively encouraging. Due to the wide scope of the topic, some of the areas were analysed or discussed only briefly.

Keywords
Android, FreeNEST, Jenkins, Robotium, Robolectric, testing automation

Miscellaneous

SISÄLTÖ

KÄSITTEET....4
1. TYÖN LÄHTÖKOHDAT....7
1.1. Tavoitteet....7
1.2. Työnantajan taustat....7
1.3. Android-käyttöjärjestelmä....8
2. SOVELLUSTESTAUS YLEISESTI....8
2.1. Johdanto....8
2.2. Miksi testataan?....9
2.3. Mitä testataan?....11
2.4. Miten testataan?....15
2.4.1. Yleisesti....15
2.4.2. Testauksen tasot....16
2.4.3. Testauksen lähestymistavat....20
2.5. Testauksen standardit ja historia....23
3. ANDROID-SOVELLUSTEN TESTAUS....24
3.1. Johdanto alemman tason testauksiin....24
3.2. Alemman tason testaus....25
3.2.1. Yleistä yksikkötestauksesta....25
3.2.1. Junit....25
3.2.2. Monkey....28
3.2.3. Mockito....29
3.3. Android sovellusten käyttöliittymätestaus....30
3.3.1. Yleistä käyttöliittymistä....30
3.3.2. Robotium....32
3.3.3. Robolectric....34
3.3.4. Calabash....36
4. ANDROID-TESTIEN AUTOMATISOIMINEN JA JATKUVA INTEGROINTI....38
4.1. Johdanto automatisoituihin testeihin....38
4.2. Jenkins....39

4.2.1. Jenkinsin yleiskuvaus..39
4.2.2. Jenkins ja Android-sovellusten testaus..41
4.3. Robot Framework..42
4.3.1. Mikä on Robot Framework?..42
4.3.2. Mahdollisuudet Android-sovellusten kanssa..42
5. KÄYTÄNNÖN TOTEUTUS...43
5.1. Robotiumin hyödyntäminen Jenkinsissä...43
5.2. Robolectricin hyödyntäminen Jenkinsissä...45
5.3. Calabash ja Jenkins...46
6. FREENEST JA ANDROID-KEHITYS..47
6.1. Mikä on FreeNest?...47
6.2. Yleinen sovelluskehitys...48
6.3. FreeNest yhdistettynä Jenkinsin kanssa...49
6.4. FreeNest Robot Frameworkin kanssa...49
7. YHTEENVETO..50
7.1. Johdanto...50
7.2. Testaus ilman keskitettyä palvelinta...50
7.3. Testauspalvelimen kanssa automatisoidusti...52
7.4. Kaupalliset vaihtoehdot...54
8. POHDINTA...55
8.1. Tutkimus kokonaisuutena..55
8.2. Tulevaisuuden kehityskohteita..56
LÄHTEET...57
LIITTEET...60
Liite 1: JUnit4 testin perusrunko...60
Liite 2: Mockito-testin esimerkkiluokat...61
Liite 3: Yksinkertainen Robotium-testi...62
Liite 4: Robotium-projektin build.xml..63
Liite 5: Robolectric testiprojektin luominen..63
Liite 6: Robolectric-projektin build.xml...65
Liite 7: Calabashin asentaminen..67
Liite 8: Jenkinsin asentaminen FreeNest-ympäristöön.......................................69

Liite 9: Robotium projektin määritteleminen Jenkinsissä....................................70

TAULUKOT

Taulukko 1. Virheiden aiheuttamat korjauskustannukset projektin eri vaiheissa........10

Taulukko 2. Eri Android versioiden jakauma aikavälillä 21.01.13 – 04.02.13..............31

KUVIOT

Kuvio 1. Aktiviteetin elinkaari ja siirtymävaiheiden metodit......................................12

Kuvio 2. Verifiointi, validointi ja testauksen tasot..16

Kuvio 3. Laatikkomallien erilaisuus havainnollistettuna...21

Kuvio 4. Monkey stressitestin tulos Jenkinssissä..29

Kuvio 5. Testien tulokset suorituksen jälkeen Eclipse-ympäristössä...........................33

Kuvio 6. Robolectric testin tulos..35

Kuvio 7. Calabash testin perusrakenne..36

Kuvio 8. Calabash-testin ajo...37

Kuvio 9. Jenkins:in käyttöliittymän päänäkymä..40

Kuvio 10. Robotium-testien ajo Jenkinsillä...44

Kuvio 11. Robolectricin tulokset Jenkinsin testinäkymässä..46

Kuvio 12. FreeNest 1.4. version uudistettu ilme..48

KÄSITTEET

Aktiviteetti – Activity. Android-sovelluksen perusosanen, joka on yksittäinen asia, jonka käyttäjä voi tehdä.

Alfatestaus – Käyttäjätesti sovelluksen tai järjestelmän toimittajan tiloissa.

Android – Googlen omistama yksi suurimmista mobiilikäyttöjärjestelmistä tällä hetkellä.

Ant – Javalla kirjoitettu työkalu, jolla voidaan kääntää, ajaa ja testata Java-pohjaisia sovelluksia. Ant:ia voidaan käyttää myös muiden kuin Java-pohjaisten sovellusten kääntämiseen.

APK – Android application package file. Tiedostomuoto jolla asennetaan ja jaetaan Android-alustan sovelluksia.

Beetatestaus – Alfatestauksen jälkeinen vaihe jossa isompi joukko tavallisia käyttäjiä testaa sovellusta tai järjestelmää.

Debuggeri – Yleinen nimitys työvälineelle, jolla voidaan paikallistaa sovelluksessa olevia virheitä käymällä koodia läpi suorituksen aikana.

FreeNest – Ryhmäajatteluun pohjautuva projekti- ja sovelluskehitysalusta, jota kehittää SkyNest projekti.

GitHub – Verkkopohjainen ylläpitopaikka Git-versionhallintaa käyttäville projekteille. Avoimen lähdekoodin projekteille palvelu on ilmainen.

Harmaalaatikkotestaus – Yhdistelmä musta- ja lasilaatikkomenetelmiä.

Hyväksymistestivetoinen kehitys – Acceptance Test Driven Development (ATDD). Muistuttaa suurelta osin testivetoista kehitystä mutta sillä erolla, että asiakas tekee testien määrittelyt, joita vasten tehdään automaattisesti suorittavia testejä. Sitten kun varsinainen ohjelmakoodi läpäisee testit, asiakkaan vaatima toiminnallisuus on toteutettu.

Hyväksyntätestaus – ISTQB:n mukaan hyväksyntätestaus (acceptance testing) on testauksen vaihe, jossa todetaan täyttääkö järjestelmä asiakaan sille laatimat vaatimukset.

Instanssi – Olio-ohjelmoinnissa yksittäinen luokan edustaja. FreeNestin tapauksessa tarkoitetaan yksittäistä erillistä virtualisoitua ympäristöä, jossa FreeNest sijaitsee.

ISEB – ”ISEB on British Computer Society:n tietojenkäsittelyn tutkintolautakunta, ja sillä on kolmiportainen sertifiointijärjestelmä”

ISTQB – International Software Testing Qualifications Board. Järjestö, joka vastaa kansainvälisestä ohjelmistotestaajien sertifiointijärjestelmästä.

Jatkuva integraatio – (Continuous Integration). Ketterien menetelmien yhteydessä yleisesti käytetty prosessi jonka tarkoituksena on vähentää muutosten integroimisesta syntyvää työmäärää tekemällä siitä jatkuvaa tai ainakin hyvin useasti tapahtuvaa.

Jenkins – Avointa lähdekoodia oleva jatkuvan integraation toteutukseen kehitetty Java-pohjainen työkalu.

JUnit – JUnit on avoimen lähdekoodin yksikkötestauskirjasto Java-pohjaisten sovellusten testaukseen.

JVM – Java Virtual Machine. Nimensä mukaisesti Java-virtuaalikone jolla suoritetaan Java-tavukoodia

Lasilaatikkotestaus – Testausmenetelmä, jossa ohjelmakoodi tunnetaan ja tätä tietoa käytetään ohjelmiston testaamiseen.

Maven – Java-pohjaisten sovellusten kääntämiseen ja riippuvuuksien hallintaan kehitetty sovellus. Ominaisuuksiltaan hieman erikoistuneempi kuin Ant.

Mock-objekti – Korvikeolio, jolla voidaan simuloida erilaisia rajapintoja ja toisia olioita silloin, kun testataan jotain toista luokkaa tai oliota.

Mustalaatikkotestaus – Testausmenetelmä, jossa ohjelmakoodin toimivuutta tarkastellaan sille annettujen syötteiden ja niitten vastineiden oikeellisuuden perusteella.

Robot Framework – Nokia Siemens Networks:in kehittämä testausautomaatio sovelluskehys hyväksyntätestaukseen ja hyväksymistestivetoiseen kehitykseen.

Ruby – Dynaaminen avoimen lähdekoodin ohjelmointikieli, jonka tavoitteena on yksinkertaisuus ja tuottavuus.

Testitapaus – Syötearvojen, suorituksen esiehtojen, odotettujen tulosten ja suorituksen jälkiehtojen muodostama kokonaisuus, joka on muodostettu tiettyä testauksen

kohdetta varten.

Testivetoinen kehitys – Test Driven Development (TDD). Kehitystapa jossa sovelluksen tekeminen aloitetaan sillä että luodaan ensin testitapauksia, joita vasten tehdään sitten suoritettava ohjelmakoodi, joka läpäisee testit.

Roskien kerääjä – Garbage collector. Automaattinen muistinhallintamekanismi jolla pyritään vapauttamaan muistia, mitä sovellus ei enää tule tarvitsemaan.

SkyNest – Projekti jonka puitteissa ylläpidetään ja kehitetään FreeNestiä. Projekti on käynnissä Jyväskylän Ammattikorkeakoulussa.

1. TYÖN LÄHTÖKOHDAT

1.1. Tavoitteet

Opinnäytetyössä tavoitteena oli käydä läpi sovellustestauksen eri vaihtoehtoja ja miten sovellustestausta voidaan tehdä Android-sovelluksille. Tarkoituksena oli tutkia, millaisia vaihtoehtoja Android-sovellusten testaamiseen on olemassa tällä hetkellä ja millaisia keinoja testien automatisoinnille on olemassa. Läpikäydyistä vaihtoehdoista arvioidaan niiden soveltuvuutta ja helppokäyttöisyyttä päivittäiseen sovelluskehitykseen. Lopullisena tavoitteena oli tutkia, onko mikään olemassa olevista vaihtoehdoista sellainen, mitä pystyttäisiin hyödyntämään FreeNest-alustassa saumattomasti osana muita työkaluja. Päällimmäisenä kysymyksenä oli, voidaanko kattaa Android-sovelluskehityksen koko elinkaari aina suunnittelusta testaukseen kautta julkaisuun FreeNest-ympäristöä hyödyntäen.

Opinnäytetyön alkuosa koostuu teoriaosuudesta, jossa käydään läpi ohjelmistotestauksen peruskäsitteet ja käytänteet, sisällyttäen hieman historiallisia taustoja. Näitä asioita on käsitelty luvussa kaksi. Toinen osa koostuu käytännön tutkimustyöstä ja sen tulosten läpikäymisestä. Nämä asiat käsitellään luvuissa kolme ja viisi. Opinnäytetyön loppuosassa läpikäydään tiivistetysti yhteenvetona se, miten työ on saavuttanut sille asetetut tavoitteet, sekä suositukset siitä miten Android-sovelluksia voitaisiin parhaiten testata FreeNest-ympäristössä. Loppuosa sisältää myös pohdintaosion, jossa käyn läpi omia ajatuksiani koko opinnäytetyöstä ja mahdollisista parannusehdotuksista.

1.2. Työnantajan taustat

Opinnäytetyön aihe ja taustat liittyvät Jyväskylän ammattikorkeakoulussa tällä hetkellä olevaan SkyNest-projektiin. Projekti on yksi monista projekteista, jotka saavat rahoitustaan Tekes:in Cloud Software Finland -projektista. Cloud Software Finland on

nelivuotinen projekti, jonka tarkoituksena on nostaa suomalainen pilvipalveluosaaminen maailman kärkeen. Idea opinnäytetyön aiheeseen tuli siitä, että FreeNest-ympäristölle on ollut ajatuksen asteella tarkoitus saada oma Android-sovellus. Sovelluksen tekeminen ei ole tällä hetkellä SkyNest-projektissa korkeimmalla prioriteetilla, mutta ehkä sekin tulee ajankohtaiseksi jossain vaiheessa. Sitten kun sovelluksen tekeminen tulee ajankohtaiseksi, niin toivottavasti tästä tutkimuksesta on hyötyä jossain määrin kehityksen aikana.

1.3. Android-käyttöjärjestelmä

Android on Googlen omistama avoimeen lähdekoodiin perustuva Linux-pohjainen mobiilikäyttöjärjestelmä. Se on asennettuna satoihin miljooniin mobiililaitteisiin melkein 200 eri maassa, jonka käyttäjäkunta kasvaa miljoonalla ihmisellä joka päivä. (Android Developers 2013b.) Pääasiallisena ohjelmointikielenä Android-sovelluksissa käytetään Java-kieltä, mutta on myös mahdollista käyttää C- ja C++-kieliä Android Native Development Kit:in avulla.

2. SOVELLUSTESTAUS YLEISESTI

2.1. Johdanto

Mitä sovellustestaus oikeastaan on ja miksi sitä tehdään? Seuraavan kappaleen aikana kerrotaan yleisesti sovellustestauksesta ja käydään läpi muutamia peruskysymyksiä. Miksi sovelluksia ylipäätään testataan? Mitä sovelluksista yleensä testataan? Miten sovelluksia testataan? Kappaleen lopussa käydään vielä hieman läpi sovellustestauksen standardeja ja historiallisia taustoja.

2.2. Miksi testataan?

Suurissa projekteissa iso osa ohjelmiston kustannuksista tulee itse ohjelmiston testauksesta, jos se päätetään hoitaa asianmukaisesti. Nämä kustannukset olisivat silloin normaalisti 20 – 40 prosenttiyksikön luokkaa koko projektin budjetista. (Niittyvirta 2012.) Sovelluksissa, joissa vaaditaan suurta luotettavuutta tai sovellus on toiminnaltaan kriittisessä ympäristössä, sovellusten testaukseen saattaa kulua huomattavasti suurempi osa budjetista verrattuna tavallisiin projekteihin. Tällaisia suuren luotettavuusvaatimuksen projekteja ovat esimerkiksi erilaiset satelliittien ja avaruusluotainten ohjelmistot.

Esimerkiksi Saturn V kantorakettien ja Gemini avaruusohjelman vaatimissa sovelluksissa kehitysajan testauskustannukset olivat 45 %:n luokkaa. Samaan aikaan itse ohjelmointi muodosti vain 20 – 25 % kokonaiskustannuksista. Loppuosa kustannuksista muodostui ohjelmistojen suunnittelusta. (Wolverton, 1974.)

Sovellustestaus on yleisesti ollut hyvin paljon ihmistyötä vaativaa eikä automatisointi ole ollut mahdollista. Tästä syystä sovelluksen testauksen automatisointi on tärkeä toimenpide, jolla voidaan karsia sovelluksen kehittämisen aikaisia kustannuksia. Sovelluksen testauksen automatisointi helpottaa myös tulevaa testausta ja auttaa huomaamaan, jos sovelluksen kehityksen aikana tapahtuu regressiota.

Taulukko 1. Virheiden aiheuttamat korjauskustannukset projektin eri vaiheissa (Taina 1999.)

Projektin vaihe	Suhteellinen hinta
Määrittely	Perushinta
Suunnittelu	3-6 -kertainen
Koodaus	10-kertainen
Testaus	15-40-kertainen
Kenttätestaus	30-70-kertainen
Ylläpito	40-1000-kertainen

Vaikka testauksella voidaan karsia sovelluksen kehittämisen aikaisia kustannuksia, myös ohjelmistoissa olevien virheiden korjaaminen jälkikäteen tulee erittäin kalliiksi sovelluksen käyttäjille. Kustannuksia aiheuttavat itse virheen korjaaminen, mutta myös käyttäjien kautta aiheutuu kustannuksia työajan menetyksinä ja asiakastuen takia. Jos taas kyse on jostain myytävästä tuotemerkistä, niin silloin asiakkaiden saama huono mielikuva sovelluksesta voi ajaa tulevat ja nykyiset käyttäjät jonnekin muualle. Esimerkiksi tuotteiden asiakasarvioita on aina vain enemmissä määrin erilaisissa palveluissa ja verkkokaupoissa.

Yhdysvaltalaisen National Institute of Standards and Technologyn vuonna 2002 julkaisemassa tutkimuksessa käytiin läpi ohjelmistovirheistä aiheutuvia kokonaiskustannuksia yhteiskunnalle eri osa-alueittain. Tutkimuksessa kävi ilmi, että puutteellisesta testauksesta aiheutuu vuodessa jopa 60 miljardin dollarin välittömät ja välilliset kustannukset, joista voitaisiin välttää jopa kolmannes pudottamalla sovelluksissa esiintyvien virheiden määrää 50 prosentilla.

(National Institute of Standards and Technology 2002, 174.)

2.3. Mitä testataan?

Tiukasti ajateltuna kaikki pitäisi testata. Mitä suurempi osa sovelluksen koodista on testattu, sitä suuremmalla todennäköisyydellä säästetään projektin kokonaiskustannuksia. Kun tarkastellaan sitä, mitä pitäisi testata Android-sovelluksista, se eroaa paljon siitä mitä esimerkiksi testattaisiin tavallisessa Java-pohjaisessa sovelluksessa. (Torres Milano 2011, 11-12.)

Aktiviteetin elinkaari

Android-sovelluksen käyttäjän käyttökokemukseen vaikuttavat eniten aktiviteetin elämänkaaren metodit. Niiden testaamiseen tulisi kiinnittää erityistä huomiota, sillä ne vastaavat siitä, miten tietoa käsitellään eri tilojen välillä. (Torres Milano 2011, 11-12.)

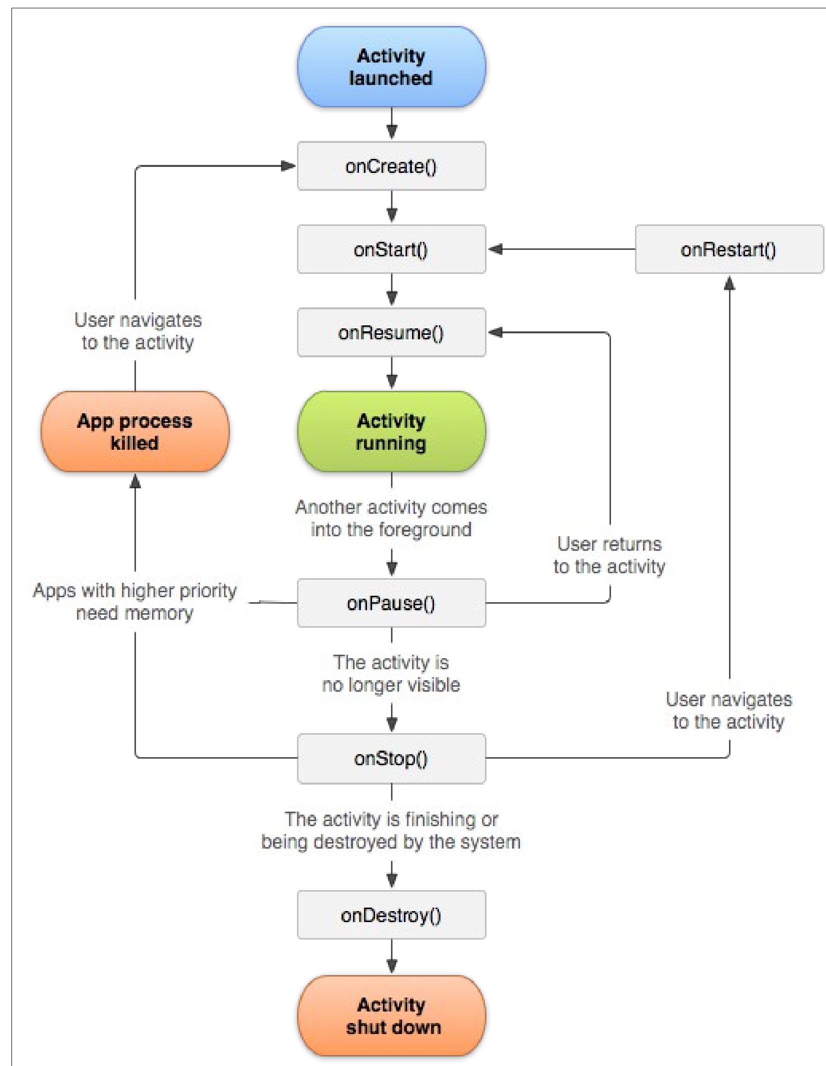
Android-sovelluksen aktiviteetin elämänkaaren tapahtumat voidaan jakaa neljään eri päätilaan:

- Ensimmäisessä tilassa sovellus on näkyvissä näytöllä ollen samalla aktiviteettipinon päällimmäisenä. Sovellus on tällöin aktiivinen (active) tai käynnissä (running).
- Toisessa tasossa aktiviteetti on menettänyt huomion (focus), mutta on silti näkyvissä. Tällainen tilanne tulee yleensä sellaisessa tilanteessa, missä jokin ei koko ruutua täyttävä tai läpinäkyvä aktiviteetti on edellä mainitun aktiviteetin päällä. Tässä tilassa aktiviteetti on keskeytynyt (paused), se on elossa ja säilyttää kaikki tietonsa ja on yhteydessä ikkunointikäsittelyihin. Silti tämä aktiviteetti saatetaan lopettaa äärimmäisissä tilanteissa, joissa muistia on erittäin vähän.
- Kun aktiviteetti on kokonaan poissa näkyvistä jonkin toisen aktiviteetin alla, silloin aktiviteetin tila on pysähtynyt (stopped). Pysähtyneessä tilassa olevat aktiviteetit säilyttävät tietonsa, mutta yleensä järjestelmä lopettaa pysähty-

neet aktiviteetit muistin käydessä vähiin.

- Silloin kun aktiviteetti on joko keskeytynyt tai pysähtynyt, järjestelmä voi tällöin vapauttaa muistia joko lopettamalla aktiviteetin tai pyytämällä aktiviteettia lopettamaan toimintansa. Jos lopetettuun aktiviteettiin palataan niin silloin sen sisältö ja edellinen tila täytyy palauttaa kokonaan uudestaan.

Kuviossa 1 on esitetty aktiviteetin elinkaari. Suorakulmiot esittävät eri metodeja, joita voidaan käyttää aktiviteetin tilojen tutkimiseen ja käsittelyyn eri siirtymävaiheissa. Ovaalin muotoiset kuviot taas edustavat aktiviteetin erilaisia tiloja.



Kuvio 1. Aktiviteetin elinkaari ja siirtymävaiheiden metodit (Android Developers 2013a.)

Kuviosta 1 nähdään myös kuinka seitsemän eri metodia voidaan jakaa karkeasti kolmeen ryhmään, joita olisi hyvä tarkkailla aktiviteetin elinkaaren aikana (Android Developers 2013a):

- Yhden aktiviteetin koko elinkaari sijoittuu onCreate() ja onDestroy()-metodien väliin. Aktiviteetin onCreate() metodia kutsutaan vain kerran ja silloin alustetaan kaikki globaalit muuttujat ja resurssit, joita muut aktiviteetin osat käyttävät. Tässä metodissa voidaan myös palauttaa aktiviteetin edellinen tila, jos se on tallennettu, ennen kuin se lopetettiin. Ennen aktiviteetin lopettamista kutsutaan onDestroy() metodia jonka aikana vapautetaan loput sen varaamat resurssit.
- Aktiviteetin näkyvä elinkaari alkaa onStart()-metodista ja loppuu onStop()-metodiin. Tänä aikana aktiviteetti on näkyvissä, muttei välttämättä päällimmäisenä. Molempia metodeja voidaan kutsua useita kertoja kun aktiviteetin näkyvyys muuttuu. onRestart()-metodia kutsutaan aina silloin kun aktiviteetti palaa näkyviin ja tätä seuraa aina onStart()-metodi.
- Etualan elinkaari sisältää sen ajan kun aktiviteetti on etualalla ja yhteydessä käyttäjään. Tähän tilaan voidaan tulla ja poistua useita kertoja tiheään tahtiin. Siksi olisikin tärkeää pitää tästä tilasta huolehtivat onResume()- ja onPause()-metodit keveinä.

Tiedostojärjestelmän operaatiot

Silloin kun Android-sovelluksessa käsitellään esimerkiksi kameran ottamia kuvatiedostoja tai tallennetaan tietoja paikalliseen tietokantaan, silloin tulee ottaa huomioon erilaiset tiedostojärjestelmään liittyvät operaatiot. Muistikortille tallennettaessa tulee tarkistaa, onko muistikortti kirjoitussuojattu tai poissa käytöstä. Tietokantaan kirjoitettaessa tulisi testata, että tauluista luetaan ja kirjoitetaan tietoja oikealla tavalla. Nämä testit tulisi pyrkiä suorittamaan eristetysti ilman vaatimuksia erillisiin tietokantoihin tai tiedostoihin. (Torres Milano 2011, 11-12.)

Tämä vaatimus voidaan tietokantojen kohdalla saavuttaa siten, että tarvittavista tietokantaliittymistä tehdään mock-objekteja, joilla simuloidaan käytössä olevaa tietokantaa. Tiedostojen kohdalla voidaan käyttää menetelmää, jossa luodaan testitiedosto oikean tiedoston rinnalle käyttämällä etuliitteitä. Esimerkiksi testi.teksitiedosto.txt, silloin kun oikea tiedosto on tekstitiedosto.txt. Tällöin voidaan testitapauksissa käyttää RenamingDelagateContext-luokkaa, jolla voidaan viitata tiedostoihin erilaisella etuliitteellä kuin alkuperäisiin viitattiin. Näin voidaan testata tiedostoihin liittyviä toimintoja kontrolloiduilla testitiedostoilla. (Torres Milano 2011, 181-184.)

Fyysiset ominaisuudet

Ennen sovelluksen julkaisemista tulisi ottaa huomioon laaja skaala puhelimia, joihin sovellus mahdollisesti asennetaan. Jos sovellus käyttää jotain tiettyjä puhelimelta vaadittavia ominaisuuksia, niin nämä tulisi testata, tai ainakin varautua siihen että niiden puute käsitellään oikein. Tämä siitäkin huolimatta, että sovellukselle voidaan määritellä tiettyjä rajoituksia, millaisiin puhelimiin se voidaan asentaa. (Torres Milano 2011, 11-12.)

Torres Milano (2011, 12-13) on kirjassaan listannut seuraavat asiat, joita kannattaa ottaa testauksen aikana huomioon:

- Verkon käyttö ja käytössä olevat nopeudet (GPRS, 3G, WLAN)
- Erilaiset näyttökoot ja -resoluutiot
- Erilaisten sensorien olemassaolo (Kompassi, kiihtyvyysanturi, NFC, GPS)
- Erilaiset näppäimistöt tai muut syötelaitteet
- Ulkoinen muistikortti ja sen tila

Listassa olevia asioita voi onneksi kohtuullisen kattavasti testata käyttämällä Android-emulaattoria erilaisilla asetuksilla. Lopullinen käyttäjillä suoritettu beetatestaus antaa kuitenkin parhaan kuvan siitä, miten sovellus todellisuudessa käyttäytyy erilaisilla laitteistokokoonpanoilla. (Torres Milano 2011, 13.)

2.4. Miten testataan?

2.4.1. Yleisesti

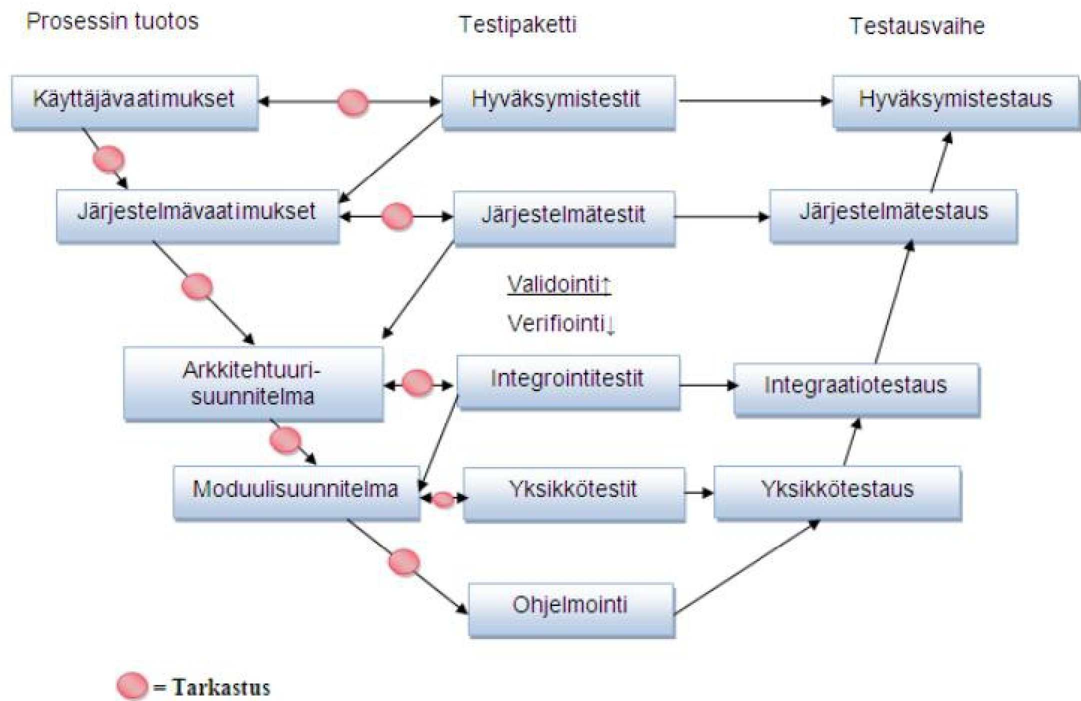
Ohjelmistotestauksen pääasiallisena tavoitteena on varmistua siitä, että toteutettu sovellus vastaa sille asetettuja vaatimuksia. Tämän tavoitteen saavuttamiseksi tulisi käyttää erilaisia menetelmiä sovelluskehityksen eri vaiheissa säännöllisin väliajoin. (Sainio 2010, 32.)

Ohjelmistotuotannossa tulee esille kaksi pääkysymystä, joihin ohjelmistotestauksen tulisi antaa vastauksia. Ensimmäinen kysymys kuuluu: ”Rakennammeko ohjelmistoa oikein?”. Tämä kysymys liittyy sovelluksen verifiointiin eli sovelluksen alemman tason toiminnan todentamiseen. Toinen kysymys taas kuuluu: ”Rakennammeko oikeanlaista ohjelmistoa?”. Tähän kysymykseen vastaamalla taas pyritään validoimaan sovelluksen toiminnallisuus korkeammalla tasolla. (Sainio 2010, 32.)

ISTQB on laatinut verifioinnille ja validoinnille omat määritelmänsä. Verifioinnin määritelmä: ”Määrättyjen vaatimusten täyttymisen vahvistaminen kokeellisesti ja objektiivisen todistusaineiston avulla.” Vastaavasti validoinnin määritelmä kuuluu: ”Määrättyä käyttöä varten tai sovellukselle asetettujen vaatimusten täyttymisen vahvistaminen kokeellisesti ja objektiivisen todistusaineiston avulla.” (ISTQB 2007.)

Kuviosta 2 nähdään verifioinnin ja validoinnin jakautuminen sovelluskehityksen elinkaaren eri vaiheisiin. Samalla tavalla kuin sovelluksen suunnittelussa ja toteutuksessa päästään aina vain tarkemmalla tasolle, samalla lailla myös testit ja testausme-

netelmät kulkevat karkeammasta tasosta tarkempaan.



Kuvio 2. Verifiointi, validointi ja testauksen tasot (Sainio 2010, 33.)

2.4.2. Testauksen tasot

Ohjelmistotestauksessa testauksen eri tasot jaotellaan usein kolmeen eri kategoriaan, jossa siirrytään aina tarkkuustasolta seuraavalle. (Guide to the Software Engineering Body of Knowledge 2004.) Kuviossa 2 kuvattiin verifioinnin ja validoinnin sijoittumista sovellustestauksen eri vaiheisiin. Samasta kuviosta nähdään myös testauksen tasojen sijoittuminen V-malliin.

Yksikkötestaus

V-mallissa alimmalla hierarkiatasolla sijaitsee yksikkötestaus. Käytännössä yksikkötestaus (unit testing) on pienimpien järkevien kokonaisuuksien testausta. Yksikkötestauk-

sesta voidaan myös käyttää nimitystä komponenttitestaus tai moduulitestaus. Yksikön määritelmä riippuu suuresti siitä, mitä kontekstia vasten sitä tarkastellaan. Esimerkiksi olio-ohjelmoinnissa testattava yksikkö voi koostua yksittäisestä luokasta, oliosta tai vaikkapa metodista. (Sainio 2010, 33-34.)

Integraatiotestaus

Integraatiotestaus sijaitsee V-mallissa toiseksi alimmaisella portaalla. Integraatiotestauksen (Integration testing) on tarkoitus todentaa, että tehdyt ohjelmiston osaset eli komponentit toimivat toistensa kanssa yhteen. Tässä vaiheessa ei enää tulisi todeta virheitä, jotka aiheutuvat itse komponenteista, koska nämä virheet pyritään eliminoimaan yksikkötestausvaiheessa. Integraatiotestien suunnittelussa tulee kuitenkin olla tarkkana sen suhteen että ei tehdä päällekkäistä työtä ja testata samoja asioita kuin yksikkötestauksen puolella on tehty. Oleellista on myös huomata, että integraatiotestaukseen ei tule sekoittaa ulkopuolisten rajapintojen testausta, sillä tämä tehdään järjestelmätestauksen aikana. (Sainio 2010, 37-38.)

Integraatiotestaukseen on useita erilaisia lähestymistapoja riippuen siitä, miten sen haluaa toteuttaa. Ehkä yleisin tapa, jolla tätä testausta on suoritettu, on ns. Big Bang -testaus, jossa ohjelmiston kaikki komponentit yhdistetään kerralla toisiinsa ja toivotaan, että kaikki sujuu mahdollisimman sujuvasti. Testauksen kannalta tämä ei ole kaikista optimaalisin vaihtoehto, sillä tästä on useita haittapuolia. Silloin kun tällaista lähestymistapaa käytetään, sovellus ja sen eri osa-alueet ovat jo hyvin pitkälle valmiita, ja tällöin testaukselle jää hyvin rajallisesti aikaa. Toinen esille tuleva ongelma on virheiden syy- ja seuraussuhteiden vaikea hahmottaminen. Isoa järjestelmää testatessa ei voida olla täysin varmoja siitä, mistä ohjelmiston osasta virhe loppujen lopuksi kumpuaa jos kaikki komponentit on yhdistetty kerralla. (Sainio 2010, 38-41.)

Parempi ja suositeltu testausmenetelmä integraatiotestaukseen on ns. jatkuvan integroinnin testausmenetelmä. Jatkuva integraatio pohjautuu siihen ajatukseen että kun ohjelmiston eri osien integraatiota testataan mahdollisimman usein, niin silloin tarvitsee integroida vain pieni osa kerrallaan. Samaan aikaan ajettavilla regressiotes-

teillä voidaan myös todentaa, onko uusi komponentti, tai siihen tehdyt muutokset, aiheuttaneet muissa aiemmin toimineissa osissa virheitä. (Sainio 2010, 41.)

Jatkuvaa integraatiota on myös kuvailtu seuraavasti: Yksinkertaisimmillaan se voi olla uusimpien lähdekoodien hakemista sekä kääntämistä, jonka jälkeen sovellus on valmis jaettavaksi. Toisessa ääripäässä taas sovellukselle ajetaan kaikki mahdolliset testit, aina palveluiden asentamisesta lähtien, dokumentaation päivittämiseen ja sovelluksen julkaisuun projektin nettisivuilla. (Puomala & Tolvanen 2011.)

Järjestelmätestaus

ISTQB:n määrittelemän testaussanaston mukaan järjestelmätestaus, toiselta nimeltään myös systeemitestaus (system testing), määritellään seuraavasti: ”Testaus, jolla varmistetaan, että integroitu järjestelmä täyttää sille asetetut vaatimukset” (ISTQB 2007). Koska järjestelmätestaus sijoittuu V-mallissa toiseksi ylimmäiselle tasolle, niin tässä vaiheessa löytyvät virheet ovat yleensä tasoltaan kriittisiä, ja niiden korjaus tulee yleensä kalliiksi. (Sainio 2010, 42.)

Lähteistä ja tapauksesta riippuen järjestelmätestaus on hyvinkin lähellä hyväksyntätestausta, minkä vuoksi rajan vetäminen selkeästi näiden kahden välille on hankalaa. Jos järjestelmätestausta pidetään omana tasonaan, kuten kuviossa 2 on kuvattu, silloin testataan järjestelmävaatimusten toteutumista. Tällöin hyväksyntätestaus kattaa sen, miten sovellus tai järjestelmä toteuttaa käyttäjävaatimukset. (Sainio 2010, 41-42.) Esimerkiksi sovelluksen pitää tukea Android API-versiota 8 tai sovelluksen tulee suoriutua sadan yhtäaikaisen käyttäjän kuormasta.

Järjestelmätestaus on tässä tapauksessa ehkä kaikista laajin neljästä eri V-mallin portaasta. Se pitää sisällään laajan kirjon erilaisia testaustapoja, muun muassa kuormitustestauksen, tietoturvatestauksen, suorituskykytestauksen, asennustestauksen, ylläpidettävyystestauksen, yhteensopivuustestauksen ja muutamia muita erilaisia testejä. Mahdollista on, että järjestelmätestaukseen kuuluu myös testaus toisia järjestelmiä vasten, ja miten integraatio niiden välillä toimii. (Sainio 2010, 42.)

Järjestelmätestauksen laaja-alaisuudesta johtuen se on luultavasti kaikista väärinymmärretyin testauksen taso. Silloin kun tähän testausvaiheeseen siirrytään, iso osa toiminnallisista virheistä on jo löydetty. Tämä tarkoittaa sitä, että sovellusta on hyvin hankala testata, jollei ole määriteltynä hyvin selkeitä ja mitattavissa olevia tavoitteita, mitkä sovelluksen pitää täyttää. (Sainio 2010, 42.) Tällainen konkreettinen mitattavissa oleva asia voisi olla Android-sovelluksessa se, että kun käyttäjä valitsee jonkin toiminnon, odotusaika ei saisi ylittää tiettyä aikarajaa. Tietokantakeskeisessä sovelluksessa voitaisiin mitata tietokannan suorittamien kyselyiden käyttämää aikaa, ja vaatia tiettyä aikarajaa suoritukselle.

Hyväksyntätestaus

Neljäntenä yleisenä testauksen tasona on myös pidetty hyväksyntätestausta (Software Testing Fundamentals 2012). Hyväksyntätestaus sijoittuu V-mallissa korkeimmalle tasolle ja toimii käyttäjävaatimusten todentajana. Tarkoituksena on tutkia sitä, täyttääkö testattava sovellus tai järjestelmä asiakkaan sille asettamat vaatimukset. Hyväksyntätestaukseen voidaan sisällyttää myös käyttöohjeiden ja muun dokumentaation hyväksyntä, ellei sitä ole jo aikaisemmassa vaiheessa tehty. (Sainio 2010, 50.)

Käytännössä järjestelmätestauksessa ja hyväksyntätestauksessa käydään läpi ja testataan osittain samoja asioita. Suurin ero näiden kahden testausvaiheen välillä on se, että hyväksyntätestauksen suorittaa asiakas. Lähteistä riippuen ns. alfa- ja beetatestaus voivat olla osa hyväksyntätestausta, tai sitten ne voivat tulla vasta hyväksyntätestauksen jälkeen. (Sainio 2010, 50) Tavallisesti isommissa yrityksissä projektin hyväksyntätestaus etenee ensin muutamalle avainkäyttäjälle, joiden kautta tulee korjausehdotuksia ja palautetta siitä, täyttääkö sovellus sille asetetut vaatimukset. Seuraavassa vaiheessa sovellus voidaan vielä asteittain julkaista todellisille loppukäyttäjille siten, että laajempi käyttöönotto tapahtuu portaittaisesti.

2.4.3. Testauksen lähestymistavat

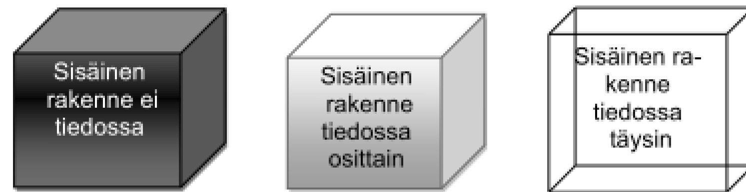
Eri lähestymistavat jaotellaan testausmielessä yleensä staattisiin ja dynaamisiin testaustapoihin. Staattisina testaustapoina pidetään yleensä erilaisia koodin katselmointeja ja tarkistuksia, joita suoritetaan ilman, että itse koodia suoritetaan. Voidaan esimerkiksi tarkistaa, että koodin syntaksi on oikein kirjoitettu ja logiikkaehdot ovat halutun kaltaisia. Tällainen tapahtuu yleensä silloin kun joku kirjoittaa koodia. On myös mahdollista, että koodia katselmoidaan aina pienissä ryhmissä erillisissä katselmoinneissa. (Sainio 2010, 54.)

Staattiset testauksen menetelmät on hyvä ottaa käyttöön jo suunnittelu- ja määrittelydokumenttien aloittamisesta lähtien. Silloin kun joku huomaa jo projektin alkuvaiheessa mahdollisen virheen, sen korjaaminen tulee erittäin edulliseksi verrattuna siihen, että olisi ehditty jo aloittaa varsinaisen ohjelmakoodin kirjoittamista. (Sainio 2010, 54.)

Dynaamisessa testauksessa taas itse ohjelmaa suoritetaan ja ajetaan läpi joko erilaisilla testitapauksilla tai sovelluskehitystyökalun debuggerilla. Tarkoituksena on käydä sovellusta läpi systemaattisesti, jotta virheitä löytyisi. Dynaaminen testaus on jaettavissa ei-toiminnalliseen ja toiminnalliseen testaukseen. (Sainio 2010, 60.)

Laatikkomalli

Yleisin tapa lähestyä toiminnallista testausta on ns. laatikkomalli. Laatikkomallissa testaus voidaan jakaa kolmeen erilaiseen lähestymistapaan. Normaali jaottelu menee mustalaatikkotestaukseen, lasilaatikkotestaukseen sekä harmaalaatikkotestaukseen.



Kuvio 3. Laatikkomallien erilaisuus havainnollistettuna (Sainio 2010, 61.)

Mustalaatikkotestauksessa ei tunneta testattavan osan sisäistä toimintaa tai rakennetta. Testaajan näkökulmasta ne ovat tuntemattomia. Testit rakennetaankin tässä tapauksessa sen pohjalta, miten sovelluksen on määritelty toimivan ja tiedetään, millainen lopputulos on odotettavissa tietyillä arvoilla. Vaikka lopputulosta ei varmuudella tiedettäisi, voidaan tehdä oletuksia halutusta lopputuloksesta. Esimerkkinä voidaan testata yksinkertaista metodia, jonka tarkoitus on laskea lukuja yhteen. Tälle metodille, annetaan parametreina numerot 2 ja 3. Testaaja tietää määrittelyn perusteella että metodin pitäisi palauttaa 5. Jos tulokset ovat yhteneväisiä haluttujen tulosten kanssa, niin silloin metodi todennäköisesti toimii oikein. Tämän tyyppisesti testaamalla voidaan havaita sovelluksen eri osien logiikassa olevia virheitä ja epäjohdonmukaisuuksia syötteiden käsittelyssä. (OAMK 2006) Todellisuudessa testattavat metodit ja funktiot eivät ole näin suoraviivaisia, vaan niiden sisäinen logiikka voi olla hyvinkin monimutkainen ja vaiheikas.

Lasilaatikkotestauksessa taas tunnetaan testattavan osion toiminta ja koodi. Lasilaatikkotestauksen tavoitteena on saada tehtyä tarpeeksi kattavat testit, joilla koko testattavan osion koodi tulisi käytyä läpi. Toisin kun mustalaatikkotestauksessa, lasilaatikkotestauksessa ei riitä pelkästään se että syötteet antavat oikean tuloksen, vaan myös tulos on saatava oikealla tavalla. (OAMK 2006)

Kolmantena ryhmänä laatikkomalleissa on harmaalaatikkotestaus, joka on yhdistelmä aikaisempia lasilaatikkomenetelmiä. Esimerkiksi web-sovellusten tapauksessa mustalaatikkotestaus ei välttämättä paljasta virheitä lähdekooditasolla, kun taas lasilaatik-

kotestaus saattaa jättää huomiotta käyttöjärjestelmästä tulevat riskit ja virheet. Harmaalaatikkotestauksessa tarkastellaan, niin käyttäjälle näkyvää lopputulosta, kuin taustalla olevien teknisten järjestelmien toimintaa ja tietoja. Näin saadaan paljastettua mahdolliset virheet tiedonkulussa tai järjestelmien yhteensopivuudessa. (OAMK 2006) Tässä ollaan ns. ”harmaalla alueella” kahden eri menetelmän välissä.

Koodikattavuus

Aikaisemmassa kappaleessa käsiteltiin erilaisia laatikkomenetelmiä, joista lasilaatikkotestaus oli kaikista läpinäkyvin. Tällainen testien ja lähdekoodin tietämys mahdollistavat testien kattavuuden mittaamisen. Erilaisia koodikattavuuslukuja (code coverage) vertailemalla voidaan arvioida, kuinka laajasti sovellus on testattu.

Erilaisia kattavuuslukuja ovat muun muassa lausekattavuus (statement coverage), päätöskattavuus (decision coverage), ehtokattavuus (condition coverage) sekä haarakattavuus (branch coverage). Lausekattavuus kertoo kuinka monta prosenttia sovelluksen lauseista, pienimmistä jakamattomista sovelluksen osasista, on testattuna ja läpikäytynä. (Sainio 2010, 62-63.)

Päätöskattavuus kuvaa sitä miten suuri osa päätöksistä, esimerkiksi if-lauseen tosi ja epätosi vaihtoehdoista, on käyty läpi. Silloin kun päätöskattavuus saavuttaa 100 %, tarkoittaa se myös sitä että lausekattavuus on 100 %. Toisin päin tämä ei kuitenkaan toimi. Vaikka kaikki sovelluksen lauseet olisi ajettu läpi, niin kaikkia päätöksiä ei välttämättä ole käyty läpi. (Sainio 2010, 62-63.)

Ehtokattavuus kuvastaa kuinka monta prosenttia ehtojen tuloksista on testien aikana käyty läpi. Tarkoituksena on käydä jokaisen päätöslauseen tulokset läpi totena ja epätotena. Haarakattavuudella ilmaistaan kuinka iso osuus erilaisista sovelluksen haaroista on käyty läpi. Haara on tässä tapauksessa yksittäinen lohko koodia, joka voidaan suorittaa valintatilanteen jälkeen. Haarautuminen tapahtuu yleensä switch-case-rakenteessa tai if-then-else-rakenteessa. Silloin kun testin haarakattavuus on

100 % niin silloin myös päätöskattavuus ja lausekattavuus ovat 100 % (Sainio 2010, 63.)

2.5. Testauksen standardit ja historia

Ammattimaisen ohjelmistotestauksen juuret juontavat 1970-luvulle, jolloin Yhdysvalloissa järjestettiin ensimmäinen ohjelmistotestausta käsittelevä konferenssi. Tuosta konferenssista sai lähtölaukauksen koko nykyinen testaamisen ammattimaistuminen nykyisine standardeineen ja sertifikaatteineen. Ensimmäisten virallisten standardien valmistelu aloitettiin vuonna 1979. Samana vuonna julkaistiin ensimmäinen ohjelmistotestausta käsittelevä kirja, The Art of Software Testing. (Sainio 2010, 17.)

Kun ensimmäistä standardia oltiin valmisteltu nelisen vuotta, se julkaistiin vuonna 1983 nimellä: ”829-1983 IEEE Standard For Software Test Documentation”. Tämä standardi kuvasi ja määritteli kahdeksan dokumenttia, sekä niiden sisällön. Standardista on myöhemmin tullut kaksi uutta versiota. Ensimmäisen uudistettu versio julkaistiin vuonna 1998 ja toinen versio vuosikymmen myöhemmin, vuonna 2008. Ensimmäinen standardi koskien yksikkötestausta julkaistiin vuonna 1987 nimellä: ”1008-1987 IEEE Standard For Software Unit Testing”. (Sainio 2010, 17.)

Testaus onkin nykypäivänä saanut tunnustuksen omana erikoisalanaan ohjelmistotuotannossa. Aiheesta on kirjoitettu huomattava määrä erilaisia kirjoja ja ohjelmistotestaus on otettu osaksi ohjelmistotekniikan opetusta yhtenä osa-alueena.

Koska testaus on laaja osa-alue ohjelmistotuotannossa ja vaatii omaa erikoisosaamista, niin sille on pyritty luomaan oma sertifiointijärjestelmä. Vuonna 2002 perustettiin International Software Testing Qualifications Board, lyhyemmin ISTQB. Sen tehtävänä oli tuolloin kehittää kansainvälinen ja yhtenäinen sertifiointijärjestelmä ohjelmistotestaajille. Vuonna 2006 ISTQB:n oma sertifikaatti yhdistettiin ISEB:n, omiin sertifikaat-

teihin. Sertifikaattitasoja on siten kolme, joista viimeisellä tasolla voidaan valita kahdesta eri suuntautumisvaihtoehdosta. Ensimmäinen vaihtoehto on testianalyysiin ja toinen testien hallintaan suuntautuva. (Sainio 2010, 17-18.)

3. ANDROID-SOVELLUSTEN TESTAUS

3.1. Johdanto alemman tason testauksiin

Aikaisemmassa luvussa käsiteltiin muutamia testauksen perusajatuksia ja lähtökohtia. Nämä toimivat pohjana sille, miten Android-sovelluksia voisi testata, ja millaisia menetelmiä voitaisiin käyttää. Tässä luvussa tullaan käymään läpi osa olemassa olevista vaihtoehdoista, sekä myöhemmissä luvuissa selvitetään niiden hyvät ja huonot puolet kehittäjän näkökulmasta.

Päätin tutkimusten aikana karkeasti jakaa käytettävät työkalut ja menetelmät kahteen kategoriaan. Alemman tason testausmenetelmiin, joissa käsitellään itse koodia ja sen toimintaa. Sekä korkeamman tason testaukseen, jonka tarkoituksena on todentaa enemmän käyttäjälle näkyvää toiminnallisuutta. Tämä sinällään ei ole paras mahdollinen tapa jakaa käsiteltäviä työkaluja, sillä harva käyttöliittymän testaukseen käytetty työkalu on pelkästään puhtaasti käyttöliittymän testaukseen sopiva. Yleisempää on, että useamman työkalun yhdistelmällä saadaan aikaan kattavampi kokonaisuus, jolla molemmat puolet hoituvat paremmin kuin pelkästään yhtä työkalua käyttämällä.

Käytettävät työkalut ja menetelmät valitsin sen perusteella, onko niillä vielä aktiivisesti kehitystä tai vaihtoehtoisesti laaja käyttäjäpohja. Tämä helpottaa ongelmatilanteiden ratkaisua, kun voi tukeutua joko kehittäjiin tai käyttäjä-yhteisöön.

3.2. Alemman tason testaus

3.2.1. Yleistä yksikkötestauksesta

Aikaisemmassa luvussa käytiin läpi V-malli ja sen eri tasot, sekä verifiointi ja validointi. Tässä kappaleessa käydään läpi, miten alemman tason testausta voidaan toteuttaa Andorid-sovelluksia tehtäessä.

Yksikkötestauksessa ja muussakin testauksessa sovelluksen toiminta pyritään varmistamaan asettamalla väittämiä (assertions), joita vasten todellista toiminnallisuutta mitataan. Android-sovelluksia testatessa voidaan käyttää olemassa olevia väittämiä jotka tulevat JUnit-kirjaston mukana. Lisäksi voidaan hyödyntää myös varta vasten Android-sovellusten testaukseen tarkoitettuja väittämiä jotka tulevat Googlen ja muiden tekijöiden tarjoamien kirjastojen mukana.

3.2.1. Junit

Historiaa

JUnit on avoimen lähdekoodin yksikkötestauskirjasto, jonka historia juontaa juurensa 90-luvun alkuun. Varhaisimmat versiot näkivät päivänvalon 1992 ja muutamaa vuotta myöhemmin julkaistiin ensimmäinen versio nimellä Sunit. Myöhemmässä vaiheessa JUnit on käynyt yhden suuren rakennemuutoksen, jolloin siirryttiin metodiajattelusta huomautustyyppiseen testien merkitsemiseen. (Software Engineering Radio 2010)

JUnit3

Android-kehitysympäristön mukana tulee oletuksena mahdollisuus käyttää yksikkötestaukseen JUnit3-testikirjastoa. Nämä testit voidaan ajaa automatisoidusti kehitysympäristössä siten, että niiden tulokset saadaan takaisin kehitysympäristöön ilman erillisiä ohjelmia.

Mukana tuleviin väittämiin kuuluu erilaisia yksinkertaisia vertailuita, kuten assertEquals tai assertNotSame, joilla voidaan testata samankaltaisuutta. Lisäksi on monia muita väittämiä jotka antavat pohjan totuusvertailuille. Näiden perusväittämien lisäksi tarjolla on Googlen tekemä ViewAsserts ja MoreAsserts -luokat, jotka antavat mahdollisuuden tarkastella totuusehtoja näkymistä ja olioista hieman tarkemmalla tasolla. (Torres Milano 2011, 50 – 56.)

JUnit3 testitapaukset sisältävät hieman vähemmän vaihtoehtoja testitapausten alustamiseen verrattuna uudempaan JUnit4-kirjastoon. Käytössä ovat vain setUp()- ja tearDown() -metodit ennen jokaisen testin suorittamista. Siitäkin huolimatta nämä metodit tarjoavat riittävän toiminnallisuuden, jotta voidaan kirjoittaa tarpeeksi monipuolisia testejä. Seuraavassa kohdassa käsitellään hieman tarkemmin JUnit testiluokan ominaisuuksia.

JUnit4

Liitteessä 1 on kuvattuna JUni4-testin perusrunko. JUnit4- ja JUnit3-kirjaston eroavaisuudet testien rakenteessa ovat selkeitä, mutta testien perusperiaatteet pysyvät samoina. Sanwald (2013) on kiteyttänyt näiden kahden kirjaston eroavaisuudet hyvin lyhyesti:

- Java5-tyylisten huomautusten @before ja @after käyttö setUp() ja tearDown() -metodien sijaan.
- Testiluokan ei enää tarvitse periytyä TestCase-luokasta. Tämä on hyvä muutos, koska Javassa ei ole moniperintää.
- @Test-huomautus korvaa vanhan nimeämiskäytännön, jossa testitapauksen nimen täytyi alkaa test-sanalla. Esimerkkinä testOneThing().
- Import-määrittelyn muuttuminen staattiseksi väittämien kohdalla.
- JUnit 'Teoria'-väittämät. Tarkoituksena on luoda testitapauksia jotka pystyvät käsittelemään oletuksia eivätkä pelkästään staattisesti määriteltyjä arvoja.

Vaikka liitteessä käytetäänkin vielä JUnit3-tyylistä nimeämiskäytäntöä, se ei ole pakollista, mutta helpottaa testien lukemista ja ymmärtämistä. Neljäs versio tarjoaa lisäksi uudet @BeforeClass- ja @AfterClass -huomautukset. Niiden avulla voidaan luoda testitapauksia, jossa osa alustuksista suoritetaan vain kerran ja osa aina testitapausten jälkeen @Before- ja @After -huomautuksilla.

Näitä JUnit4 ympäristön testejä ja niiden perustoiminnallisuuksia ei voi suoraan ajaa Android-sovelluksissa mukana tulleilla työkaluilla, mutta kiertoteitse niiden käyttö on mahdollista JUnit4Android-kirjastoprojektia käyttäen. Tosin tästä saatu hyöty jää rajalliseksi siinä vaiheessa, kun testejä haluttaisiin automatisoida, joten se ei pitemmän päälle ole paras mahdollinen ratkaisu.

Huomioitavaa Android-testauksessa

Kun ollaan suunnittelemassa Android-sovellukselle yksikkötestaukseen soveltuvia testitapauksia, tulee ottaa huomioon muutamia alustasta riippuvaisia seikkoja. Yleisesti hyvänä käytänteenä voidaan pitää sitä, että jokaiseen yksikkötestiin sisällytettäisiin yksi testi, jolla testattaisiin sitä, että testattava laite täyttää ennakkovaatimukset. (Torres Milano 2011, 14-16.) Niin Android-laitteilla kuin emulaattoreillakin testatessa ei voida aina tarkkaan tietää, täyttääkö se tietyt ehdot esimerkiksi kameran tai verkkolaitteiden suhteen. Silloin ennakkovaatimusten testaamisesta on hyötyä, vaikka ei voida taata, että testit suoritettaisiin halutussa järjestyksessä.

Toinen huomioitava seikka testitapauksia suunniteltaessa on niiden lukumäärä. Yhteen yksikkötestiin voidaan sisällyttää useita pienempiä testejä, jotka sitten ajetaan läpi jossain järjestyksessä. JUnit3 on suunniteltu siten, että se rakentaa kaikki testitapauksen instanssit yhdellä kertaa ja ajaa kaikki testit näille instansseille sen jälkeen. Jos testitapauksessa testataan jotain sellaista osa-aluetta, joka voi varata huomattavan määrän muistia, tulisi aina huolehtia siitä, että nämä resurssit vapautettaisiin eksplisiittisesti tearDown()-metodissa asettamalle niille null-arvo. (Torres Milano

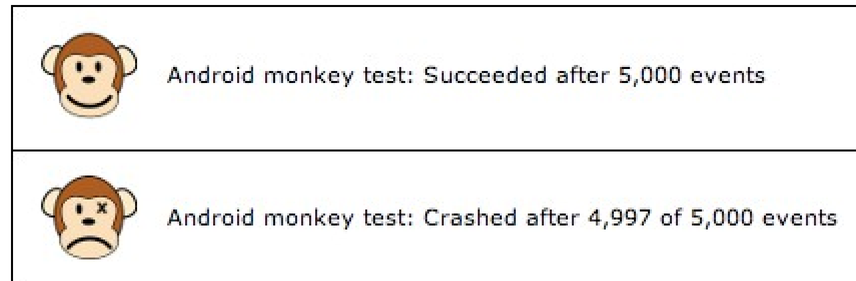
2011, 14 -16.)

Muutoin testi voi virheellisesti antaa tulokseksi epäonnistumisen pelkästään siitä syystä, että testattavasta laitteesta loppuu muisti kesken, kun sitä on varattu liikaa testitapausten käyttöön. Kun varatut resurssit vapautetaan asianmukaisesti tearDown()-metodissa ja testien koko suhteutetaan oikein, silloin automaattinen roskien keräys voi hoitaa muistin vapauttamisen hallitusti.

3.2.2. Monkey

Monkey on Googlen kehittämä komentorivipohjainen työkalu, jolla voidaan lähettää satunnaisia syötteitä testattavalle ohjelmalle. Tällä tavalla voidaan luoda esimerkiksi testi, joka painelee kaikkia painikkeita ja muuttaa puhelimen orientaatiota satunnaisesti viisi tuhatta kertaa. Tällaisen ”rasitustestin” läpäiseminen antaa hyviä suuntaviivoja sille, miten sovellus selviää tavallisen käyttäjän käytössä. Jos sovellus läpäisee tämän testin, silloin on todennäköistä, että aktiviteetin syötteiden käsittely ja erilaiset asynkroniset taustatoiminnot käsittelevät virheet sovellusta kaatamatta.

Monkey-työkalua ei siis tule pitää minään täsmätyökaluna, jolla voidaan testata sovelluksen toimintaa tarkasti, saatika sitten varmistaa, että sovellus käsittelee erilaiset ääri- ja erikoistapaukset oikein. Työkalu antaa hyvän lisän muiden testien rinnalle ja auttaa sovelluskehittäjää löytämään ilmeisimmät toiminnalliset virheet sovelluksesta. Monkey voidaan ajaa myös automatisoidusti Jenkinsin kanssa silloin, kun sovelluksesta on käännetty uusi versio. On myös mahdollista ajastaa stressitestit omaksi testikseen aina määräajoin tapahtuvaksi.



Kuvio 4. Monkey stressitestin tulos Jenkinssissä (Orr 2013)

3.2.3. Mockito

Mockito on avoimen lähdekoodin 'Mocking framework', jolla pystytään helposti testaamaan olioita ja liittymiä eri rajapintoihin. Koska testien tulisi olla mahdollisimman paljon riippumattomia erilaisista rajapinnoista, voidaan Mockiton avulla luoda tarvittavia korvikeolioita eri tilanteisiin.

Mockiton käyttämisen perusteet ovat hyvin yksinkertaiset ja testi koostuu yleensä seuraavista kolmesta vaiheesta.

1. Luodaan yksi tai useampi Mock-objekti, jotka sitten alustetaan ja niille määritellään halutut ominaisuudet.
2. Testataan toiminnallisuus, joka halutaan todentaa.
3. Validoidaan että tapahtui juurikin se, mitä piti, eikä mitään ylimääräistä tapahtunut.

Toimintaperiaate väittämien kanssa on hyvin samanlainen kuin JUnit-testeillä. Liitteessä 2 on kuvattu muutama luokka ja rajapinta, joita voidaan testata käyttäen Mockitoa. Alla olevassa esimerkissä määritellään varastolle tietty toimintamalli kun kutsutaan varaston mock-objektia. Tässä tapauksessa varastokyselyyn vastataan myöntävästi. Tämän jälkeen varastoa vasten tehdään tilaus. Tilaus todennetaan verify()-metodilla ja varmistetaan, että tuotteet todellakin poistuvat varastosta.

```
Warehouse mockWarehouse = mock(Warehouse.class);
when(mockWarehouse.hasInventory("Talisker", 50)).thenReturn(true);

Order order = new Order("Talisker", 50);
order.fill(mockWarehouse);

assertTrue(order.isFilled());
verify(mockWarehouse).remove("Talisker", 50);
```

Tällä tavalla testatessa tarkistetaan tiluksen toimintaa ja sen toimivuutta varaston kanssa. Tärkeintä on kuitenkin huomata mock-objektilla testaamisen ja esimerkiksi JUnitilla testaamisen ero. Perinteisellä yksikkötestauksella alustettaisiin varasto-olio ja tilaus-olio samalla lailla, kuin tässä tapauksessa Order-olio. Sitten tarpeellisia metode- ja kutsumalla tehtäisiin samat varasto-operaatiot. Tällä tavalla selvitetään että olioiden **tilat** muuttuivat odotusten mukaan.

Mock-objekteja käytettäessä ei testata vain sitä, että olioiden tilat muuttuvat oikein, vaan myös sitä, että olioita kutsuttiin oikealla tavalla. Näin varmistetaan olioiden **toiminnallinen** virheettömyys. Kun mock-objekteille on määritelty tietyt odotukset tulevista kutuista ja toimintamalli, miten niihin tulisi reagoida, voidaan muutokset todeta testeissä. (Fowler 2007.)

Mockito Androidissa

Mockito toimii Android-sovellusten testauksessa hyvin. Testiprojektin riippuvuuksiin täytyy vain muistaa lisätä mockito-all.jar ja sen lisäksi dexmaker, jotta Mockito saa- daan mukaan yksikkötestaukseen projektin käännösvaiheessa.

3.3. Android sovellusten käyttöliittymätestaus

3.3.1. Yleistä käyttöliittymistä

Android-sovellusten käyttöliittymän testaus on haastavaa monella tapaa katsottuna.

Android-puhelimen käyttöjärjestelmä on alustana varsin hajanainen, eikä yhtä selkeää versiota ole, jota vasten sovellus kannattaisi testata. Taulukosta 2 nähdään, että suurin osa käyttöjärjestelmän versioista jakautuu joko hieman vanhempiin tai sitten aivan uusimpiin versioihin. Kaikista vanhimmat versiot alkavat olla jo marginaalisia API-versiosta 10 alaspäin ja tableteissa alussa ollut API-versiolla 12 – 13 on jäänyt uudemman Jelly Bean version jalkoihin.

Taulukko 2. Eri Android versioiden jakauma aikavälillä 21.01.13 – 04.02.13 (Platform Versions 2013)

Versio	Koodinimi	API taso	Jakauma
1.6	Donut	4	0,2%
2.1	Eclair	7	2,2%
2.2	Froyo	8	8,1%
2.3 - 2.3.2	Gingerbread	9	0,2%
2.3.3 – 2.3.7		10	45,4%
3.1	Honeycomb	12	0,3%
3.2		13	1,0%
4.0.3 - 4.0.4	Ice Cream Sandwitch	15	29,0%
4.1	Jelly Bean	16	12,2%
4.2		17	1,4%

Tämän lisäksi sovelluksen tekijä on otettava huomioon että näyttötarkkuudet ja kieliversiot tuotat oman lisänsä testaukseen. Googlen antaman kehitysympäristön mukana tulee mahdollisuus luoda lähes minkälaisia kombinaatioita niin käyttöjärjestelmässä, näytön koossa kuin laitteen ominaisuuksissakin. Tästä seuraa se, että jos haluaa testata mahdollisimman laajalla laitekannalla, niin testien ajaminen käsin on hyvin haastavaa.

Luvuissa 3.3.2 – 3.3.4 on käyty läpi muutamia esimerkkejä käyttöliittymän testaukseen soveltuvista kirjastoista, vaikkakin esimerkiksi Robolectric- ja Robotium-kirjastoja voi hyvin käyttää myös matalamman tason yksikkötestaukseen.

3.3.2. Robotium

Robotium on Android sovellusten mustalaatikkotestaukseen kehitetty ohjelmistokehys. Sen tarkoituksena on auttaa kehittäjiä kirjoittamaan tehokkaita automatisoituja testejä helposti. Sitä voidaan käyttää niin testitapauksia aina funktionaalisista testausta, kuin systeemitestausta ja hyväksyntätestausta varten. (User scenario testing for Android 2013.)

Käytön aloittamisen helppoudessa Robotium on aloittavalle Android kehittäjälle ehkä kaikista helpoin. Jos koneella on jo ennestään asennettuna kaikki Android kehitykseen tarvittavat työkalut, niin riittää pelkän robotium-solo.x.x.x.jar tiedoston lisääminen testiprojektiin.

Liitteessä 3 on kuvattu yksinkertainen testitapaus, jolla avataan asetusvalikko. Koska pohjalla käytetään vain pelkkää jar-kirjastoa, silloin testien rakenne on samanlainen kuin JUnit3 pohjaisissa testeissä. Esimerkkitapauksessa setUp()-metodissa kutsutaan Robotiumin solo-luokkaa, joka sisältää tarvittavan toiminnallisuuden kutsua käyttöliittymän komponentteja.

Itse testissä käsketään vain painaa käyttöliittymässä olevaa painiketta, jotta saadaan avattua sovelluksen asetusnäkymä. Tämän jälkeen tarkistetaan, onko avautuva aktiviteetti sellainen kuin sen pitäisi olla.

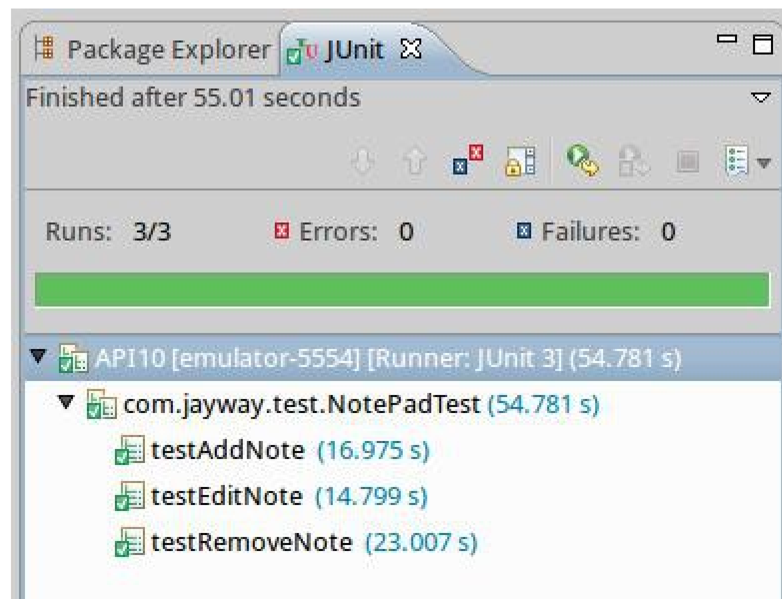
```
solo.clickOnButton("Application settings");

assertTrue("Wrong activity",
solo.waitForActivity("VeripalveluSettings"));
```

Kun testitapaukset on saatu kirjoitettua, niiden ajaminen onnistuu helposti valitsemalla sovellukseen liittyvä testiprojekti ja valitsemalla ”Run As → Run as Android Junit test”. Tällä tavalla sovellus käännetään ja käynnistetään siten että testitulokset saadaan takaisin Eclipse-ympäristöön.

Testiprojekti on linkitettynä alkuperäiseen testattavaan projektiin siten, että se vaatii alkuperäisen sovelluksen toimiakseen. Testiprojektin käynnistyessä ympäristö kääntää ja asentaa emulaattorille tai puhelimelle uusimman version apk-tiedostosta, jota vasten testit suoritetaan. Lisävaihtoehtona Robotiumilla on kuitenkin mahdollista testata myös pelkästään apk-tiedostoa ilman lähdekoodia. Tällöin testiprojektin luominen on kuitenkin paljon monimutkaisempi prosessi.

Kuviossa 5 on esitetty kehittäjän sivuilta ladattavan testiprojektin ajon tulos. Testien suoritusta voi myös seurata reaaliaikaisesti, joko käynnissä olevan emulaattorin näytöltä, tai sitten fyysiseltä laitteelta.



Kuvio 5. Testien tulokset suorituksen jälkeen Eclipse-ympäristössä

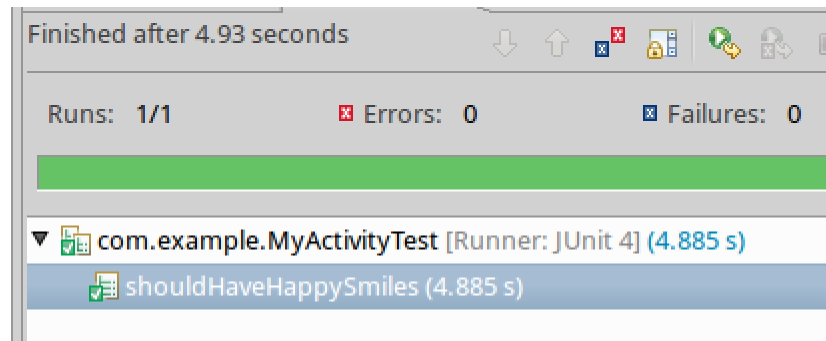
3.3.3. Robolectric

Robolectric eroaa Robotium ja Calabash testauskehyksistä siinä, että se ei aja sovelluksen koodia emulaattorissa tai laitteessa. Robolectric alustassa on otettu pääasialliseksi lähestymistavaksi se, että sillä olisi mahdollisimman helppo tehdä nopeasti ajettavia TDD testejä. Koska testit ajetaan paikallisella JVM alustalla ilman, että sovellusta tarvitsee pakata ja asentaa laitteeseen, nopeuttaa se testien ajoa huomattavasti. (Pivotal Labs 2013.)

Aikaisemmassa luvussa käsiteltiin yksikkötestauksen puolella Mockito testauskehystä, jolla testattiin olioita ja niiden rajapintoja. Robolectric toimii samaan tapaan hoitaen suurelta osin Android rajapintojen 'mockaamisen' korvaamalla metodien palautuskutsut. Robolectric luo Android luokista ”varjo-olioita” (Shadow objects), jotka jäljittelevät Android SDK -luokkien käytöstä. Käytössä on tällä hetkellä suurin osa sellaisista luokista, joita tarvitaan jokapäiväisessä testauksessa ja luokkien määrä kasvaa koko ajan. Tämän takia erillistä Android-emulaattoria ei tarvita.

Kuten aiemmasta kuviosta 5 voidaan havaita, menee Robotiumin testien suorittamiseen joskus hyvinkin pitkä aika. 55 sekuntia kestävä testiajo on huomattavan pitkä aika silloin, kun sen suorittamista odottaa paikallisella koneella. Tämä hitaus testien suorituksessa vaivaa etenkin emulaattorilla ajettavia testejä. Jos toistoja tulee useita kymmeniä päivän aikana, tällöin voidaan menettää huomattavan suuria määriä tehokasta työaikaa pelkästään testien odotteluun.

Kuviosta 6 nähdään, että testien suorittamiseen kulunut aika ei ole läheskään niin suuri kuin emulaattorin kautta testejä ajettaessa. Suoritusaika Robolectric-testeissä oli noin viisi sekuntia. Tuloksissa täytyy kuitenkin huomata se seikka, että ne eivät ole suoraan vertailukelpoisia. Melkein minuutin kestävä testiajo sisälsi useita ja monimutkaisempia testejä, joten todenmukaisempi vaikutus ei tietenkään ole kymmenkertainen suoritusteho Robotiumiin nähden.



Kuvio 6. Robolectric testin tulos

Tuloksiin vaikuttavat myös suurelta osin käytetty laitteisto ja sen käytössä olevat resurssit. Todenmukaisempi tehoetu testauksessa käytetyllä laitteistolla ja kyseisillä testitapauksilla on noin 3 – 5 kertainen. Tässä vaiheessa on kuitenkin hyvä muistaa se, että vaikka testaus tällä tavoin onkin huomattavasti nopeampaa, ei se silti vastaa täysin sitä, millaista on testata sovellusta emulaattorilla tai fyysisellä laitteella.

Robolectricin vaatimukset

Robolectric tarvitsee toimiakseen uudemman JUnit4-kirjaston, eikä tavallinen Android SDK:n mukana tuleva JUnit3 riitä. Kuitenkin Eclipsen mukana tulee vaadittavat kirjastot myös JUnit4-tyylisten testien tekemiseen. Tässä on kuitenkin vaatimuksena se, ettei testejä luoda Android-tyyppiseen projektiin, kuten tavallisesti tehdään, vaan testejä varten luodaan tavallinen Java-projekti. Testiprojektin luominen on selostettu tarkemmin liitteessä 5

Robolectricin syntaksista

Robolectric ei eroa suuresti muista testaamiseen käytetyistä testikehyksistä. Testien syntaksi noudattaa aikaisemmassa luvussa käsiteltyä JUnit4-kirjastoa, joten testien kirjoittaminen on hyvin suoraviivaista. Poikkeuksen tekee testitapauksen alkuun laitettava ”@RunWith(RobolectricTestRunner.class)” huomautus ja tarve alustaa varjo-olioita. Huomautuksen ansiosta Robolecricin testit voidaan ajaa tarpeellisten kirjastoluokkien kautta.

3.3.4. Calabash

Calabash on Ruby:llä kirjoitettu hyväksyntätestaukseen tarkoitettu testauskehys. Calabash pohjautuu Cucumber ympäristöön joka mahdollistaa testitapausten kuvaamisen ihmisläheisesti ja ymmärrettävässä muodossa (LessPainful 2013). Vaikka Calabashin taustalla toimii kaupallinen yritys, ovat sen lähdekoodit kaikkien vapaasti saatavilla ja kehitettävissä. Yritys tarjoaakin mahdollisuutta ajaa omien projektien testejä suuremmassa ympäristössä, jossa sovellus voidaan testata useita eri puhelinmalleja vasten. Testausympäristön asentaminen on selostettu tarkemmin liitteessä 7.

Testien rakenne voi koostua kolmesta perustoiminnosta joita voidaan suorittaa testattavalle sovellukselle:

- Eleet (gestures) (esim. painallus, pyyhkäisy, kääntö)
- Erilaiset väittämät (esim. ”Minun tulisi nähdä kirjautumispainike tai uloskirjautumispainike”)
- Kuvakaappaukset

Kuviossa 7 voidaan nähdä, että testit ovat perusrakenteeltaan hyvin selkeitä ja jokaisen tulkittavissa. Testien rakenne ja ulkoasu ovat hyvin pitkälle samanlainen kuin Cucumber testikehyksen kanssa. Esimerkissä kirjaudutaan sovelluksella sisään palveluun, johon vaaditaan käyttäjätunnuksen ja salasanan syöttämistä.

```
Feature: Login feature
  Scenario: As a valid user I can log into my app
    Given I am a valid user
    And I enter my username
    And I enter my password
    And I press "Login"
    Then I see "Welcome to coolest app ever"
```

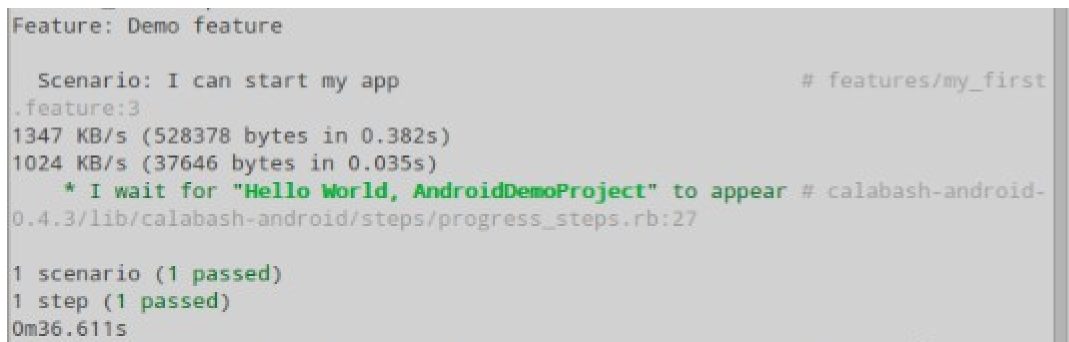
Kuvio 7. Calabash testin perusrakenne (Calabash android 2013)

Projektin rakenne

Testiprojektin luominen on Calabashin tapauksessa hyvin yksinkertaista. Tarvittava projektirunko saadaan aikaan kun luodaan tarvittava kansiorakenne ja tiedostot joko käsin tai sitten automaattisesti android-calabash työkalulla. Aikaisemmista työkaluista poiketen erillisen testiprojektin luomista ei tarvita, vaan testit sijaitsevat projektikansion alla sijaitsevassa calabash nimisessä kansiossa.

Itse testit ajetaan suorittamalla komentokehotteessa komento projektin calabash-kansion sisällä. Kuviossa 8 on esitettynä miltä testien tulokset näyttävät komentokehotteen kautta katsottuna, kun alla oleva komento ajetaan.

```
calabash-android run path/to/apk_file.apk
```



Kuvio 8. Calabash-testin ajo

4. ANDROID-TESTIEN AUTOMATISOIMINEN JA JATKUVA INTEGROINTI

4.1. Johdanto automatisoituihin testeihin

Testien automatisointi on ideana loistava. Pyritään siirtämään mahdollisimman paljon mekaanista toistoa vaativat toimenpiteet koneen suoritettavaksi ja näin säästetään aikaa sekä resursseja. Todellisuudessa asia ei ole aivan näin yksinkertainen, vaan tässäkin tapauksessa liika automatisointi on pahasta. Tähän onkin eräs testausta pitkään hoitanut henkilö sanonut seuraavaa:

> *Ainakin se, että automatisoida pitää harkiten, tai sillä ampuu itseään jalkaan. Mitä tahansa ei kannata automatisoida vaan vain stabiilit asiat / paljon toistoa vaativat asiat / yksikkötestaus mahdollisuuksien mukaan. Suurin harhaluulo on, että automatisointi vähentää testaustyön määrää - se pääasiassa vaan siirtää painopistettä toiseen kohtaan ja suorastaan räjäyttää työmäärän, jos automatisoidaan ilman harkintaa. Sen lisäksi testauksen automatisointi ei ole testausta, vaan ohjelmointia, jossa tulee ymmärtää testausta, eli kuka tahansa testaaja ei automaattisesti ole testauksen automatisointiin kykenevä henkilö vaan tarvitaan erikoistaitoja (tämä tuli omalla painavalla kokemuspohjalla). Ja sitten vielä asia, joka minua on askarruttanut tässä, että kuka testaa sen, että automaatiokoodi toimii oikein? (Sainio 2010, 120.)*

Automaatio ei poista manuaalisten testien tarvetta, eikä tarvetta testaajille, jotka osaavat analysoida testien tuloksia, eivätkä automatisoidut testit ole verrattavissa manuaalisiin testeihin. Automaatiota tuleekin käyttää manuaalisen testauksen jatkeena silloin, kun sen avulla voidaan tehdä jotain, mihin ei manuaalisesti pystyttäisi. (Sainio 2010, 120.)

Testauspalvelimet

Luvussa 2.4.2 käytiin läpi testauksen eri tasoja. Näistä yksi taso oli integraatiotestaus, johon liitettiin myös käsite jatkuva integrointi. Mitä keinoja jatkuvaan integrointiin on sitten käytössä? Sovellusten lähdekoodien ja testien automaattiseen ajamiseen on kehitetty erilaisia tehtävään sopivia palvelinsovelluksia.

Tarjolla on monia kaupallisella lisenssillä olevia, kuin avoimella lähdekoodillakin olevia ohjelmistoja. Ehkä tunnetumpia kaupallisia vaihtoehtoja ovat Microsoftin Team Foundation Server ja Atlassian Software Systemsin Bamboo. Vastaavasti avoimen lähdekoodin vaihtoehdoista tunnetuimpia Jenkins, Builbot ja Mozilla Foundationin kehittämä Tinderbox.

Tässä tutkimuksessa päätettiin keskittyä Jenkinsin mahdollisuuksien tutkimiseen, sillä sitä on hyödynnetty aikaisemmin FreeNest-ympäristössä. Toiseksi mahdolliseksi vaihtoehdoksi jatkuvaan integrointiin ja testaukseen päätettiin ottaa Nokia Siemens Networkin Robot Framework.

4.2. Jenkins

4.2.1. Jenkinsin yleiskuvaus

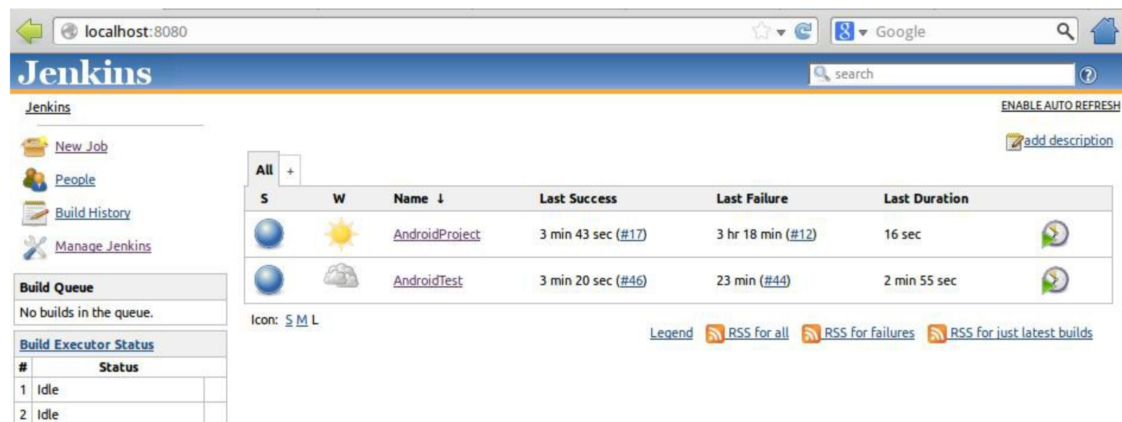
Alun perin Hudson nimisestä projektista eriytetty Jenkins on jatkuvaa integraatiota varten toteutettu palvelinsovellus. Sovelluksella voidaan toteuttaa projektien jatkuvaa integraatiota, niin pienissä kuin suurissakin projekteissa.

Jenkins on yksi suosituimmista avoimeen lähdekoodiin perustuvista jatkuvan integraation palvelimista. Itse pääprojektia kehitetään muutaman pääkehittäjän voimin aktiivisesti ja uusi versio julkaistaan aina viikon tai kahden välein. Nopean kehityshaaran lisäksi ylläpidetään vakaata pitkän tuen versiota, jonka päivittyy noin 3

kuukauden välein uuteen vakaaseen versioon.

Itse Jenkins on jo kohtuu monipuolinen ominaisuuksiltaan, mutta erittäin joustavaksi ja muovautuvaksi Jenkinsin tekee mahdollisuus käyttää satoja eri laajennoksia. Laajennoksilla on mahdollista saada tuki erilaisille versionhallintajärjestelmille ja testausmenetelmille. Jos jostain syystä ei löydä itselleen sopivaa laajennosta, niin sellaisen voi myös helposti tehdä itse Jenkinsin kattavaa avointa rajapintaa vasten.

Kuviossa 9 on Jenkinsin perusnäkymä silloin kun tullaan etusivulle. Etusivun näkymästä nähdään kullakin hetkellä olemassa olevat ”Tehtävät” (Jobs), joiden edellinen suoritustila on indikoitu värillisillä palloilla. Niiden avulla nähdään, miten viimekertainen ajo on onnistunut. Yleiskatsaus eri tehtäville nähdään taas vieressä olevasta sää-ikonista, joka kertoo säätilana viimeisimpien käännösten ja testien tulokset. Jos esimerkiksi käännös on jäänyt kesken virheeseen useampana kertana peräkkäin, säätila muuttuu aurinkoisesta myrskyisään.



Kuvio 9. Jenkins:in käyttöliittymän päänäkymä

Kuvion tapauksessa Jenkinsille on olemassa Android-projekti, ja Android-projektiin liittyvä testiprojekti, jotka molemmat ovat läpäisseet viimekertaisen ajon. Samalla myös nähdään aika, milloin kukin projekti on suoritettu onnistuneesti, ja milloin

viimeksi on tapahtunut epäonnistunut ajo. Suoritusajat ovat näkyvissä oikeassa laidassa, josta voidaan myös käynnistää ajo käsin ikonia painamalla.

4.2.2. Jenkins ja Android-sovellusten testaus

Aikaisemmissa luvuissa käsiteltiin, miten Android-sovelluksen testauksen pystyy toteuttamaan käyttämällä paikallisessa koneessa toimivia kirjastoja ja ympäristöjä. Jenkinsin monipuolisuuden ansiosta kaikkia aikaisemmin käsiteltyjä kirjastoja voidaan ajaa myös automaattisesti erillisellä palvelimella. Tämä tarjoaa sovellusta kehittäville ohjelmoijille ja testaajille käsityksen siitä, missä tilassa sovellus ja sen testaus on kullakin hetkellä.

Tarjolla on Jenkinsille suunniteltu ”Android Emulator”-niminen laajennos, joka osaa hoitaa tavallisimmat tehtävät emulaattoreiden käynnistämisen, luomisen, käyttämisen ja puuttuvien versioidenkin osalta. On erittäin kätevää, että laajennos osaa automaattisesti asentaa projektin vaatimat Android-versiot, jos sellaista ei ole valmiiksi asennettuna.

Se miten eri lähestymistavoilla toteutetut ympäristöt soveltuvat käytettäväksi Jenksinsin kanssa, on huomattavia eroja helppoudessa ja toimivuudessa. Toiset toimiva hyvin pienellä vaivalla, kun taas toiset vaativat enemmän asetusten säätämistä kohdilleen. Yleisesti ottaen lopullisen valinnan ratkaisee se, mitä järjestelmää kukin kehitystiimi haluaa käyttää. Luku 5 kertoo tarkemmin näistä havainnoista.

4.3. Robot Framework

4.3.1. Mikä on Robot Framework?

Nokia-Siemens-Networkin kehittämä Robot Framework on yleiskäyttöinen testausautomaatiokehys, jolla voidaan suorittaa hyväksyntätestausta ja testausvetoista hyväksyntätestausta (ATDD). Se sisältää helppokäyttöisen taulukkomuotoisen testidatasyntaksin, joka hyödyntää avainsanoja testien kirjoituksessa. Olemassa olevia ominaisuuksia voidaan laajentaa testikirjastoilla, joita voidaan kirjoittaa joko Pythonilla tai Javalla. Käyttäjällä on myös mahdollisuus luoda omia avainsanoja, jos niille on tarvetta. (Robot Framework 2013.)

4.3.2. Mahdollisuudet Android-sovellusten kanssa

Android-sovellusten testaus on tällä hetkellä hyvin pitkälle keskittynyt Jenkinsiin, kun testit halutaan suoritttaaa avoimen lähdekoodin työkaluilla. Tällä hetkellä Robot Frameworkille on tehty yksi testikirjasto, millä voidaan ajaa Android-sovelluksille tarkoitettuja testejä. AndroidLibrary nimellä kulkeva kirjasto käyttää hyödykseen Calabashia.

Kun Calabash on asennettu onnistuneesti kehitysympäristöön, voidaan Robot Framework ottaa käyttöön hyvin helposti. Sovellusten hallintaan voidaan käyttää Python-skriptien hallintaan tarkoitettua python-pip -sovellusta.

```
sudo apt-get install python-pip
sudo pip install robotframework
sudo pip install --upgrade robotframework-androidlibrary
```

Tämän jälkeen testit voidaan ajaa yksinkertaisella komennolla: pybot testi.txt. Valitettavasti lupaavalta vaikuttava alku ei tässä tapauksessa kantanut asennusta pidemmälle. Robot Framework-kirjastoa ylläpitävät kehittäjät eivät ole testanneet sitä uusimmalla Calabash-versiolla, joten testien ajaminen onnistuneesti ei yrityksistä huolimatta onnistunut.

Emulaattorin käynnistäminen ja testien ajamisen aloittaminen onnistuivat, mutta testitapaukset eivät jostain syystä palauttaneet muuta kuin negatiivisia tuloksia. Tämä saattoi johtua monesta eri tekijästä, mutta ongelmien syvällisempi tutkiminen jäi ajanpuutteen vuoksi kesken. Toivonkin että kirjaston kehittämistä jatkettaisiin ja se saataisiin toimimaan uudempien Calabash-versioiden kanssa joskus tulevaisuudessa.

5. KÄYTÄNNÖN TOTEUTUS

5.1. Robotiumin hyödyntäminen Jenkinsissä

Jenkins tarvitsee build.xml tiedoston kääntääkseen Android-projektit ja muodostaa tarvittavat apk-tiedostot. Build -tiedostot voidaan pääprojektille luoda komennolla ”android update project -n NAME -p PATH_TO_PROJECT”.

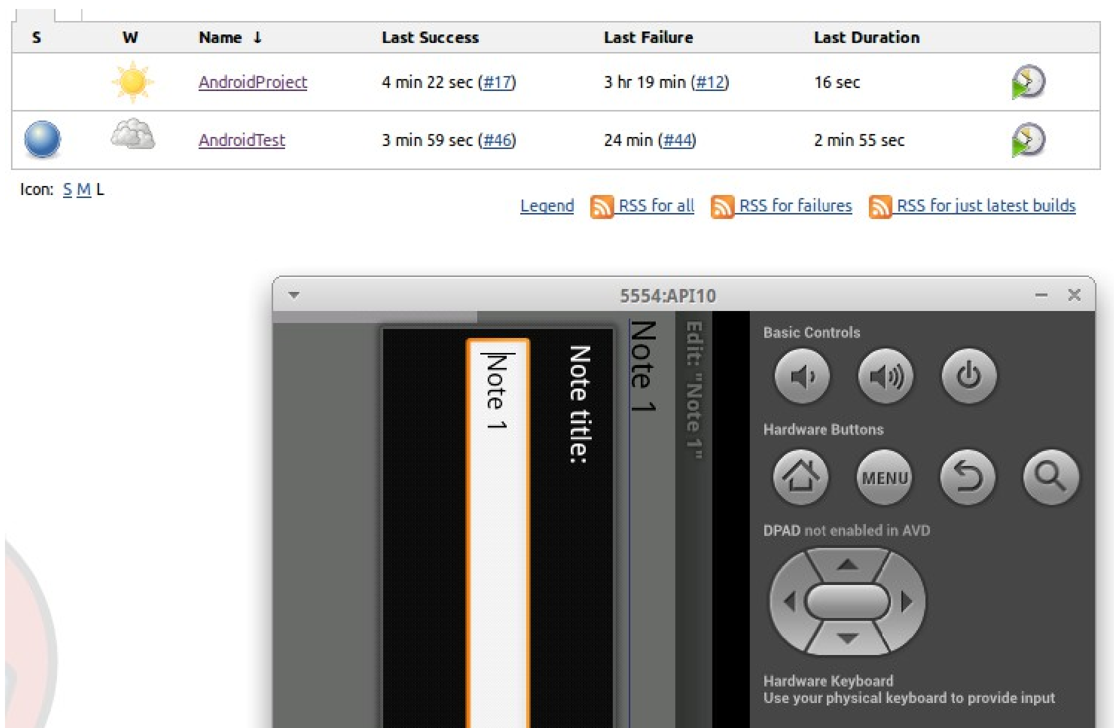
```
freenest@freenest-VirtualBox:~/android-sdk-linux/tools$ ./android update project -n NotePad -p /home/freenest/workspace/NotePad
```

Seuraavaksi luodaan testiprojektille build.xml tiedosto update test-project komennolla ja parametrina annetaan pääprojektin polku.

```
freenest@freenest-VirtualBox:~/android-sdk-linux/tools$ ./android update test-project -n NotePadTest -p /home/freenest/workspace/NotePadTest -m /home/freenest/workspace/NotePad
```

Kun molemmille projekteille on luotu oma kääntötiedosto niiden toimivuus kannattaa testata että Ant-kääntötyökalu, jota Jenkins käyttää, saa ajettua käännökset ongelmitta. Ajamalla molemmat ajotiedostot saadaan selvyys siihen, onnistuiko tiedostojen luonti. Jos halutaan testata pelkkää käännöstä niin silloin riittää komento ant clean debug. Jos halutaan testata sovelluksen asennusta samalla kertaa, voidaan lisätä komentoon ”install” parametri perään.

Kun ant-kääntäjän vaatimat build.xml tiedostot ovat valmiina, voidaan sovelluksen kääntäminen ja testien ajaminen hoitaa Jenkinsin kautta. Liitteessä 9 on kerrottu tarkemmin, millaisilla asetuksilla Robotium-projektia voidaan ajaa. Pääpiirteissään prosessi etenee siten, että ensin käännetään sovellus lähdekoodista ja jos se onnistuu, aloitetaan testien suorittaminen. Tapahtumaketjun aikana sovellus ja testisovellus asennetaan puhelimelle, sekä ajetaan määritellyt testit. Kuviossa 10 on esitys testiajosta näkyvää emulaattoria hyödyntäen.



Kuvio 10. Robotium-testien ajo Jenkinsillä

Vaikka Robotium soveltuu hyvin käytettäväksi paikallisella koneella, niin sen liittäminen osaksi Jenkins-palvelimen käännösrutiineja ei ollut aivan yksinkertaista. Matkan varrella tuli useita ongelmia, joista suurin osa oli enimmäkseen tiedostojen käyttöoikeuksista johtuvia.

Toinen vielä alkuperäisellä testaushetkellä ylitsepääsemätön ongelma oli saada Jenkins-palvelin käynnistämään Android-emulaattori. Android-sovelluksen ja testipakettien asentaminen emulaattoriin onnistui ongelmitta, kuten myöskin niiden ajaminen. Myöhemmin suoritetuissa testeissä ongelmia ei enää ilmennyt uudempia versioita käytettäessä.

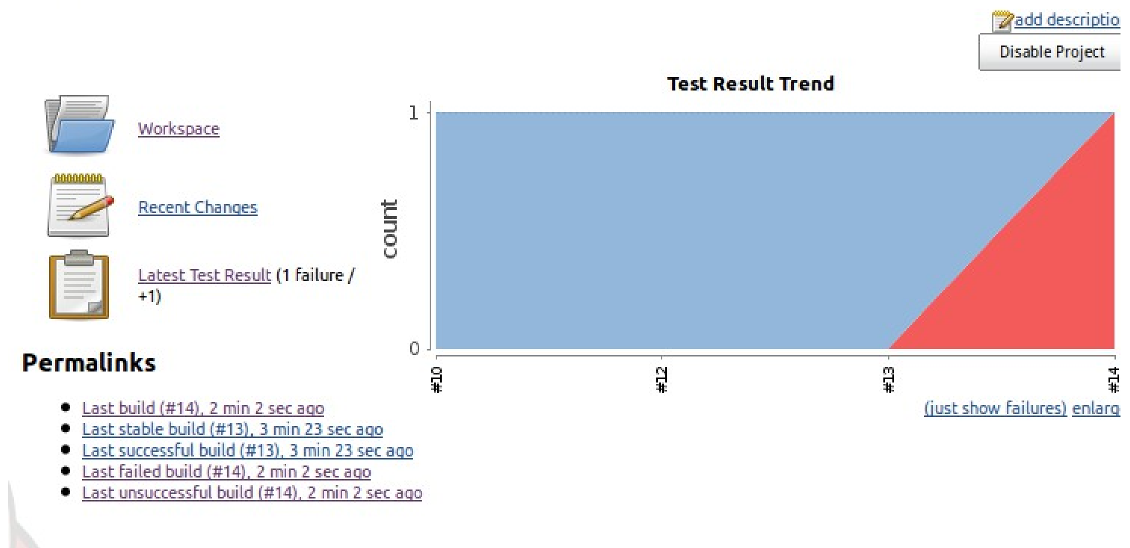
Vaikka Robotium-kirjasto itsessään on helppo ja monipuolinen testauskäyttöön, niin sen vaatima Android-emulaattori tuotti vaikeuksia automaatiota ajatellen. Onneksi kehittäjäyhteisö oli tiedostanut ongelman ja tuottaneet siihen sopivan korjauksen.

5.2. Robolectricin hyödyntäminen Jenkinsissä

Robolectricin liittäminen osaksi jatkuvaa integraatiota on huomattavasti yksinkertaisempaa kuin Robotiumin. Aikaisemmin vaikeuksia ja turhaa monimutkaisuutta Jenkins-määrityksiin aiheuttanut Android Emulator -laajennos on poissa. Emulaattorin puuttuminen tuo toki omat hyvät ja huonot puolensa testejä ajatellen. Tilannetta tarkasteltaessa palvelimen näkökulmasta voidaan todeta, että kuormitustaso on huomattavasti alhaisempi ja asennus yksinkertaisempaa.

Liitteessä 5 kuvataan lyhyesti sitä, miten Robolectricillä tehtäviä testejä varten luodaan tarvittavat projektit oikeine asetuksineen. Kun Robolectric on saatu liitettyä Android-testiprojektiin, vaatii testitulosten saaminen näkyville hieman enemmän työtä. Liitteessä 6 on määriteltynä Ant-työkalulle sopiva build-tiedosto. Tämän avulla

Ant osaa kääntää sovelluksen ja muodostaa tarvittavat xml-tiedostot testien tuloksista.



Kuvio 11. Robolectricin tulokset Jenkinsin testinäkymässä

Mutta Robolectricin kanssa testitapauksia ajettaessa ei voitu välttyä ongelmilta. Joissain tapauksissa Robolectric ei löytänyt testattavan sovelluksen manifest-tiedostoa ja ei suostunut ajamaan testejä. Osa käyttäjistä on raportoinut kyseisen ongelman Robolectricin GitHub-sivuilla, jossa lähdekoodit ovat saatavilla, ja tilanteeseen on luultavasti jo tämän työn julkaisun aikaan tehty tarvittavat korjaukset.

5.3. Calabash ja Jenkins

Kaikista aikaisemmin testatuista testauskehyksistä Calabash osoittautui kaikkein vähiten vaivaa vaativaksi. Ainoastaan siinä tapauksessa että asentamisesta syntyvää vaivaa ei lasketa mukaan. Kun Calabash on saatu asennettua, automatisointi ei vaadi Jenkinsin osalta monimutkaisia toimenpiteitä.

Riittää että projektille luodaan tarvittava build-tiedosto Android-SDK:n työkaluilla ja lisätään projekti versionhallintaan. Jenkinsille määritellään paikka lähdekoodien hakemiseen, jonka jälkeen suoritetaan lähdekoodille käännös. Testit suoritetaan samalla käskyllä kuin aikaisemminkin paikallisella koneella. Lisäksi tarvitaan vain suoritusparametriin ehto että halutaan ulos JUnit-muotoinen
testitulos /tmp/junit_out kansioon.

```
calabash-android run path/to/apk_file.apk --format junit --out /tmp/junit_out
```

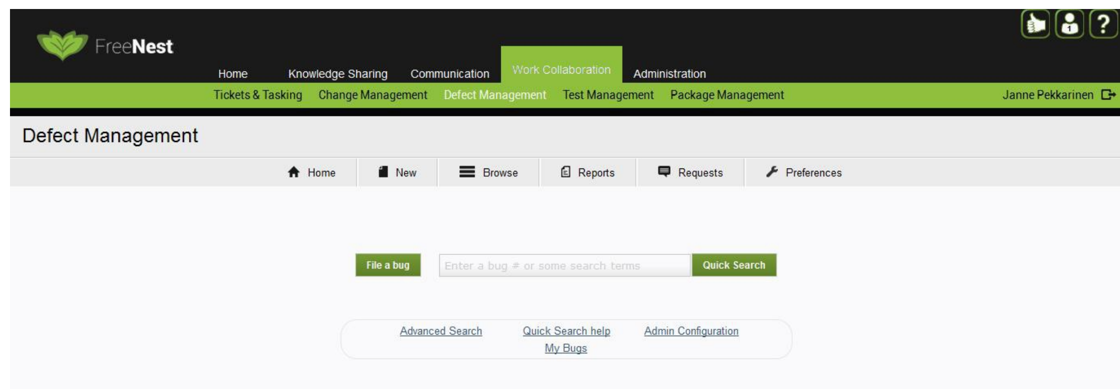
6. FREENEST JA ANDROID-KEHITYS

6.1. Mikä on FreeNest?

FreeNestin tarkoitus on yhdistää yhteen paikkaan useita eri avoimen lähdekoodin sovelluksia, joita yleisesti käytetään ohjelmistotuotannon eri vaiheissa ja toimia ”liimana” niiden välillä. Tavoitteena on ollut luoda ympäristö, joka on modulaarinen ja helposti laajennettavissa. Täten jokaista projektia varten voidaan luoda juuri sopiva ympäristö projektin tarpeisiin. (FreeNest 2013.)

Perinteiset tuotekehitysympäristöt on pyritty tarjoamaan aina keskitetysti ja useampi projekti sijaitsee samalla palvelimella ja samassa instanssissa. FreeNest eroaa tässä muista siinä, että se on suunniteltu helpoksi pystyttää ja jokaiseen instanssiin saadaan vain ne työkalut, joita tarvitaan. Tämä selkeyttää käyttöä, koska käyttäjillä ei ole käytössään turhia ominaisuuksia, ainoastaan sellaiset joista on heille oikeasti hyötyä. (FreeNest 2013.)

Kuviossa 12 on esitettynä FreeNestin uusin versio. Tällä hetkellä menossa on 1.4. -version kehityshaara, joka on nyt julkistettu Alpha-vaiheeseen. Aikaisempiin versioihin verrattuna ulkoasua ja käytettävyyttä on parannettu huomattavasti, mikä on tämän uuden version yksi päätavoitteista.



Kuvio 12. FreeNest 1.4. version uudistettu ilme

Seuraavissa kappaleissa 6.2 ja 6.3 käsitellään lyhyesti sitä, miten FreeNest soveltuu Android-sovellusten kehittämiseen ja testauksen automatisointiin.

6.2. Yleinen sovelluskehitys

FreeNest soveltuu hyvin Android-sovellusten kehittämiseen, koska ympäristö ei ota mitään kantaa siihen millaista projektia ollaan tekemässä. Kehitystiimillä on valittavanaan suuri määrä erilaisia työkaluja moniin eri tarpeisiin. Tärkeimpänä versionhallintajärjestelmä GIT tai SVN mahdollisuudella. Jenkins pystyy siten hoitamaan aina uusimpien lähdekoodien kääntämisen ja testien ajamisen uusinta versiota vasten.

Muita hyödyllisiä työkaluja yhteisen tiedon jakamiseen on Foswiki, joka on integroitu hyvin Bugzillan ja Trac:in kanssa. Näillä työkaluilla kehitystiimi voi ylläpitää listaa tehtävistä töistä ja sovelluksesta löytyvistä virheistä.

6.3. FreeNest yhdistettynä Jenkinsin kanssa

FreeNest-projektin pakettivarastosta ei vielä tätä työtä kirjoittaessa ollut saatavilla erikseen asennettavaa freenest-jenkins pakettia, josta Jenkins voitaisiin asentaa helposti. Asentaminen on kuitenkin kaiken kaikkiaan hyvin suoraviivaista ja integroiminen muuhun ympäristöön onnistuu hyvin pienellä vaivalla.

Asentaminen onnistuu helposti ja päivitykset pysyvät ajantasaisina kun asennus suoritetaan pakettienhallinnan kautta. Liitteessä 8 tämä prosessi on kuvattu ja asennuksen pitäisi onnistua ilman suurempia ongelmia.

Jos FreeNest-ympäristössä halutaan aja käännöksiä Ant-työkalulla, joudutaan se asentamaan erikseen, koska se ei tule oletuksena pakettien mukana. Asentamiseen riittää komentokehote:

```
sudo apt-get install ant
```

Testattujen testauskehysten toimivuus

Koska Jenkins on vain yksi mahdollinen lisätyökalu FreeNest-ympäristössä, niin aikaisemmin testattujen välineiden käyttämiseen ei ole mitään esteitä. Samat ongelmat ja rajoitukset tulevat ilmi myös FreeNestin puolella, eikä mitään erikoisia FreeNestistä johtuvia ongelmia syntynyt.

6.4. FreeNest Robot Frameworkin kanssa

Robot Framework itsessään on toimiva ja käytössä osana FreeNest-ympäristön testausta. Tämän vuoksi olisikin ollut hienoa yhdistää Android-sovellusten testaus samaan työkaluun jo olemassa olevan järjestelmän kanssa.

7. YHTEENVETO

7.1. Johdanto

Tarkoituksena on antaa suuntaa antavia viitteitä siitä, miten Android-sovellusten testaus ja automatisointi omasta mielestäni kannattaisi toteuttaa. Kaikissa tavoissa on aina omat hyvät ja huonot puolensa, eivätkä kaikki tekniikat ja menetelmät sovellu jokaiseen tilanteeseen. Huomionarvoista on myös tekniikoiden ja työkalujen nopea kehitys, sillä tilanne voi muuttua hyvinkin lyhyellä aikavälillä.

7.2. Testaus ilman keskitettyä palvelinta

Vaikka omassa Android-projektissa ei työskentelisikään kuin muutama henkilö ja sovellus olisi pienikokoinen, silti sovelluksen testaaminen kannattaa aina. Vähin mitä sovellustestaamisen eteen voi tehdä, on ajaa Monkey-työkalulla sovellus muutamaan kertaan läpi. Tätä en sanoisi kovinkaan päteväksi testaamiseksi, mutta sekin on parempi kuin ei mitään. Tästä päästäänkin siihen, mitä hyviä ja huonoja puolia kullakin testausympäristöllä on ilman palvelinpuolen käyttöä. Tällainen tilanne on yleensä silloin kun kehittää omaa yhden hengen projektia, tai vaihtoehtoisesti rakentaa testitapauksia omalla paikallisella koneellaan isompaan projektiin.

Robotium

Robotium on testauksen aloittamisen kannalta kaikista varteenotettavin vaihtoehto. Alkuun pääsemiseen riittää pelkästään testiprojektiin liitettävä jar-tiedosto, joka sisältää kaiken tarpeellisen. Tällainen helppokäyttöisyys testauksen aloittamiseen madaltaa kynnystä huomattavasti ja innostaa siihen, ettei testauksen suunnittelu jää pelkästään ajatuksen asteelle. Lisäksi mahdollisuus testata jo olemassa olevaa sovellusta ilman, että testattavan sovelluksen lähdekoodeja on edes saatavilla, tekee siitä hyvän työkalun mustalaatikkotestausta ajatellen.

Robotiumin hyviin puoliin voidaan lukea myös sen pitkä elinkaari, sillä versio 1.0 julkaistiin yli kolme vuotta sitten. Tämän perusteella voidaan olettaa että se on varteenotettava vaihtoehto myös tulevaisuudessa laajan käyttäjäkunnan ansiosta.

Robolectric

Android-sovellusten testaukseen Robolectric sopii hyvin ja sille on kertynyt oma laaja käyttäjäkuntansa. Alkuun pääseminen on hieman haastavaa ensimmäisellä kerralla, mutta pienen totuttelun jälkeen testien ajaminen ja kirjoittaminen on yksinkertaista. Suurimmat vaikeudet tulevat tarvittavien kirjastojen lataamisessa ja testiprojektin pystyttämisessä. Voi mennä useita yrityksiä ennen kuin koko ketju saadaan toimimaan moitteettomasti.

Robolectricin hyviin puoliin voidaan lukea laaja dokumentaatio ja testiesimerkkien määrä. Kaikki lähdekoodi on dokumentoitu javadoc-muodossa, josta on generoitu käyttäjiä hyvin palveleva online-versio. Myös muilta käyttäjiltä saatavat neuvot ja keskustelupalsta tarjoavat hyviä eväitä oman testauksen aloittamiseen.

Calabash

Sovellustestauksessa Calabash sijoittuu hieman erikoiseen välimaastoon kun sitä verrataan kahteen aikaisempiin testauskehyksiin. Testitapausten kirjoittaminen on

erilaista ja käyttäjäystävällistä. Lukija ymmärtää jo suoraan testiä katsoessaan mitä sen tulisi tehdä. Tämä voidaankin lukea yhdeksi vahvuudeksi, sillä tämä nopeuttaa testien kirjoittamista.

Huonoihin puoliin voidaan lukea asennuksen vaikeus. Linux pohjaisessa ympäristössä päänvaivaa aiheuttivat oikeanlaisen Java-version, kuin Ruby-sovelluskehyksenkin asentaminen. Kehitysyhteisön sivuilla annetut ohjeet linux-ympäristön asennukseen olivat valitettavan huonot ja epäselkeät.

Mikä on paras?

Kaikista testatuista menetelmistä tekijän oma mielipide kallistuu Robotium-kirjaston puoleen. Suureksi osaksi siitä syystä, että käytön aloittaminen on helppoa, eikä vaadi kohtuutonta asentamista missään vaiheessa. Tämän lisäksi voidaan yksikkötestaukseen lisätä Mockito helpottamaan rajapintojen ja luokkien testausta tarvittaessa. Lisäksi Eclipse-ympäristö tarjoaa valmiit työkalut ja integroinnin testien tulosten näkemiseen eikä ylimääräisiä laajennoksia ole pakko asentaa.

7.3. Testauspalvelimen kanssa automatisoidusti

Sovelluksen testauksen siirtäminen keskitettyyn palvelimeen vaatii huomattavasti enemmän suunnitelmallisuutta kuin äkkiseltään voisi kuvitella. Palvelimen ylläpito ja asennus vaativat perehtymistä. Tämän jälkeen testaajien ja sovelluskehittäjien tulee muistaa ylläpitää testitapauksia aina projektin edetessä. Näin ollen testauksen automatisointiin voi kulua huomattavan paljon aikaa ennen kuin sen todelliset hyödyt tulevat esille.

Kun testausprosessi on saatu kerran automatisoitua, niin siitä eteenpäin testien toteuttaminen on helpompaa. Kynnys aloittaa automaattinen testaus pienissä Android-sovelluksissa voi kuitenkin olla kohtalaisen korkea.

Robotium

Aikaisemmin helppokäyttöisyydellä yllättänyt Robotium osoittautui hieman haasteellisemmaksi kun halutaan suorittaa testejä Jenkinsin kautta. Testien tulosten saaminen helposti ymmärrettävään muotoon vaatii erinäisiä lisäyksiä projektiin. Vaihtoehtoina on joko Jenkinsin antaman konsolisyötteen parsiminen LogParser-laajennoksella, tai vastaavasti Robotium projektiin on lisättävä android-junit-report -kirjasto. Tällä tavalla testien tulokset saadaan XML-muodossa ulos ja ne voidaan esittää Jenkinsin puolella graafisesti.

Robolectric

Muista poikkeava lähestymistapa testien suorittamisessa antaa Robolectricille hyvän etumatkan testien suoritusajoissa. Tällä voi olla merkitystä silloin kun kyseessä on iso projekti, jossa on paljon suoritettavia testejä jokaisella ajokerralla. Testitulosten saaminen Jenkinsiin vaatii myös oman build.xml tiedoston tekemistä, johon voi kulua huomattavasti aikaa, elleivät valmiit mallit toimi odotetulla tavalla.

Calabash

Calabash on testauksen automatisoinnin kannalta hyvinkin helppo testauskehys. Samat testit voidaan ajaa niin paikallisella koneella kuin myös palvelimella helposti. Jenkinsiä ajatellen mahdollisuus saada testien tulokset ulos vaaditussa xml-muodossa helpottaa huomattavasti testien raportointia.

Automatisointi Jenkinsin kanssa onnistuu helposti ja testien suorittamisessa ei ilmennyt mitään erityistä ongelmaa. Calabash on myös testatuista sovelluskehyksistä ainoa, jota on mahdollista hyödyntää myös Robot Frameworkin kanssa. Nämä seikat helpon syntaksin kanssa nostavat sen hyvin korkealle vertailussa.

Mikä on paras?

Tällä kertaa Robolectric ja Calabash ovat olleet hyvin tasaväkisiä automatisoinnin suhteen. Kuitenkin oma suositus kallistuu tällä kertaa Calabashin puoleen. Tähän tulokseen vaikutti eniten tulosten esittämisen helppous, sekä testiprojektin tekeminen ja ylläpito ovat helppoja. Robolectric on ansainnut myös paikkansa testausautomaatiossa silloin, kun vaaditaan tarkempaa lähestymistapaa käyttöliittymän testaukseen.

7.4. Kaupalliset vaihtoehdot

Vaikka opinnäytetyön ensisijaisena tavoitteena oli tutkia, millaisia ilmaisia vaihtoehtoja on olemassa tavalliseen pieneen ja vähän isompaankin sovelluskehitykseen, Android-alustan valtava suosio ja testauksen tarve ovat tuoneet tullessaan myös huomattavan määrän erilaisia yrityksiä, jotka tarjoavat automatisoituja testauspalveluita oikeilla laitteilla.

Yksi tällaisia palveluita tarjoava yritys on suomalainen Bitbar Technologies Oy. Tarjolla on pilvipalvelu, jonka avulla voidaan sovellusta testata useilla erilaisilla laitteilla. Toisena palveluna on tarjolla sovellus, jolla voidaan luoda Robotiumin kanssa yhteensopivia testitapauksia yksinkertaisella käyttöliittymällä.

Aikaisemmassa luvuissa käsiteltyjen Robolectric- ja Calabash-sovelluskehysten taustalla toimii myös ammattimaisia testaukseen erikoistuneita yrityksiä. Calabashista vastaava LessPainful on luonut samanlaista toimintamallia kuin suomalainen BitBar Technologies Oy.

8. POHDINTA

8.1. Tutkimus kokonaisuutena

Opinnäytetyön aihe oli mielenkiitoinen ja ajoittain jopa haastava. Testausmenetelmien kehittyminen on ollut hyvin voimakasta viimeisen vuoden aikana, varsinkin Android-sovellusten testauksen osalta. Tällä hetkellä kehittäjällä on käytössään monipuolinen arsenaali erilaisia tapoja testata sovelluksia ja niiden ominaisuuksia. Myös Google on ottanut käyttöliittymän testaukseen ja yksikkötestauksen huomioon tuodessaan uusia ominaisuuksia kehitystyökaluihin. Käyttöliittymän testaukseen on jo nyt saatavilla Googlen tukema työkalu ja toivottavasti testien raportoimiseen on tulossa parannuksia tulevaisuudessa.

Koko opinnäytetyöprosessi kesti melkein vuoden verran, enemmän tai vähemmän aktiivisesti kirjoittaen. Tuona aikana testausmenetelmien nopea kehitys tuli hyvin esille. Jos jotain testausmenetelmää oli kokeillut kesällä laihoin tuloksin, saattoi reilun puolen vuoden päästä tilanne olla jo aivan toinen. Tämä kiteyttää hyvin sen, miten avoimen lähdekoodin projekteissa voi tapahtua suuriakin muutoksia lyhyen ajan sisällä, varsinkin silloin kun projektin taustalla laaja käyttäjäkunta ja mahdollisesti rahallista tukea yritysmaailmasta.

Kokonaisuudessaan työ täyttää suurimmalta osaltaan sille asetetut tavoitteet, koska työn tarkoituksena oli kartoittaa erilaisia tapoja Android-sovellusten testaamiseen. Näitä tapojen soveltuvuutta tuli sitten arvioida FreeNestin kontekstia ajatellen. Tutkittujen työkalujen tuli myös olla mahdollisuuksien mukaan avointa lähdekoodia.

Itse tutkimuksen tulokset eivät ole niin selkeitä kuin olisin toivonut. Tutkitut testauskehykset ovat lähestymistavoiltaan erilaisia, minkä vuoksi niiden toimintaa on hankala

vertailla suoraan toisiinsa. Toinen testikehys toimii paremmin tietyissä tilanteissa kuin toinen. Tämän takia ei ole olemassa yhtä totuutta siitä, mitä tutkituista vaihtoehtoista tulisi käyttää nyt ja tulevaisuudessa.

8.2. Tulevaisuuden kehityskohteita

Opinnäytetyön aihealue osoittautui loppua kohden huomattavasti laajemmaksi kuin alun perin olin osannut ajatella. Osa teoriaosuudessa käsitellyistä menetelmistä ja toimintavoista jäin vain pieneksi pintaraapaisuksi todellisuudesta. Esimerkiksi testaajan kannalta hyvinkin olennainen koodikattavuus ja sen ilmaisu erinäisillä välineillä jäi kokonaan pois lähemmästä tarkastelusta. Tältä osalta riittää vielä tutkittavaa. Lähinnä ajan puute vaikutti tähän seikkaan ja aihealueen tiukempi rajaus olisi voinut olla ehkä paikallaan.

Toinen kehittämisen arvoinen asia opinnäytetyössä olisi laajempi vertailu erilaisten jatkuvan integraation palvelinten ja testauskehysten välillä. Täten saataisiin kattava kuva siitä, millaisia eri mahdollisuuksia Android-sovellusten testauksen saralla todella on.

Lisäksi yksi mielenkiitoisimmista tulokkaista testauksen saralla on Robot Framework ja sen hyödyntäminen testauksessa. Tämän osa-alueen parantaminen Android-sovellusten testauksessa voisi olla yksi mahdollisista tutkimusaiheista tulevaisuudessa.

LÄHTEET

Android Developers. 2013a. Activity. Viitattu 08.04.2013.
http://developer.android.com/reference/android/app/Activity.html

Android Developers. 2013b. Android, the world's most popular mobile platform.
Viitattu 13.05.2013. http://developer.android.com/about/index.html

Calabash android. 2013. Readme.md. Viitattu 16.03.2013.
https://github.com/calabash/calabash-android

Collin, N. 2010. TDD ja ATDD. Viitattu 14.02.2013.
https://www.cs.tut.fi/~testaus/s2010/luennot/Collin.pdf

Fowler, M. 2007. Mock's Aren't Stubs. Viitattu: 08.05.2013.
http://martinfowler.com/articles/mocksArentStubs.html

FreeNest. 2013. About FreeNest. Viitattu 25.03.2013. http://freenest.org/about

Guide to the Software Engineering Body of Knowledge, Chapter 5. n.d. IEEE
Computer Society:n sivuilta. Viitattu 09.02.2013.
https://www.computer.org/portal/web/swebok/html/ch5#Ref2 SWEBOK HOME,
SWEBOK 2004, html (free)

Hayes, L. 2011. The Automated Testing Handbook. Viitattu 13.02.2012.
http://books.google.com

ISTQB. 2007. ISTQB:n testaussanasto. Viitattu 10.03.2013.
http://www.fistb.fi/sites/fistb.ttlry.mearra.com/files/istqb_sanasto.pdf

JUnit4Android. 2013. Viitattu 16.02.2013.
https://github.com/dthommes/JUnit4Android#view-test-results-in-eclipse

Knott, D. 2012. Robotium, Jenkins and Ant. Viitattu 16.02.2013.
https://dnlkntt.wordpress.com/2012/08/02/robotium-jenkins-and-ant/

LessPainful. 2013. Automated Acceptance Testing for Mobile Apps. Viitattu
16.03.2013. http://calaba.sh/

National Institute of Standards and Technology. The Economic Impacts of Inadequate
Infrastructure for Software Testing. 2002. Viitattu 09.02.2013.
http://www.nist.gov/director/planning/upload/report02-3.pdf

Niedermeyer, Paul E. 2012. Mockito on Android step-by-step. Viitattu 10.05.2013.
http://paulbutcher.com/2012/05/15/mockito-on-android-step-by-step/

Niittyvirta, A. 2012. Testaus voi tuplata ohjelmistotuotteen tuloksen. Viitattu 10.04.2013. http://ohjelmistotestaus.fi/2012/03/testaus-voi-tuplata-ohjelmistotuotteen-tuloksen/

Orr, C. 2013. Android Emulator Plugin – Jenkins. Viitattu 18.04.2013. https://wiki.jenkins-ci.org/display/JENKINS/Android+Emulator+Plugin

Pivotal Labs. 2013. Robolectric: Unit Test Your Android Application. Viitattu 13.03.2013. http://pivotal.github.com/robolectric/

Platform Versions. 2013.Viitattu 12.02.2013. https://developer.android.com/about/dashboards/index.html#Platform

Poimala, S. & Tolvanen, P. 2011. Ketteryys haltuun: Yleisimmät ketterät käytännöt. Sininen meteoriitti. Viitattu 11.02.2013. https://www.meteoriitti.com/fi-FI/tiedotteet/ajankohtaista/ketteryys-haltuun-yleisimmat-ketterat-kaytannot

Sainio, L. 2010. Ohjelmistotestauksen menetelmät ja työvälineet. Viitattu 10.03.2013. https://publications.theseus.fi/bitstream/handle/10024/12297/Sainio_Laura_liite1.pdf

Sanwald, P. 2013. The differences between JUnit 3 and JUnit 4. Viitattu 10.05.2013. http://stackoverflow.com/questions/6685730/the-differences-between-junit-3-and-junit-4

OAMK. 2006. Software Business Competence, ohjelmistotestaus. Viitattu 25.03.2013. https://www.oamk.fi/sbc/testaus/testausstrategiat.htm

Robot Framework. 2013. Robot Framework – Introduction. Viitattu 10.05.2013. http://robotframework.org

Schultz, J. 2012. Eclipse-Robolectric-Sample. Viitattu 04.05.2013. https://github.com/jmschultz/Eclipse-Robolectric-Example

Software Testing Fundamentals. 2012. Viitattu 11.02.2013. http://softwaretestingfundamentals.com Levels, Acceptance Testing.

Software Engineering Radio. 2010. Episode 167: The History of JUnit and the Future of Testing with Kent Beck. Viitattu 10.05.2013. http://www.se-radio.net/2010/09/episode-167-the-history-of-junit-and-the-future-of-testing-with-kent-beck/

JUnit. 2013. Viitattu 10.02.2013. http://en.wikipedia.org/wiki/JUnit

Taina, J. 1999. Ohjelmiston laatu. Viitattu 13.02.2012.
https://www.cs.helsinki.fi/u/taina/ohtu/luennot/k99/laatu/kaikki.html

Taina, J. 2007. Ohjelmistojen testaus – luentokalvot. Viitattu 10.03.2013.
https://www.cs.helsinki.fi/u/taina/ohte/s-2008/luennot/

Torres Milano, D. 2011. Android Application Testing Guide. Birmingham: Pack Publishing Ltd.

User scenario testing for Android. 2013. Viitattu 15.02.2013.
https://code.google.com/p/robotium/

Wolverton, Ray W. 1974. The Cost of Developing Large-Scale Software. Viitattu 14.04.2013. http://www.computer.org/csdl/trans/tc/1974/06/01672595.pdf

LIITTEET

Liite 1: JUnit4 testin perusrunko (JUnit 2013.)

```
import org.junit.*;

public class TestFoobar{
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code executed before the first test method
    }

    @Before
    public void setUp() throws Exception {
        // Code executed before each test
    }

    @Test
    public void testOneThing() {
        // Code that tests one thing
    }

    @Test
    public void testAnotherThing() {
        // Code that tests another thing
    }

    @After
    public void tearDown() throws Exception {
        // Code executed after each test
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        // Code executed after the last test method
    }
}
```

Liite 2: Mockito-testin esimerkkiluokat (Niedermeyer 2012, muokattu)

Rajapinta varastoon

```
package com.example;

public interface Warehouse {
    boolean hasInventory(String product, int quantity);
    void remove(String product, int quantity);
}
```

Varasto-luokka

```
package com.example;

import java.util.HashMap;

public class RealWarehouse implements Warehouse {

    public RealWarehouse() {
        products = new HashMap();
        products.put("Talisker", 5);
        products.put("Lagavulin", 2);
    }

    public boolean hasInventory(String product, int quantity) {
        return inStock(product) <= quantity;
    }

    public void remove(String product, int quantity) {
        products.put(product, inStock(product) - quantity);
    }

    private int inStock(String product) {
        Integer quantity = products.get(product);
        return quantity == null ? 0 : quantity;
    }

    private HashMap products;
}
```

Tilaus-luokka

```
package com.example;

public class Order {

    public Order(String product, int quantity) {
        this.product = product;
        this.quantity = quantity;
    }

    public void fill(Warehouse warehouse) {
        if (warehouse.hasInventory(product, quantity)) {
            warehouse.remove(product, quantity);
            filled = true;
        }
    }

    public boolean isFilled() {
        return filled;
    }

    private boolean filled = false;
    private String product;
    private int quantity;
}
```

Liite 3: Yksinkertainen Robotium-testi

```
package fi.jamk.e6912.veripalveludroid.test;
import android.test.ActivityInstrumentationTestCase2;
import fi.jamk.e6912.veripalveludroid.*;
import fi.jamk.e6912.veripalveludroid.R;
import com.jayway.android.robotium.solo.Solo;

public class BasicTest extends
ActivityInstrumentationTestCase2<VeripalveluDroidActivity> {
      private Solo solo;

      public BasicTest() {
            super("fi.jamk.e6912.veripalveludroid",
VeripalveluDroidActivity.class);
      }

      @Override
      protected void setUp() throws Exception {
            super.setUp();
            solo = new Solo(getInstrumentation(), getActivity());
      }

      public void testOpenSettings() {
            solo.clickOnButton("Application settings");
            assertTrue("did not get PrefsActivity",
solo.waitForActivity("VeripalveluSettings"));
            }
```

```
	protected void tearDown() throws Exception {
		solo.finishOpenedActivities();
	}
}
```

Liite 4: Robotium-projektin build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="NotesList" default="help">

   <!-- local.properties sisältää polun Android-SDK tiedostoihin -->
   <property file="local.properties" />

   <property file="ant.properties" />

   <property environment="env" />
   <condition property="sdk.dir" value="${env.ANDROID_HOME}">
      <isset property="env.ANDROID_HOME" />
   </condition>


   <loadproperties srcFile="project.properties" />

   <!-- quick check on sdk.dir -->
   <fail
         message="sdk.dir is missing. Make sure to generate local.properties using 'android update project' or to inject it through the ANDROID_HOME environment variable."
         unless="sdk.dir"
   />


   <import file="custom_rules.xml" optional="true" />

   <!-- version-tag: 1 -->
   <import file="${sdk.dir}/tools/ant/build.xml" />

</project>
```

Liite 5: Robolectric testiprojektin luominen

Uusin versio ohjeista löytyy osoitteesta: http://pivotal.github.io/robolectric/eclipse-quick-start.html

Android-projektin luominen.

Perus Android-projektin luominen menee samalla lailla kuin minkä tahansa muunkin projektin luominen. Kun perusrunko on luotu, lisätään projektikansion alle test-niminen kansio jonka alle kaikki testit tulevat. Tämä tulisi tehdä pakettienhallinnan kautta

Package Exploerer → New... → Folder

Kansiosta ei tehdä projektin lähdekansiota.

Testiprojektin luominen

Projektin tulee olla tavallinen **Java-projekti**.

File → New Project → Java Project

Luodaan projektin alle kansio **lib**, johon kaikki vaadittavat tiedostot tulevat. Sitten kopioidaan uusin versio Robolectric jar-kirjastosta.

```
cp .../robolectric-X.X.X-jar-with-dependencies.jar
.../MyProjectTest/lib
```

Lisätään projektin riippuvuuksiin JUnit4:

Oikea hiirenpainike → Properties → Java build-path

Library-välilehti: Add Library → Junit → JUnit4.

Lopuksi muut tarvittavat Jar-tiedostot:

Add JARs → valitaan listasta testiprojektin lib kansio ja sieltä robolectric.jar

Add External JARs → Käydään etsimässä Android SDK kansiosta riittävän ison tason android.jar polusta sdk_polku/platforms/android-X/android.jar

Add External JARs → Käydään etsimässä Android SDK kansiosta samaa tasoa oleva map.jar polusta sdk_polku/add-ons/addon_google_apis_google_inc_X/libs/maps.jar

Määritellään ajoasetukset:

Oikea hiirenpainike → Run As → Run Configurations

Kaksoisklikataan ”Junit” valintaa oikeassa laidassa. Huom. ei ”Android Junit Test”.

Valintaan ruksi: ”Run all tests in the selected project, package or source folder: ”

Test runneriksi JUnit4

Valitaan testien ajajaksi ”Eclipse JUnit Launcher”

Vaihtoehtoinen tapa asettaa projekti, löytyy osoitteesta:
https://github.com/bkiers/notes/wiki/Create-new-Eclipse-Android-projects-with-Robolectric-unit-test-support-and-Ant

Liite 6: Robolectric-projektin build.xml

Lähde: (Schultz 2012, Muokattu)

```
<project name="AndroidProjectTest" default="test-report-junit" basedir=".">
	<description>
		Android-Project-Test - Robolectric Tests for Android Application
	</description>

	<!-- set global properties for this build -->

	<property name="libs.dir" value="./libs/"/>
	<property name="build.dir" value="./build/"/>
	<property name="android.project.classpath" value="../android-
project/bin/classes/"/>
	<!--<property name="test.report.dir" value="./test-reports/"/> -->
	<property name="test.report.dir"
value="/var/lib/jenkins/jobs/ElectricTest/workspace/test-reports/"/>
	<property name="test.html.dir" value="./test-report-html/"/>
	<property name="source.dir" value="./src/"/>

	<filelist id="android_jars" dir="${libs.dir}">
		<file name="android.jar"/>
		<file name="maps.jar"/>
	</filelist>

	<filelist id="libs_jars" dir="${libs.dir}">
		<file name="junit.jar"/>
		<file name="hamcrest.jar"/>
		<file name="robolectric-with-dependencies.jar"/>
	</filelist>

	<path id="compile_classpath">
		<filelist refid="libs_jars"/>
		<filelist refid="android_jars"/>
		<pathelement path="${android.project.classpath}"/>
		<pathelement path="${build.dir}"/>
	</path>
```

```
	<path id="junit_classpath">
		<pathelement path="${build.dir}"/>
		<pathelement path="${android.project.classpath}"/>
		<!-- NOTE: junit.jar must come before android.jar! -->
		<filelist refid="libs_jars"/>
		<filelist refid="android_jars"/>
	</path>

	<!-- targets -->

	<target name="init">
		<!-- Create the time stamp -->
		<tstamp/>
		<mkdir dir="${build.dir}"/>
	</target>

	<target name="compile" depends="init" description="compile test source">
		<javac srcdir="${source.dir}" destdir="${build.dir}" debug="true" >
			<classpath refid="compile_classpath" />
		</javac>
	</target>

	<target name="test-run" depends="compile" description="Run JUnit tests">
		<mkdir dir="${test.report.dir}"/>
		<echo message="Running JUnit Tests in directory ${source.dir}..."/>
		<junit showoutput="true" printsummary="yes"
failureproperty="junit.failure" fork="yes" forkmode="once" maxmemory="512m">
			<formatter type="plain"/>
			<formatter type="xml"/>
			<batchtest todir="${test.report.dir}">
				<fileset dir="${source.dir}">
					<include name="**/*Test.java"/>
				</fileset>
			</batchtest>
			<classpath refid="junit_classpath"/>
		</junit>
		<fail if="junit.failure" message="Unit test(s) failed. See reports!"/>
	</target>

	<target name="test-report-junit" depends="test-run" description="Generate JUnit
HTML reports">
		<mkdir dir="${test.html.dir}"/>
		<junitreport todir="${test.report.dir}">
			<fileset dir="${test.report.dir}" includes="TEST-*.xml"/>
			<report format="frames" todir="${test.html.dir}"/>
		</junitreport>
	</target>

	<target name="clean" description="Clean Up" >
		<delete dir="${build.dir}"/>
		<delete dir="${test.report.dir}"/>
		<delete dir="${test.html.dir}"/>
		<delete file="${basedir}/tmp/cached-robolectric-classes.jar"/>
```

</target>
</project>

Liite 7: Calabashin asentaminen

Lähde: https://github.com/calabash/calabash-android/wiki/Ubuntu-Setup

Calabashin (0.4.3.) asentaminen vaatii hieman enemmän työtä verrattuna muihin käytettyihin menetelmiin. Vaatimuksena käyttämiselle on se, että koneelle on asennettuna calabash wikin mukaan:

- Oracle JDK 7 (Ohje)
- Ruby 1.9.3. joko rvm asennustyökalua käyttäen tai sitten erillisestä reposta

Lähde: http://www.wikihow.com/Install-Oracle-Java-JDK-on-Ubuntu-Linux

Javan asennus:

Ensimmäisenä täytyy korvata olemassa oleva OpenJDK versio Oraclen omalla, jos sellainen on koneeseen asennettuna. Ohjeet koskevat 32-bittistä versiota mutta lähdeluettelon kautta löytyy linkki 64-bittiseen asennukseen.

Tarkistetaan, onko käyttöjärjestelmästä käytössä 32- vai 64-bittinen versio.

```
file /sbin/init
```

Tarkistetaan olemassa oleva versio ja poistetaan tarvittaessa.

```
java -version
```

Tuloksena pitäisi olla joko se, ettei Javaa ole asennettu tai sitten käytössä on OpenJDK versio Javasta.

```
java version "1.7.0_15"
OpenJDK Runtime Environment (IcedTea6 1.10pre) (7b15~pre1-0lucid1)
OpenJDK 64-Bit Server VM (build 19.0-b09, mixed mode)
```

Poistetaan vanha versio ja luodaan kansio uudelle Javalle

```
sudo apt-get purge openjdk-\*
sudo mkdir -p /usr/local/java
```

Ladataan uusi versio Oraclen sivuilta ja valmistellaan asennusta. Tiedosto on linux ympäristössä .tar.gz päätteinen 'jdk-7u21-linux-i586.tar.gz'. Oletetaan että lataus on mennyt oman kotikansion Downloads-kansioon.

http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

```
cd /home/"your_user_name"/Downloads
sudo cp -r jdk-7u21-linux-i586.tar.gz /usr/local/java
cd /usr/local/java
sudo chmod a+x jdk-7u21-linux-i586.tar.gz
```

Puretaan pakattu tiedosto:

```
sudo tar xvzf jdk-7u21-linux-i586.tar.gz
```

Muutetaan viittaukset profiiliin jotta järjestelmä löytää uuden Java-version.

```
sudo nano /etc/profile
```

Lisää:

```
JAVA_HOME=/usr/local/java/jdk1.7.0_21
PATH=$PATH:$HOME/bin:$JAVA_HOME/bin
export JAVA_HOME
export PATH
```

Tallenna muutokset ja poistu. Seuraavaksi kerrotaan järjestelmälle mitä Java-versiota tulisi milloinkin käyttää.

```
sudo update-alternatives --install "/usr/bin/java" "java"
"/usr/local/java/jdk1.7.0_21/bin/java" 1

sudo update-alternatives --install "/usr/bin/javac" "javac"
"/usr/local/java/jdk1.7.0_21/bin/javac" 1

sudo update-alternatives --install "/usr/bin/javaws" "javaws"
"/usr/local/java/jdk1.7.0_21/bin/javaws" 1

sudo update-alternatives --set java
/usr/local/java/jdk1.7.0_21/bin/java

sudo update-alternatives --set javac
/usr/local/java/jdk1.7.0_21/bin/javac

sudo update-alternatives --set javaws
/usr/local/java/jdk1.7.0_21/bin/javaws
```

Päivitetään tiedot:

source /etc/profile

Tarkistetaan että asennus on onnistunut:

```
java -version
javac -version
```

Kun Eclipse:ä asennetaan uudelle Javalle, niin silloin saattaa tulla virheilmoitus ja sovellus ei käynnisty. Silloin yleensä auttaa seuraava keino:

```
ln -s /usr/lib/jni/libswt-* ~/.swt/lib/linux/x86/
```

Muokataan vielä kerran profiilia:

```
sudo nano /etc/profile
```

Lisättävä Android-SDK polku:

```
ANDROID_HOME=/home/freenest/android-sdk-linux
export ANDROID_HOME
```

Päivitetään tiedot:

source /etc/profile

Rubyn asennus:

Asennuksen voi tehdä joko rvm:ää käyttäen tai sitten asentaa pakettienhallinnan kautta.

Lähde: http://wiki.brightbox.co.uk/docs:ruby-ng

```
sudo apt-get install python-software-properties
sudo apt-add-repository ppa:brightbox/ruby-ng
sudo apt-get update
sudo apt-get install ruby1.9.3
```

Calabash-android kirjaston asennus:

```
sudo gem install calabash-android
```

Liite 8: Jenkinsin asentaminen FreeNest-ympäristöön

Helpoimmin Jenkinsin päivitys ja ylläpito onnistuvat kun se asennetaan virallisesta Jenkins-pakettivarastosta. Tällä tavalla päivitykset hoituvat helpoiten.

Lisätään pakettivarasto listaan:

```
wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key |
sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ >
/etc/apt/sources.list.d/jenkins.list'

sudo apt-get update
```

Asennetaan Jenkins:

```
sudo apt-get install jenkins
```

Sitten käydään muuttamassa hieman porttien ohjausta, jotta saadaan käyttöön haluttu portti HTTPS-yhteydelle. Tämä on asennuksen kannalta tärkeä vaihe, jolla saadaan Jenkins käyttämään pelkästään salattua yhteyttä.

```
sudo nano /etc/default/jenkins
```

```
HTTP_PORT=-1
HTTPS_PORT=8443
JENKINS_ARGS="--webroot=/var/run/jenkins/war --httpPort=$HTTP_PORT
--httpsPort=$HTTPS_PORT –ajp13Port=$AJP_PORT
```

Liite 9: Robotium projektin määritteleminen Jenkinsissä

Lähteenä käytetty: (Knott 2012, muokattu)

Jenkinsille projektia luotaessa esivaatimuksena on se, että Android Emulator -laajennos on asennettuna ja testattaville projekteille on luotu build-tiedostot Ant-työkalua varten.

Pääprojekti

Pääprojektin luonti aloitetaan valitsemalla New Job ja luomalla uusi vapaamuotinen projekti valitsemalla ”freestyle software project”

Valitaan halutaanko projekti hakea jostain tietystä kansiosta vai haetaanko uusin versio aina versionhallinnan kautta. Tämän jälkeen lisätään uusi ”Build step” ja valitaan ”Invoke ant”. Parametreiksi annetaan: ”clean debug” ja advanced napin

tapaa osoitetaan build.xml tiedoston sijainti tarvittaessa.

Nyt Jenkinsin pitäisi osata kääntää aina uusin versio ja luoda tarvittavat tiedostot.

Määritellään projektille vielä käännöksen jälkeinen toiminto, joka aloittaa testien ajamisen jos käännös onnistuu. Valitaan ”Add post-build actions”-napista vaihtoehto ”Build other projects”. Lisätään tuleva testiprojekti tähän kohtaan.

Testiprojekti

Testiprojektin alku luonti menee samaan tapaan kuin pääprojektin luominen. Testiprojekti ei määritellä käännöksen jälkeiseen toimintoon uuden projektin kääntöä, mutta voidaan valita jotain muita tarvittavia toimintoja testien julkaisua varten.

Kun Android Emulator -laajennos on asennettuna, niin valittavana pitäisi olla ”Run an Android emulator during build”. Tämän valinnan alta voidaan sitten määritellä emulaattorin ominaisuudet.

Lisätään ”Add buld step”- napilla kaksi ”Install Android Package”-tyyppistä tehtävää. Ensiksi tavallinen sovellus ja polku mistä apk-tiedosto löytyy. Sen jälkeen testiprojektin käännöksestä tuleva apk-tiedosto.

Viimeisenä vaiheena laitetaan ”Excecute shell”, jolla ajetaan Android SDK:n mukana tulevalla työkalulla testiajo.

/var/lib/jenkins/tools/android-sdk/platform-tools/./adb shell am instrument -w com.jayway.test/android.test.InstrumentationTestRunner

Android SDK:n polku voidaan tarvittaessa korvata määritellyllä muuttujalla ”ANDROID_HOME”.