LAPIN AMK
Lapland University of Applied Sciences

ANDROID-SOVELLUKSEN TESTAUSTYÖKALUT JA TESTAUSMENETELMÄT

Söderström Arto

Opinnäytetyö
Tekniikka ja liikenne
Tieto- ja viestintätekniikka
Insinööri (AMK)

2018

LAPIN AMK
Lapland University of Applied Sciences

Opinnäytetyön tiivistelmä

Tekniikka ja liikenne
Tieto- ja viestintätekniikka
Insinööri (AMK)

Tekijä Arto Söderström Vuosi 2018
Ohjaaja Aku Kesti
Toimeksiantaja Lapin ammattikorkeakoulu - pLAB
Työn nimi Android-sovelluksen testaustyökalut ja testausmenetelmät
Sivu- ja liitesivumäärä 47 + 6

Opinnäytetyön tarkoituksena oli tutkia Android-sovelluksen testaukseen käytettyjä testaustyökaluja ja -menetelmiä, joiden avulla pystyttäisiin parantamaan ja automatisoimaan testausprosessia.

Työssä käytiin läpi sovellustestauksen tavoitteita, testaustapoja ja testaustasoja. Yleisen sovellustestauksen teorian jälkeen tutkittiin Android-sovelluksen automatisoitua testausta. Tutkimuksessa tarkasteltiin useita testaustyökaluja ja niiden käyttöä testauksen eri vaiheissa. Tietoperustana käytettiin sähköisiä kirjoja ja dokumentaatioita sekä internetistä löytyviä artikkeleita.

Työn tuloksena on tutkimusdokumentti sovellustestauksesta ja Android-sovelluksen testausprosessista. Tutkimuksen avulla saatiin tietoa, miten sovellustestausta tulisi suorittaa ja mitä työkaluja testaamisessa kannattaa käyttää.

Pääosin opinnäytetyön tavoitteet saatiin toteutettua, mutta aiheen laajuuden takia aiheesta pystyttiin käymään läpi vain pieni osa. Varsinkaan testaustyökaluja ja niiden ominaisuuksia ei pystytty tutkimaan kovin syvällisesti.

Avainsanat Android, Espresso, JUnit, Mockito, ohjelmistotestaus

LAPIN AMK
Lapland University of Applied Sciences

Abstract of Thesis

School of Technology,
Communication and Transport
Degree Programme in Information
and Communication Technology
Bachelor of Engineering

Author Arto Söderström Year 2018
Supervisor Aku Kesti
Commissioned by Lapland University of Applied Sciences - pLAB
Subject of thesis Android application testing tools and methods
Number of pages 47 + 6

The purpose of this thesis was to examine the testing tools and methods used in Android application testing, and how those could be used to automatize and improve the testing process.

First the objectives of software testing, testing methods and testing levels were examined, followed by the examination of Android-application testing with automated tests. In the study several testing tools were selected for closer inspection to study their use at various stages of testing. The sources used were electronic books, developers' documentation and articles on the Internet.

The result of this work is a research document on software testing and Android application testing process. Study provided information on how to perform software testing and what tools can and should be used for Android application testing. The objectives of the thesis were mainly achieved, but due to the extent of the topic, only a small part of the subject could be studied. In particular, the testing tools and their features could not be thoroughly examined in the thesis.

Key words Android, Espresso, JUnit, Mockito, software testing

SISÄLLYS

KÄYTETYT MERKIT JA LYHENTEET

Alfatestaus	ennen julkaisua suoritettu testausvaihe, jossa organisaation sisäiset henkilöt testaavat ohjelmistoa
Android-emulaattori	koneella pyörivä emuloitu virtuaalinen Android-laite
Annotaatio	luokkaa, metodia, muuttujaa, parametria tai pakettia edeltävä @-alkuinen merkintä, joita käytetään metatietojen määrittämiseen ja ajon aikaisen suorituksen muokkaamiseen
Betatestaus	testausvaihe, jossa organisaation ulkopuoliset henkilöt testaavat ohjelmistoa
Get-metodi	funktio, joka palauttaa muuttujan arvon (*Getter*)
Gradle	Android-projektin koonnissa käytetty koontityökalu
Integraatiotesti	kahden tai useamman komponentin välistä toimivuutta testaava testi (*Integration test*)
JVM	Java virtuaalikone, jossa Java-sovellukset suoritetaan (Java Virtual Machine)
Matcheri	vertailuobjekti tai -metodi, jonka avulla tarkastellaan vastaavuutta
Mockata, matkia	testattavan ominaisuuden ulkoisia riippuvuuksia jäljitellään mock-objektien avulla (*Mock*)
Mock-objekti, testitynkä	testauksessa käytetty komponentin sijaisobjekti, jonka avulla simuloidaan komponentin toimintaa (*Mock object, test stub*)
Kirjasto	kokoelma, joka sisältää valmiita ominaisuuksia ja komponentteja (*Library*)
Käyttöliittymätesti	käyttöliittymän toimivuutta testaava testi (*UI-test, user interface test*)
Poikkeus	sovelluksen ajonaikainen virhe, joita voi tulla esimerkiksi nollalla jaettaessa (*Exception*)
Refaktoroida	parannetaan ohjelmakoodin luettavuutta, uudelleenkäytettävyyttä ja rakennetta muokkaamalla ohjelmakoodia (*Refactor*)
Set-metodi	funktio, joka asettaa muuttujan arvon (*Setter*)

Sisällöntarjoaja	tietojen tallennuksessa ja jakamisessa käytetty komponentti (*Content provider*)
Taustapalvelu	sovelluksen taustalla pyörivä komponentti, jota voidaan käyttää toiminallisuuksien jakamiseen muille sovelluksille tai pitkäaikaisten operaatioiden suorittamiseen (*Service*)
Testitapaus	yksi ominaisuuden testattava tapaus, jossa tietyillä syötteillä odotetaan tietynlaista tulosta (*Test case*)

1 JOHDANTO

Teknologian ja elektroniikan kehittyessä älylaitteet ovat nostaneet suosiotaan. Yhä useampi henkilö omistaa älypuhelimen, tablettitietokoneen, älytelevision tai älykellon. Langattoman mobiiliverkon yleistymisen ja nopeutumisen ansiosta internetyhteys on saatavilla kaikkialla, mikä on kasvattanut älypuhelimien ja tablettitietokoneiden käyttöä. Yhä useammat arkipäiväiset asiat laskujen maksamisesta elokuvien katsomiseen ovat mahdollisia mobiililaitteilla vähentäen tarvetta käyttää perinteisiä tietokoneita. Älylaitteiden suosion ja käytön kasvaessa myös sovellusten määrä ja kilpailu kasvavat. Menestyäkseen sovellukseen tulee olla laadukas erottuakseen muista kilpailijoista.

Android on Googlen omistama Linux-pohjainen avoimen lähdekoodin mobiilikäyttöjärjestelmä. Puhelimien ja tablettitietokoneiden lisäksi Androidista on versiot älykelloille (Wear OS), älytelevisioille (Android TV) ja autoille (Android Auto). Käyttöjärjestelmän variaatioita käytetään myös muissakin laitteissa kuten pelikonsoleissa, kameroissa ja tietokoneissa. (Android Developers 2018a.)

Android-käyttöjärjestelmä on suosituin älypuhelinkäyttöjärjestelmä. Vuoden 2017 alussa Androidin markkinaosuus älypuhelimissa käytetyistä käyttöjärjestelmistä oli 85,0 prosenttia. Käyttöjärjestelmän suurin kilpailija puhelinmarkkinoilla on Applen iOS-käyttöjärjestelmä 14,7 prosentin markkinaosuudella vuoden 2017 alkupuolella. (IDC 2018.)

Sovelluksia Androidille kehitetään pääosin Java-ohjelmointikielellä. Javan lisäksi voidaan käyttää myös Kotlin-ohjelmointikieltä, jonka Google lisäsi Androidin toiseksi viralliseksi ohjelmointikieleksi (Android Developers 2018a). Kielien yhteensopivuus mahdollistaa molempien kielien käyttämisen yhtäaikaisesti (Android Developers 2018i). Android-kehitys onnistuu Windows-, Mac- ja Linux-käyttöjärjestelmillä, ja kehittämiseen käytetään Googlen kehittämää Android Studio-ohjelmaa.

Opinnäytetyöni tavoitteena on tutkia miten ja millä Android-sovelluksia voidaan testata kehityksen aikana varmistaakseen sovelluksen laadun ja toimivuuden. Erityisesti keskitytään sovelluksen ohjelmalliseen testaamiseen eri testaustyökalujen avulla ja miten ne sopeutuvat jokapäiväiseen kehitykseen. Aluksi käydään yleisesti läpi sovellustestauksen teoriaa ja tavoitteita. Sovellustestauksen teoriaosuuden jälkeen keskitytään Android-sovelluksen testaukseen ja siihen käytettäviin työkaluihin.

Opinnäytetyö on tehty toimeksiantona Lapin ammattikorkeakoulun tieto- ja viestintätekniikan laboratoriolle (pLab). pLabissa on kehitetty useita Android-sovelluksia, joiden testaaminen on koettu aikaa vieväksi ja työlääksi prosessiksi, jota pyritään parantamaan työni pohjalta. Otin aiheen vastaan, koska olen useasti ajatellut tutustua sovellustestaukseen kehittääkseni ohjelmointi- ja testaustaitojani.

2 JOHDANTO SOVELLUSTESTAUKSEEN

2.1 Sovellustestauksen tavoitteet

Sovelluksen testaus on tärkeä osa sovelluskehitystä varmistaen sovelluksen toimivuuden ja käytettävyyden, oli kyseessä sitten pieni yhden henkilön tekemä sovellus tai kansainvälisen yrityksen sovellus. Testaamisen avulla virheet löytyvät aikaisemmassa vaiheessa säästäen projektiin käytettyjä resursseja ja ylläpitokuluja (Blundell & Torres Milano 2015, 29). Pienetkin virheet ja ongelmat sovellusta käyttäessä voivat johtaa huonoihin käyttökokemuksiin mahdollisesti vähentäen käyttäjämäärää. Pahemmissa tapauksissa ongelmat voivat johtaa jopa yksilöiden oikeuksien rikkomiseen kuten Yhdysvaltojen presidenttivaaleissa, joissa iso määrä ääniä jäi rekisteröimättä äänestyskoneiden ohjelmistoissa olevan vian takia. (Homès 2011, 1–2.)

Aikaisemmin sovelluksien testaaminen on suoritettu pääosin ihmisvoimalla. Tietokoneiden ja testityökalujen kehittyessä on enemmän ja enemmän siirretty testausvastuuta tietokoneille. Automaattisessa testaamisessa testaukseen käytetyt resurssit koostuvat suurimmaksi osaksi ohjelmallisten testien luomiseen ja päivittämiseen. Testien automatisointi nopeuttaa testien suorittamista kehityksen aikana. Automatisoinnin avulla testit pysyvät myös yhtenäisinä poistaen inhimillisistä tekijöistä johtuvia poikkeamia testausprosessissa. (Vocke 2018.)

Sovelluksen automaattinen testaaminen voi viedä kehityksen alkuvaiheessa huomattavasti resursseja, mutta nämä resurssit saadaan takaisin sovelluksen laajentuessa säästäen manuaalisen testauksen määrää sovelluksen toiminnallisuuksien muuttuessa. Sovellukseen tehtyjen muutoksien jälkeen voidaan ajaa läpi automaattiset testit. Ohjelmallisten testien kirjoittaminen auttaa myös kehittäjää paremmin ymmärtämään ominaisuuden vaatimukset ja mahdolliset ongelmat, koska testejä on mahdotonta kirjoittaa ohjelmakoodille, jota ei ymmärrä. Testien tekeminen helpottaa myös uusien ominaisuuksien lisäämistä ja vanhojen muokkaamista, koska lähdekoodin muokkauksesta johtuvat virheet olemassa olevissa komponenteissa tulevat esille jo testien ajovaiheessa eikä loppukäyttäjien käyttäessä sovellusta. (Blundell & Torres Milano 2015, 29.)

2.2 Sovellustestauksen kohteet

Sovelluksen testauskohteet ja -tarpeet muodostuvat useasta eri kriteeristä. Testaus suunnitellaan projektin alussa projektikohtaisesti ottaen huomioon käytettävissä olevat resurssit, ohjelmointikielet, alustat, käytetyt teknologiat, sovelluksen kohdeympäristö ja toimintavarmuustavoitteet.

Optimaalisessa tilanteessa jokainen koodilause olisi testattuna, jotta mahdolliset virheet tulisivat esille jo kehitysvaiheessa. Tässä tilanteessa testit testaisivat myös asioita, jotka eivät periaatteessa voi rikkoutua, kuten normaaleja get- ja set-metodeita. Ongelmat näissä tulevat esille jo koodin kääntövaiheessa tai viimeistään ominaisuuden testeissä. Liian tarkkojen ja hyödyttömien testien kirjoittaminen voidaan katsoa olevan resurssien tuhlaamista. Testaustarve tulee määrittää jokaisen komponentin kohdalla erikseen. (Blundell & Torres Milano 2015, 31.)

Testattavat ominaisuudet voidaan jakaa toiminnallisiin (*functional*) ja ei-toiminnallisiin (*non-functional*). Toiminnallisessa testauksessa testataan ohjelman ominaisuuksia tarkastelemalla, että ne toimivat halutulla tavalla ja täyttävät määritetyt vaatimukset. Ei-toiminnallinen testaus on ohjelman ja sen komponenttien operoinnin tutkimista ja testaamista. Suorituskyky-, turvallisuus-, käytettävyys ja luotettavuustestit ovat esimerkkejä ei-toiminnallisesta testauksesta. (Homès 2011, 66–69.)

Komponenttien ja ominaisuuksien suorittamisen jokainen mahdollinen polku tulee ottaa huomioon testeissä (Vocke 2018). Polku määrittyy suorituksen aikana ehtolauseiden ja odotettujen poikkeuksien perusteella. Pelkästään odotetun polun testaaminen todentaa toimivuuden vain optimaalisessa tilanteessa, jossa ei ole epäkelpoisia syötteitä eikä virheitä tapahdu suorittamisen aikana. Esimerkiksi sovelluksessa voisi olla yksinkertainen yhteydenottolomake, joka sisältää pakolliset tekstikentät viestin, nimen ja sähköpostin täyttämiseen sekä napin tietojen lähettämistä varten. Lomakkeen testien pitäisi testata, että nappia painettaessa lomakkeen tiedot lähetetään tekstikenttien ollessa täytettynä, mutta muissa tilanteissa lähettämisen sijaan käyttäjälle näytetään virheilmoitus puuttuvista tiedoista.

Testauksen aikana voi kuitenkin tulla vastaan tilanteita, joissa toimintojen ja syötteiden määrä on niin suuri, että kaikkia kombinaatioita (polkuja) ei ole mitenkään mahdollista testata. Esimerkiksi laskinsovelluksen testauksessa ei ole mitenkään mahdollista testata laskimen kaikkia toimintoja kaikilla mahdollisilla syötteillä. Näissä tilanteissa testien määrää tulee rajoittaa, jotta testaus on ekonomisesti järkevää. (Homès 2011, 11.)

2.3 Sovellustestauksen toteutustavat

Testaus voidaan suorittaa manuaalisesti ihmisten toimesta tai tietokoneilla. Pienemmässä sovelluksessa testaus voi olla kokonaan manuaalista, mutta mitä suuremmaksi ja monimukaisemmaksi sovellus kasvaa sitä enemmän testaus tulisi automatisoida. Manuaaliset testit ovat usein aikaa vieviä, tylsiä, toistavia ja virheherkempiä.

Testaajien tulee ymmärtää, että kaikki ohjelmistot sisältävät ohjelmointivirheitä. Jokaisen virheen löytäminen ja poistaminen ohjelmistosta on mahdotonta. Tämän takia testaajien ja testien tärkein tavoite on löytää merkittävät ongelmat, jotka voivat haitata ohjelman toimivuutta. (Loveland, Miller, Prewitt & Shannon 2004, 6.)

Automaattisten testien yksi isoimmista hyödyistä on ohjelmakoodin muokkaamiseen aiheuttamien virheiden huomaaminen. Testit voivat kuitenkin ominaisuuden koodin muokkaamisen jälkeen epäonnistua, vaikka ominaisuus toimisikin odotetulla tavalla. Tällöin testi on selvästi suunniteltu testaamaan liikaa ominaisuuden implementaatiota käyttäytymisen sijaan. Implementaation testaaminen joissakin tilanteissa on hyväksyttävää, jopa pakollista, mutta yleisesti sitä kannattaisi välttää. Ominaisuuden käyttäytymisen ja tuloksien testaaminen vähentää tarvetta uudelleenkirjoittaa alkuperäisiä testejä. (Vocke 2018.)

Testaukseen liittyy eri testaustapoja, jotka määrittelevät miten eri tavoilla testausta voidaan lähestyä. Ohjelmistokehitystä voidaan myös tehdä käyttäen erilaisia kehitysprosesseja, joissa testaaminen suoritetaan vaihtelevin menetelmin. Seuraavissa osioissa käydään läpi testaustapoja ja testauksen eri tasoja.

3 TESTAUSTAVAT

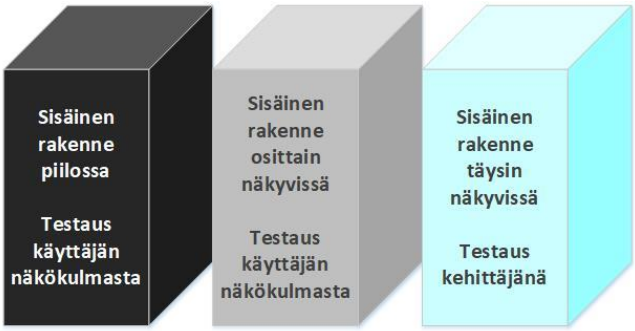
3.1 Staattiset ja dynaamiset testaustavat

Testaustavat voidaan jaotella kahteen kategorioihin: staattisiin ja dynaamisiin. Staattisia testaustapoja ovat erityyppiset katselmoinnit ja tarkastelut. Tyypillisiä katselmointeja ovat koodikatselmoinnit, joiden aikana ohjelmakoodia ei suoriteta, vaan keskitytään koodin logiikkaan, rakenteeseen, nimeämiskäytäntöihin ja ylläpidettävyyteen. Suuri etu staattisissa testaustavoissa on mahdollisuus käyttää niitä ilman funktionaalista tai toimivaa sovellusta, jolloin näitä tapoja voidaan hyödyntää sovelluksen komponenttien testaamisessa, vaikka komponentit eivät ole toimivia. Ohjelmointiympäristöjen ja -työkalujen kehityksen ansiosta ohjelmakoodin perusanalysointi on siirtynyt enemmän tietokoneille, jotka voivat jatkuvasti ja nopeasti etsiä koodista perusvirheitä, kuten syntaksivirheitä ja alustamattomia muuttujia. Sovelluksen komponenttien lisäksi katselmointeja ja tarkastuksia voidaan toteuttaa myös esimerkiksi dokumenteille, testisuunnitelmille ja testitapauksille. Staattiset testaustavat ovat usein dynaamisia halvempia ja niiden tuottoprosentti on korkeampi, koska viat löydetään aikaisemmassa vaiheessa. (Homès 2011, 91–93, 119–120.)

Dynaamisissa testaustavoissa testataan ohjelmiston suorituksen aikaista toimintaa ja käyttäytymistä, joten ohjelmiston tai järjestelmän tulee olla toimintakunnossa. Testejä voidaan kuitenkin suorittaa yksittäisille komponenteille, vaikka ohjelma ei olisikaan täysin valmis. Tarkoituksena on löytää ohjelman suorituksen aikana tapahtuvat virheet, joita ei tunnistettu tai kyetty testaamaan staattisten testaustapojen avulla. Dynaamisen testaus perustuu kirjoitettuihin testitapauksiin, joissa ohjelmiston odotetaan käyttäytyvän tietyllä tavalla annetuilla syötteillä. (Homès 2011, 91–93.)

3.2 Laatikkomalli

Laatikkomalli on yleisesti dynaamisessa ohjelmistotestauksessa käytetty malli, joka jakaa testaustavat mustalaatikko-, lasilaatikko- ja harmaalaatikkotestaukseen (Kuvio 1). Lasilaatikkotestauksesta käytetään myös termiä valkolaatikkotestaus. Mallissa testattava ohjelma on kuvattuna laatikkona, jonka sisällä sijaitsee ohjelman sisäinen rakenne ja toteutus.



Kuvio 1. Laatikkomalli

Mustalaatikkotestauksessa mustat seinät kuvaavat kykenemättömyyttä tarkastella ohjelman sisäistä rakennetta. Testauksessa keskitytään ohjelman toiminnallisuuteen ja määriteltyihin vaatimuksiin välittämättä toiminnallisuuksien toteutustavasta. Ohjelman oletetaan toimivan oikein, kun ohjelma täyttää määritetyt vaatimukset ja määrittelyt. Tämän vuoksi mustalaatikkotestausta kutsutaan myös määrittelypohjaiseksi testaukseksi. Mustalaatikkotestausta voidaan käyttää jokaisella testaustasolla yksikkötesteistä lähtien, mutta sitä käytetään erityisesti hyödyksi hyväksyntä- ja järjestelmätestauksessa, koska testaajilla ei tarvitse välttämättä olla ohjelmointiosaamista. (Homès 2011, 143–145.)

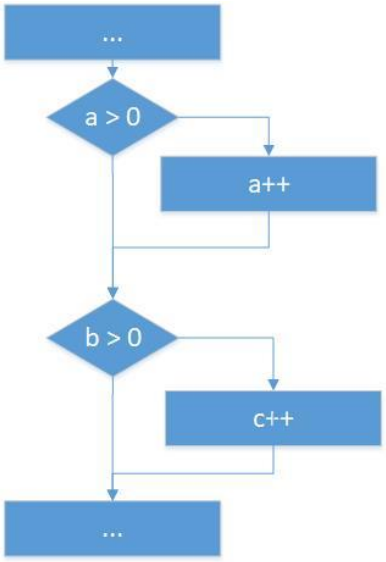
Lasilaatikkotestauksessa ohjelman sisäinen rakenne ja toteutus ovat näkyvissä testaajalle. Päinvastoin kuin mustalaatikkotestauksessa, lasilaatikkotestauksessa keskitytään vaatimuksien sijaan enemmän toiminnallisuuksien toteutustapaan kooditasolla. Ohjelman todetaan olevan toimiva, kun ohjelmiston sisäiset rakennekomponentit ovat todettu toimiviksi, minkä vuoksi testaustapaa kutsutaan usein myös rakennepohjaiseksi testaukseksi. Lasilaatikkotestausta tapahtuu pääosin yksikkötestausvaiheessa, minkä takia ominaisuuden ohjelmoijat usein suunnittelevat ja tekevät testitapaukset, koska heillä on parempi ymmärrys ominaisuuden implementaatiosta. (Homès 2011, 143–144, 172.)

Harmaalaatikkotestaus on mustalaatikko- ja lasilaatikkotestauksen välimalli, jossa testaajalla on jonkinlainen tietämys ohjelman toteutustavasta ja käytetyistä algoritmeista. Usein varsinainen testaus harmaalaatikkotestauksessa suoritetaan mustalaatikkotestauksena. Tietämystä sisäisestä rakenteesta käytetään hyödyksi kattavien testitapauksien suunnittelussa. (Homès 2011, 143–144.)

3.3 Koodikattavuudet

Koodikattavuudet (*code coverage*) ovat mittareita, jotka ilmaisevat prosentuaalisesti miten suuri osa ohjelmakoodista suoritetaan testien aikana. Kattavuudet eivät suoranaisesti ole testaustapoja, mutta niiden avulla nähdään ovatko tehdyt testit kattavia ja onko jotakin jäänyt testaamatta. Testaamaton koodi voi olla indikaattori, että testit eivät ole tarpeeksi kattavia tai testattava komponentti sisältää turhaa koodia.

Yksinkertaisin koodikattavuus on lausekattavuus (*statement coverage*), joka ilmaisee prosentuaalisesti testien aikana suoritettujen lauseiden (käskyjen) määrän. Lausekattavuuden ollessa sata prosenttia jokainen lause on suoritettu jossakin vaiheessa testejä. Tämä ei kuitenkaan takaa, että testit kattaisivat kaikki mahdolliset tilanteet. Otetaan esimerkiksi yksinkertainen ohjelmakoodi, jossa on peräkkäin kaksi ehtolausetta, jotka molemmat kasvattavat muuttujan arvoa ehdon ollessa totta (Kuvio 2). Ohjelmakoodin testauksesta saisi sadan prosentin lausekattavuuden jo yhdellä testillä, jos testissä muuttujat a ja b ovat molemmat suurempia kuin nolla. Testien tulisi vähintään kattaa tilanne, joissa molemmat ehdot ovat epätosia. (Homès 2011, 174–175.)



Kuvio 2. Yksinkertaisen ohjelmakoodin logiikka

Lausekattavuuden lisäksi on useita muita koodikattavuuksia, kuten haarakattavuus (*branch coverage*), polkukattavuus (*path coverage*) ja ehtokattavuus (*condition coverage*), joiden avulla pyritään todentamaan testien kattavuus tarkemmin kuin lausekattavuudella. Näiden kattavuuslukujen laskemisessa on paremmin otettu huomioon ehtolauseiden aiheuttamat haarautumat ja polut. Yksinkertaisin näistä kolmesta esimerkistä on haarakattavuus, joka määrittää kuinka iso prosenttimäärä haarautumista on suoritettu. Polkukattavuus ilmaisee, kuinka monta prosenttia mahdollisista ohjelman poluista on käyty läpi testauksen aikana. Jokaisen polun läpikäyminen testauksen aikana on usein mahdottomuus, koska jokainen ehtolauseiden kombinaatio muodostaa oman polkunsa. Ehtokattavuutta laskettaessa tarkastellaan ovatko ehtolausekkeiden jokainen yksittäinen ehto ollut jossakin vaiheessa testausta sekä tosi, että epätosi. (Homès 2011, 175–187.)

3.4 Testivetoinen kehitys

Testivetoinen kehitys (*test-driven development, TDD*) on ketterien menetelmien rinnalla syntynyt sovelluskehitysprosessi, jossa ominaisuuden ohjelmalliset testit suunnitellaan ja kirjoitetaan ennen ominaisuuden ohjelmakoodia. Testitapaukset suunnitellaan määrittely- ja vaatimusdokumentaatioon pohjalta. Prosessin avulla pyritään parantamaan sovelluksen ja ohjelmakoodin laatua. (Bender & McWherter 2011, 8–9.)

Testivetoisessa kehityksessä seurataan kolmevaiheista iteratiivista kehityssykliä (Kuvio 3). Ensimmäisessä vaiheessa suunnitellaan ja kirjoitetaan määrittelyiden pohjalta ominaisuuden testit. Tässä vaiheessa testit eivät saisi mennä läpi, koska testattavan ominaisuuden ohjelmakoodia ei ole vielä kirjoitettu. Vaatimuksien muuntaminen testitapauksiksi auttaa samalla kehittäjää ymmärtämään paremmin ominaisuuden vaatimukset ja mahdolliset ongelmatilanteet. (Blundell & Torres Milano 2015, 175–177.)

Kuvio 3. Testivetoisen kehityksen vaiheet

Hylättyjen testien tekemisen jälkeen kirjoitetaan varsinainen ohjelmakoodi. Aluksi ohjelmakoodia tulisi kirjoittaa minimaalisin määrä, jolla testien ohjelmakoodi saadaan kääntymään ilman virheitä. Tämän jälkeen ohjelmakoodia kirjoitetaan lisää niin paljon kuin on tarpeellista, jotta testit menevät hyväksytysti läpi. Ominaisuuden ohjelmakoodin ei tarvitse vielä tässä vaiheessa olla täydellistä. (Blundell & Torres Milano 2015, 175–176.)

Edellisen vaiheen päätarkoituksena oli saada testit hyväksytysti läpi mahdollisimman nopeasti ja yksinkertaisesti välittämättä ohjelmakoodin laadusta. Laadun parantamiseksi syklin viimeisessä vaiheessa refaktoroidaan koodi siistimmäksi. Refaktorointivaiheessa ohjelmakoodin luettavuutta ja ylläpidettävyyttä parannetaan esimerkiksi jäsentelemällä ohjelmakoodi paremmin siirtämällä koodin osia omiin metodeihin ja muuttujiin. Refaktoroimisen jälkeen on tärkeätä suorittaa testit vielä kerran, jotta varmistetaan, että ohjelmakoodin muokkaaminen ei ole rikkonut toiminnallisuuksia. (Blundell & Torres Milano 2015, 176.)

Testivetoinen kehitys ensisijaisesti varmistaa, että testit ovat ylipäänsä tehtynä. Perinteisessä kehityksessä ohjelmallisten testien kirjoittaminen voi jäädä tekemättä, koska testit kirjoitetaan vasta ohjelmakoodin jälkeen. Muita testivetoisen kehityksen hyötyjä ovat muun muassa.

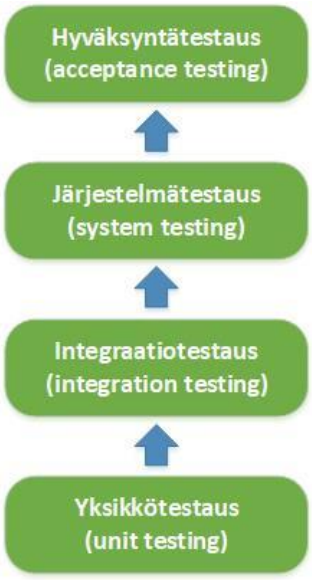
- Ohjelmakoodin laatu paranee, koska testattavan ohjelmakoodin kirjoittaminen vaatii hyvien käytänteiden seuraamista.
- Prosessi kannustaa kehittäjiä kirjoittamaan ohjelmakoodia vain sen verran kuin on tarpeellista, jotta määritellyt vaatimukset täyttyvät.
- Ohjelmakoodin taataan täsmäävän määriteltyjen vaatimuksien kanssa.
- Kirjastot ja rajapinnat ovat yksinkertaisempia ja kohdistetuimpia, jotka helpottavat niiden käyttöä, luettavuutta ja ylläpidettävyyttä.
- Käyttämättömän ja turhan ohjelmakoodin määrä vähenee.
- Sovelluksen arkkitehtuurista tulee joustavampi ja helpommin laajennettava. (Bender & McWherter 2011, 9–10.)

4 TESTAUSTASOT

4.1 Yleistä testaustasoista

Ohjelmiston testaaminen voidaan jakaa usealle eri tasolle, joissa testattavan kohteen laajuus vaihtelee. Riippumatta käytetystä kehittämismallista ohjelman kehityksen aikana testattavat asiat pysyvät samana. Alimmilla tasoilla testataan yksittäisten komponenttien toimivuutta ja näiden komponenttien integroimista toisiinsa. Integrointien jälkeen varmistetaan, että koko järjestelmä toimii odotetulla tavalla kokonaisuutena. Varsinkin ennen lopullista ohjelman julkaisemista on tärkeätä, että asiakkaat tai loppukäyttäjät varmistavat, että ohjelma ja siihen liittyvät dokumentaatiot täyttävät määritetyt vaatimukset. (Homès 2011, 58–66.)

Kehittämismallista riippuen testaustasot ja niiden nimitykset voivat hieman vaihdella mallien välillä. Seuraavissa osioissa käydään läpi ohjelmistotestauksessa yleisesti käytettyä jakaumaa, jossa testaus on jaettuna neljään eri tasoon (Kuvio 4).



Kuvio 4. Sovellustestauksen testaustasot

4.2 Yksikkötestaus

Ensimmäisellä testaustasolla on yksittäisten komponenttien testaus eli yksikkötestaus (*unit testing*), jota kutsutaan myös komponentti- ja moduulitestaukseksi. Yksikkötestien tarkoituksena on todentaa vaatimuksien pohjalta komponentin toimivuus itsenäisenä yksikkönä. Testit ovat pääsääntöisesti sidottuina lähdekoodiin, minkä takia yleensä ohjelmoijat itse suunnittelevat yksikkötestit tai suunnittelevat ne yhteistyössä testaajien kanssa. Testattava komponentti voi olla esimerkiksi moduuli, luokka tai metodi. (Homès 2011, 59–60.)

Yksittäisten komponenttien testit usein suoritetaan eristetysti muista riippuvuuksista ja muusta järjestelmästä, jotta testit eivät ota kantaa muiden komponenttien toimivuuteen. Eristetysti yksikkö voidaan testata käyttämällä mock-objekteja ja testitynkiä, joiden avulla simuloidaan riippuvuuden odotettua toiminnallisuutta. Tällöin yksikköä voidaan testata, vaikka sen vaatimat riippuvuudet olisivat vielä kehitysvaiheessa. (Homès 2011, 59–60.)

4.3 Integraatiotestaus

Seuraavalla tasolla yksikkötestauksen jälkeen sijaitsee integraatiotestaus (*integration testing*). Integraatiotesteissä keskitytään kahden tai useamman komponentin välisen rajapinnan testaukseen. Näihin luetaan mukaan myös ohjelman rajapinnat käyttöjärjestelmään, tiedostojärjestelmään, tietokantoihin ja laitteistoon. Integroinnin osana olevien yksittäisten komponenttien toimivuutta ei enää tässä vaiheessa testata, vaan tämänlaiset testit tulisi suorittaa komponenttien yksikkötesteissä. (Homès 2011, 60–62.)

Integraatiota voidaan ohjelman kehityksen aikana toteuttaa usealla eri tavalla, joissa myös testaamisen toteutus muuttuu. Yksi tapa on integroida suurin osa tai kaikki komponentit yhtäaikaisesti, joka helposti aiheuttaa useita virheitä, joita on hankala paikantaa ja korjata testauksen aikana. Hyötynä on, ettei testitynkiä ole pakko käyttää integraatiotestauksessa, koska kaikki komponentit ovat valmiita ja toiminnallisia. Toinen tapa on integroida osat yksi kerrallaan pienimmästä isoimpaan. Tämä tapa voidaan myös kääntää toisinpäin, jolloin aloitetaankin integroiminen isoimmasta osasta. (Homès 2011, 60–62.)

Nykyään ohjelmistokehityksessä ollaan kuitenkin enemmän siirrytty jatkuvaan integraatioon (*continuous integration*). Jatkuvassa integraatiossa pyritään integroimaan muutoksia ja uusia ominaisuuksia mahdollisimman usein jakaen integroimisen ja integraatiotestauksen pienempiin osiin. Pienemmät osat ovat helpompi integroida ja testata sekä virheet ovat nopeampi löytää ja korjata. (Blundell & Torres Milano 2015, 157–158.)

4.4 Järjestelmätestaus

Integraatiotestauksesta seuraava taso on järjestelmätestaus (*system testing*). Järjestelmätestaus voidaan suorittaa, kun kaikki komponentit ovat integroitu onnistuneesti ja integraatiotestauksesta löydetyt virheet ovat korjattuina. Järjestelmätestauksessa testataan kokonaista järjestelmää mukaan lukien siihen liittyvät dokumentaatiot, kuten asennus-, käyttö- ja ylläpito-ohjeet. Testauksen aikana keskitytään vahvistamaan, että ohjelma täyttää suunnitellut vaatimukset ja määrittelyt. Useimmiten testaus toteutetaan mustalaatikkotestauksena testiryhmien toimesta. (Homès 2011, 62–64.)

Järjestelmän toiminnallisuuksien testauksen lisäksi suoritetaan ei-toiminnallisten vaatimuksien ja ominaisuuksien testausta, kuten suorituskyky-, turvallisuus-, käytettävyys-, luotettavuus- ja stressitestausta. Näiden testien perusteella varmistetaan, että järjestelmä operoi hyväksyttävällä tasolla. Tärkeätä on varmistaa turvallisuustestauksen avulla, että luvattomat henkilöt tai järjestelmät eivät pysty lukemaan tai muokkaamaan suojattuja tietoja. Suorituskykytestauksessa tarkastellaan suorituskyvyn vaihtelevuutta järjestelmän suorituksen aikana ja varmistetaan, ettei suorituskyky laske liian alhaiseksi missään vaiheessa. (Homès 2011, 62–64, 68–70.)

4.5 Hyväksyntätestaus

Viimeisenä testaustasona on hyväksyntätestaus (*acceptance testing*). Hyväksyntätestaus tapahtuu, kun järjestelmätestaus on suoritettu onnistuneesti ja järjestelmä on todettu olevan hyväksyttävällä tasolla julkaisua varten. Järjestelmätestauksen tavoin myös hyväksyntätestauksessa testataan kokonaista järjestelmää ja siihen liittyviä dokumentaatioita varmistaen niiden täyttävän määritetyt vaatimukset. Julkaisua usein edeltävät alfa- ja betatestaukset ovat myös hyväksyntätestausta. (Homès 2011, 64–66.)

Hyväksyntä- ja järjestelmätestauksessa testataankin paljon samoja asioita samalla tavalla, mutta hyväksyntätestauksen suorittaa testiryhmän sijasta asiakkaan edustajat tai järjestelmän loppukäyttäjät. Päätavoite hyväksyntätestauksessa ei enää ole varsinaisesti löytää ongelmia tai vikoja, vaan yrittää kasvattaa asiakkaan tai loppukäyttäjän luottamusta järjestelmään ja sen toimivuuteen. Useiden vikojen löytäminen hyväksyntätestauksen aikana voi vaikuttaa tähän negatiivisesti. (Homès 2011, 64–66.)

5 ANDROID-SOVELLUSTESTAUS

5.1 Testauskohteet

Osiossa 2.3 käytiin yleisesti läpi sovellustestauksen kohteet. Seuraavissa kappaleissa käydään läpi, mitä näiden lisäksi erityisesti kannattaa ottaa huomioon Android-sovelluksen testaamisessa.

Android-sovelluksesta tulisi testata aktiviteetit, että ne käyttäytyvät halutulla tavalla elinkaaren eri vaiheissa. Kaikkia elinkaaren tapahtumia, kuten aktiviteetin tuhoutumista tai pysähtymistä, ei ole välttämätöntä testata, jos näiden tapahtumien käsittelylogiikkaa ei ole muutettu. Aktiviteetin reagoiminen tapahtumaan tulee testata esimerkiksi, kun halutaan tallentaa aktiviteetin tila aktiviteetin pysähtyessä tai tuhoutuessa, ja palauttaa tila takaisin aktiviteetin uudelleen käynnistyessä. Testauksessa tulee ottaa huomioon myös aktiviteetin ajonaikaiset konfiguraatiomuutokset, koska osa näistä aiheuttaa aktiviteetin uudelleen käynnistymisen. (Blundell & Torres Milano 2015, 31–32.)

Tietokanta- ja tiedostojärjestelmäoperaatiot pitää testata, jotta voidaan varmistua siitä, että operaatiot ja niistä nousevat mahdolliset virheet käsitellään oikein. Nämä testit olisivat hyvä pyrkiä suorittamaan eristetysti välttäen oikeita kutsuja tiedostojärjestelmään tai ulkoisiin tietokantoihin käyttämällä mock-objekteja tai väliaikaisia korvikkeita. (Blundell & Torres Milano 2015, 32.)

Pitkäaikaiset taustalla pyörivät palvelut kannattaa testata, että ne toimivat odotetulla tavalla. Palveluiden testien tulisi varmistaa, että palvelut varmasti käynnistyvät ja toimivat oikein käynnistämisen jälkeen. Tärkeä asia on myös testata palvelun lopettaminen, ettei palvelua lopeteta liian aikaisin eikä se jää turhaan taustalle pyörimään. (Cruz & Niñirola 2014, 144.)

Usein toteutettu ominaisuus Android-sovelluksissa on tietojen tallentaminen, noutaminen ja jakaminen Androidin sisällöntarjoajien avulla. Sisällöntarjoajien testaaminen on suositeltavaa, varsinkin jos näiden avulla halutaan jakaa tietoja muiden sovelluksien käytettäväksi. Erityisesti kannattaa huolehtia, että muut sovellukset pääsevät käsiksi vain haluttuihin tietoihin. Testaus on parasta toteuttaa

eristetysti muusta järjestelmästä. (Cruz & Niñirola 2014, 144.) Eristetyn testin kirjoittamiseen kannattaa käyttää Androidin ProviderTestCase2-luokkaa, joka huolehtii, etteivät testit häiritse järjestelmän toimintaa tai muokkaa laitteeseen tallennettuja tietoja (Android Developers 2018l).

Android-järjestelmää käyttäviä laitteita on laidasta laitaan useilta eri valmistajilta. Laitteiden fyysiset ominaisuudet, kuten näytön koko ja sensorit, vaihtelevat laitteiden välillä laajalti. Sovelluksen toimivuutta ei voida varmistaa testaamalla vain yhdellä laitteella, vaan sovellusta tulee testata mahdollisimman monella erilaisella laitteistolla. Testaus voidaan helpoiten suorittaa käyttämällä suurimmaksi osaksi Android-emulaattoria muutaman fyysisen laitteen lisäksi.

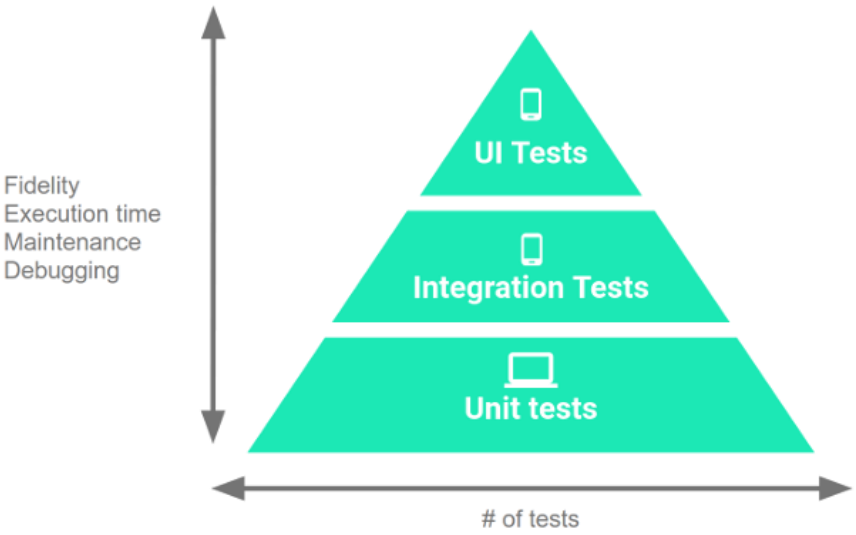
Laitteiden ominaisuuksien käyttötarve vaihtelee sovelluksien välillä, joten jokaisessa projektissa kannattaa pohtia mitkä laitteiden ominaisuudet oikeasti vaikuttavat sovelluksen toimivuuteen. Esimerkiksi GPS:n poissaolo ei vaikuta sovelluksen toimivuuteen, jos paikannusta ei ollenkaan käytetä hyväksi. Testauskokoonpanoja suunnitellessa kannattaa ottaa huomioon seuraavat ominaisuudet:

- verkkoyhteydet eli langattoman ja mobiiliverkon käyttömahdollisuus
- näyttöjen eri koot, pikselitiheydet ja resoluutiot
- käytössä olevat sensorit, kuten kiihtyvyysanturi ja gyroskooppi
- GPS:n käyttömahdollisuus paikantamiseen
- ulkoiset syötelaitteet
- ulkoisen muistin käyttäminen (SD-kortti). (Blundell & Torres Milano 2015, 32.)

5.2 Testaustasot ja testikategoriat

5.2.1 Testauspyramidi

Testauspyramidi on sovellustestauksessa käytetty malli kuvaamaan ohjelmallisten testien tasoja ja niiden määriä testien kokonaismäärästä (Kuvio 5). Pyramidimalli voi olla hieman erilainen riippuen testattavasta sovelluksesta ja sovelluksen käyttöalustasta. Android-testauksessa testauspyramidi on jaettu kolmeen tasoon: Yksikkötestit, integraatiotestit ja käyttöliittymätestit. Pyramidin alimmalla tasolla ovat yksikkötestit, jotka ovat tarkoitettua yksittäisten pienien itsenäisten komponenttien tai ominaisuuksien testaamiseen. Seuraavalla tasolla ovat integraatiotestit, jotka testaavat useamman ominaisuuden välistä toimivuutta. Pyramidin huipulla ovat käyttöliittymätestit, joiden avulla testataan käyttöliittymän toimivuutta. Kiivetessä pyramidia ylöspäin testien määrä pienenee ja testien ajamiseen, testaukseen ja ylläpitoon käytetty aika kasvaa. (Android Developers 2018g.)



Kuvio 5. Testauspyramidi (Android Developers 2018g)

Androidin ohjelmallisessa testauksessa testit jaetaan myös kolmeen eri kategoriaan: pienet testit (*small tests*), keskikokoiset testit (*medium tests*) ja isot testit (*large tests*). Pienet testit ovat yksikkötestejä, keskikokoiset testit integraatiotestejä ja isot testit integraatio- ja käyttöliittymätestejä. Testikategorioiden eri omi-

naisuuksien vuoksi testauksen tulisi sisältää jokaisen kategorian testejä. Sovelluksesta riippuen eri kategorioiden testien määrä voi vaihdella, mutta Googlen suositusten mukaan yleisesti testeistä noin 70 prosenttia tulisi olla pieniä, 20 prosenttia keskikokoisia ja kymmenen prosenttia isoja testejä. (Android Developers 2018g.)

5.2.2 Yksikkötestit

Android-sovelluksen testauksessa yksikkötesteillä testataan yksittäisten itsenäisten komponenttien toimivuutta riippumatta muista komponenteista tai produktiojärjestelmistä. Yksikkötestit useimmiten ajetaan eristettyinä mockaamalla eli matkimalla komponentin riippuvuuksia muihin komponentteihin, jolloin testit eivät ota kantaa muiden komponenttien toimivuuteen. Mockaamisen avulla pystytään myös simuloimaan riippuvuudet Android-ympäristöön, jolloin testejä ei tarvitse ajaa Android-laitteella. Usein mockaamisen helpottamiseksi ja nopeuttamiseksi käytetään erillisiä kirjastoja, joista suosituin on Mockito. (Android Developers 2018g.)

Yksikkötestaukseen käytetään pääosin JUnit-testauskirjastoa. Yksikkötestit ovat tarkoitettu ajettavaksi kehityskoneella JVM:ssä ilman emulaattoria tai fyysistä laitetta, koska testien ajaminen JVM:ssä on huomattavasti nopeampaa. Joissakin tapauksissa yksikkötestit ovat pakko suorittaa Android-laitteessa, koska testattavissa ominaisuuksissa voidaan tarvita Android-ympäristön riippuvuuksia, joiden mockaaminen on mahdotonta tai liian monimutkaista. Android-laitteella ajettavia yksikkötestejä kutsutaan instrumentaatioyksikkötesteiksi. (Android Developers 2018g.)

5.2.3 Integraatiotestit

Integraatiotestien avulla testataan kahden tai useamman komponentin välistä toimivuutta yhdessä. Testeillä lisäksi varmistetaan, että komponentit toimivat odotetusti myös emulaattorissa tai fyysisessä laitteessa, varsinkin jos komponentilla on riippuvuuksia Android-ympäristöön. Koko sovelluksen toimivuutta integraatiotestien avulla ei kuitenkaan pystytä todentamaan. (Android Developers 2018g.)

Yksikkötestauksen tapaan myös integraatiotestaus suoritetaan pääosin käyttämällä JUnit-testauskirjastoa. Toisin kuin yksikkötestit, integraatiotestit ajetaan emulaattorissa tai fyysisessä laitteessa. Testien ajaminen on usein järkevämpää emulaattorissa, koska sen avulla pystytään helpommin ja nopeammin testaamaan sovelluksen toimivuutta eri laitteistoilla ja näyttökoilla. (Android Developers 2018g.)

5.2.4 Käyttöliittymätestit

Yksikkö- ja integraatiotestit testaavat komponenttien toimivuutta ja niiden välisiä integraatioita, mutta nämä testit eivät testaa sovelluksen käytettävyyttä loppukäyttäjän näkökulmasta. Käyttöliittymätesteillä varmistetaan sovelluksen käyttäytyvän odotetulla tavalla käyttäjän suorittaessa toimintoja. Toimintoja ovat esimerkiksi napin painallukset ja tekstikenttien täyttäminen. Tärkeä osa-alue käyttöliittymätesteissä on käyttöliittymän oikeanlainen muokkautuminen käyttäjäinteraktioiden vaikutuksesta. Testataan esimerkiksi, että käyttöliittymän painike on näkyvissä ja sitä painaessa tallennetaan puhelimen tietokantaan tietoja, minkä jälkeen käyttäjälle näytetään ilmoitus onnistuneesta tallennuksesta. (Android Developers 2018c.)

Android-sovellukset usein käyttävät hyväksi muita laitteeseen asennettuja sovelluksia esimerkiksi kuvien ottamiseen, yhteystietojen hakemiseen ja tietojen jakamiseen. Näissä käyttötapauksissa käyttäjä ohjataan väliaikaisesti toiseen sovellukseen. Usean sovelluksen kattavia ominaisuuksia voidaan myös testata käyttöliittymätestien avulla. (Android Developers 2018c.)

Käyttöliittymätestaukseen käytetään yleisesti Googlen Espresso- ja UI Automator-kirjastoja. Integraatiotestien tavoin myös käyttöliittymätestit ajetaan emuloidussa tai fyysisessä laitteessa. Emulaattorin avulla voidaan tarkemmin testata käyttöliittymän toimivuutta eri laitteilla, varsinkin jos erikokoisille laitteille on rakennettu omat käyttöliittymät. (Android Developers 2018g.)

5.3 Android-testaustyökalut ja -kirjastot

5.3.1 Testaustyökalujen valinta

Android-testaukseen on saatavilla useita erilaisia testaustyökaluja ja -kirjastoa, joilla on omat käyttökohteensa. Harvoin vain yhdellä työkalulla pystytään testaamaan laajasti sovellusta, koska useimmat työkalut ovat kohdennettuja tiettyyn asiaan.

Työkalut valitaan projektikohtaisesti tarpeen mukaan ottaen huomioon sovelluksen laajuuden, käyttötapaukset ja kehityksessä käytettävän kehitysmallin. Tässä opinnäytetyössä tarkastelen erityisesti seuraavia Googlen suosittelemia työkaluja ja kirjastoja: JUnit, Espresso, Mockito ja Monkey. Nopeasti myös esittelen UI Automator ja Robolectric työkalut, mutta en yksityiskohtaisesti käy läpi niiden toimintaa ja käyttöä.

5.3.2 Testaustyökalujen käyttöönotto

Testaustyökalut asennetaan Gradlen avulla määrittelemällä halutut testaustyökalut projektin app-kansiossa olevaan build.gradle-tiedostoon, joka sisältää projektin riippuvuudet ja ”ohjeet” projektin koontiin. Työkalujen nimet ja versiot määritellään tiedostossa dependencies-osion sisään. Android-ympäristöä vaativat testaustyökalut, kuten Espresso, määritellään androidTestImplementation-etuliitteellä ja muut työkalut testImplementation-etuliitteellä (Kuvio 6). Lisäämisen jälkeen Gradle lataa riippuvuudet automaattisesti projektin koonnin yhteydessä. Jotta Android-laitteella ajettavat testit toimivat oikein, tulee varmistaa, että AndroidJUnitRunner on asennettuna ja käyttöönotettuna (Kuvio 6).

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 26
    defaultConfig {
        applicationId "com.example.company.myapplication"
        minSdkVersion 15
        targetSdkVersion 26
        versionCode 1
        versionName "1.0"
        //AndroidJUnitRunnerin käyttöönotto
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:26.1.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.0'
    //Android-laitteessa tai kehityskoneella ajettava testaustyökalu
    testImplementation 'junit:junit:4.12'
    //Android-laitteessa ajettava testaustyökalu
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    //AndroidJUnitRunner riippuvuuden lataaminen
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    //Androidin kustomi JUnit säännöt (rules)
    androidTestImplementation 'com.android.support.test:rules:1.0.2'

}
```

Kuvio 6. Testaustyökalujen määrittäminen build.gradle-tiedostossa

Testitiedostot ovat tavallisia Java-tiedostoja, jotka sijaitsevat kahdessa eri kansiossa riippuen testin suoritusympäristöstä. JVM:ssä suoritettavat yksikkötestit sijaitsevat src/test/java -kansiossa. Näitä testejä ovat useimmiten vain normaalit JUnit-testit. Instrumentaatioyksikkö-, integraatio- ja käyttöliittymätestit tulevat olla src/androidTest/java -kansiossa, koska nämä testit ajetaan emulaattorissa tai fyysisessä laitteessa. Täten Espressoa tai UI Automatoria käyttävät testit tulee sijoittaa tähän kansioon. Lisäksi Android-laitteessa ajetuissa testiluokissa tulee olla @RunWith(AndroidJUnit4.class)-annotaatio, mikä varmistaa testien oikeanlaisen suorittamisen.

5.3.3 JUnit

JUnit on suosituin ja eniten käytetty yksikkötestauskirjasto Java-pohjaisille sovelluksille. Työkalu on myös käytetyin testaustyökalu Android-sovelluksien testauksessa ja on oletuksena käytettävissä projektin luomisen jälkeen. Uusin versio JUnitista on JUnit 5. (JUnit 2018b.) Android-kehityksessä on kuitenkin vielä käytössä vanhempi 4-versio (Android Developers 2018d). Liitteessä 1 on esimerkki JUnit-testiluokasta.

JUnit4-testiluokka voi sisältää yhden tai useamman testimetodin, jotka suoritetaan automaattisesti testejä ajettaessa. Normaalit metodit merkataan testimetodeiksi @Test-annotaatiolla (Kuvio 8). Jokaisen testin tulisi testata vain yhtä komponentin toiminnallisuutta. Toiminnallisuuden testaaminen perustuu vertailuihin (assertions), joiden avulla verrataan saatua arvoa odotettuun arvoon (Kuvio 7). Saadun arvon ja odotetun arvon ollessa samat, ominaisuus toimii odotetulla tavalla. Vertailuja voi olla yksi tai useampi yhden testimetodin sisällä. Jos yksikin näistä vertailuista ei pidä paikkaansa, testi luokitellaan epäonnistuneeksi. JUnitin vertailumetodeita on yhteensä yhdeksän, joita ovat esimerkiksi assertTrue(), assertEquals() ja assertNotNull(). Perusvertailujen lisäksi valikoimaa monesti laajennetaan käyttämällä Hamcrest-kirjastoa, jonka avulla pystytään luomaan monipuolisempia ja helpommin luettavia vertailuja. (Android Developers 2018d.)

```
//Tarkistetaan, että ilmentymä ei ole NULL
assertNotNull(result);

//Tarkistetaan, että saatu tulos ja odotettu arvo ovat samoja
assertEquals(result, expected);
//TAI
assertThat(result, is(expected));
```

Kuvio 7. JUnit-kirjaston vertailumetodien käyttäminen

Testiluokassa voidaan käyttää @Before- ja @After-annotaatioita, joiden avulla voidaan suorittaa haluttuja metodeita ennen tai jälkeen jokaisen testimetodin (Kuvio 8) (Github 2018a). Metodien avulla voidaan esimerkiksi alustaa testeissä tarvittavat muuttujat tai sulkea avoinna oleva tiedosto. Testeistä saadaan luettavampia ja helpommin ylläpidettäviä siirtämällä testeissä usein toistuvat alku- ja lopputoimenpiteet näiden metodien sisälle.

```
//Kutsutaan ennen jokaista @Test-annotaatiolla merkittyä testimetodia
@Before
public void setUp() {
    //Tehdään jotakin
}

//Kutsutaan jokaisen @Test-annotaatiolla merkityn testimetodin jälkeen
@After
public void tearDown() {
    //Tehdään jotakin
}

@Test
public void testSomething() {
    //Testataan jotakin
}
```

Kuvio 8. JUnitin @Before- ja @After-annotaatioiden käyttö

Android-sovelluksen testauksessa käytetään hyväksi Androidin JUnit-sääntöjä (*rule*) esimerkiksi taustapalveluiden ja käyttöliittymätestien testaamisen helpottamiseksi (Android Developers 2018h). Säännöillä muutetaan testien normaalia suoritusta. Sääntöjen avulla lisätään lisätarkistuksia testeihin, raportoidaan testin suoritusta ja suoritetaan tarvittavia resurssien alustuksia ja siivouksia. Testiluokassa sääntömuuttujat merkitään @Rule-annotaatiolla, jonka perusteella JUnit ottaa sääntöjen logiikan huomioon testejä ajettaessa. (JUnit 2018a.)

5.3.4 Mockito

Mockito on Java-sovelluksien testauksessa käytetty avoimen lähdekoodin mockauskirjasto, jonka avulla testeissä pystytään testattavan ominaisuuden riippuvuudet matkimaan mock-objekteilla, jotka simuloivat luokkien toiminnallisuutta. Matkimisen avulla saadaan eristettyä muiden luokkien ja rajapintojen toiminnallisuus testien ulkopuolelle, jolloin testit eivät epäonnistu muiden komponenttien virheiden takia. Mockito-kirjastoa käytetään usein Android-sovelluksen yksikkötesteissä erottaakseen yksikkötestit Android-järjestelmästä. Liitteessä 2 on esimerkki JUnit-testiluokasta, jossa käytetään Mockitoa. (Mockito 2018.)

Mock-objekti on yleisesti käytetty nimitys testien aikana käytettäville sijaisobjekteille (test double). Oikeasti mock-objekti on yksi sijaisobjektin tyyppi. Sijaisobjektien eri tyyppejä ovat dummy, fake, stub, mock ja spy, jotka ovat selitettynä alla:

- **Dummy**-objekti ei sisällä minkäänlaista toimintalogiikkaa vaan sen ainoa tarkoitus on saada koodi kääntymään. Usein dummy-objekteja käytetään vain metodikutsuissa parametreinä.
- **Fake**-objektilla on yksinkertainen toimiva implementaatio, mutta se ei ole tuotantovalmis. Fake-objekti voi esimerkiksi käyttää varsinaisen tietokannan sijasta väliaikaista muistissa olevaa tietokantaa.
- **Stub**-objekti on tynkäobjekti, jolla on ennalta määrätyt vastaukset testauksen aikana suoritettuihin metodikutsuihin.
- **Mock**-objektilla on stub-objektin tavoin ennalta määrätyt vastaukset, mutta niiden lisäksi myös odotuksia mitä metodeita kutsutaan, kuinka usein ja millä parametreillä.
- **Spy**-objekti on stub-objektin kaltainen objekti, joka ennalta määrättyjen vastauksien lisäksi tallentaa tietoja objektin käytöstä. (Grzejszczak 2014, 23–24.)

Mockitoa käyttävien testiluokkien tulee sisältää @RunWith(MockitoJUnitRunner.class)-annotaatio, jotta Mockito automaattisesti alustaa mock-objektit sekä validoi, että kirjastoa käytetään oikealla tavalla (Kuvio 9). Normaali testiluokan muuttuja voidaan muuttaa mock-objektiksi @Mock-annotaatiolla, jonka avulla Mockito ymmärtää alustaa muuttujan (Kuvio 9). (Android Developers 2018d.)

```
@RunWith(MockitoJUnitRunner.class)
public class TestedClassTest {

    //Mock-objektin luominen.
    //Luokkaa ei tarvitse enää itse alustaa
    @Mock
    MockedClass mockedClass;

    //Mockito alustaa testattavan luokan automaattisesti
    //ja injektoi luokkaan MockedClass-luokan mock-objektin
    @InjectMocks
    TestedClass myClassUnderTest;
```

Kuvio 9. Mock-objektien luominen ja injektoiminen

Mock-objektien välittäminen testattavalle luokalle voidaan tehdä manuaalisesti tai käyttämällä Mockiton injektointia. Manuaalisesti objektit välitetään luokalle luokan konstruktoreiden, set-metodien tai perusmetodien avulla. Koodin vähentämiseksi ja selkeyttämiseksi voidaan käyttää @InjectMocks-annotaatiota, joka annetaan testattavan luokan muuttujalle (Kuvio 9). Mockito tällöin yrittää injektoida testiluokassa määritetyt mock-objektit joko konstruktoreiden, set-metodien tai kenttäinjektion avulla. Kenttäinjektiossa Mockito tutkii testattavan luokan jäsenmuuttujia. Injektoimisen Mockito määrittää testiluokan ja testattavan luokan muuttujien tyyppien ja nimien mukaan. Huomioitavaa on, että epäonnistunut injektoiminen ei aiheuta virhettä. (Javadoc 2018.)

Metodikutsu mockataan when() ja thenReturn() metodien avulla, eli kun metodia kutsutaan tietyillä parametreillä, niin palautetaan tietty ennalta määritetty arvo (Kuvio 10). when()-metodille annetaan parametrinä odotettu metodikutsu odotetuilla parametreillä. Palautusarvo annetaan parametrinä thenReturn()-metodille. (Android Developers 2018d.)

```
//Mockataan mockedClass-muuttujan getSomething()-metodi
when(mockedClass.getSomething()).thenReturn("Something");

//Mockataan mockedClass-muuttujan getPersonByName()-metodi
//kun metodia kutsutaan tietyllä parametrillä
Person matt = new Person( name: "Matt Michaels");
when(mockedClass.getPersonByName("Matt Michaels")).thenReturn(matt);
```

Kuvio 10. Mock-objektien metodikutsujen mockaaminen

JVM:ssä suoritetuissa testeissä jokainen metodikutsu Android-järjestelmään aiheuttaa virheen, jos metodikutsua ei ole mockattuna. Virheilmoituksen perusteella pystytään helposti paikantamaan virheen aiheuttanut metodikutsu (Kuvio 11). (Android Developers 2018d.)

```
java.lang.RuntimeException: Method d in android.util.Log not mocked. See http://g.co/androidstudio/not-mocked for details.

    at android.util.Log.d(Log.java)
    at com.testing.arto.unitconverter.Unit.parseUnitType(Unit.java:27)
    at com.testing.arto.unitconverter.Unit.<init>(Unit.java:14)
    at com.testing.arto.unitconverter.UnitClassTest.testThatUnitClassParsesTypeCorrectly(UnitClassTest.java:27) <27 internal calls>
```

Kuvio 11. ”Method not mocked” -virheilmoitus

Mockitolla pystytään objektien simuloimisen lisäksi varmistamaan, että mockattujen objektien metodeita kutsuttiin odotetulla tavalla. Voidaan esimerkiksi varmistaa, että objektin metodia kutsuttiin kaksi kertaa testien aikana tietyillä parametreillä. Varmistamiseen käytetään Mockiton verify()-metodia (Kuvio 12). Metodi ottaa parametreinä tarkastelussa olevan mock-objektin ja odotettujen kutsujen määrä. Vakiona määrä on yksi, jolloin testi epäonnistuu, jos testien aikana metodia ei kutsuta ollenkaan tai sitä kutsutaan useammin kuin kerran. Määrää ei anneta suoraan numerona, vaan käytetään metodeita, kuten times(x) ja atMost(x) (Kuvio 12). times(x)-metodin avulla varmistetaan, että metodia kutsuttiin tasan x-määrä, ja atMost(x)-metodilla varmistetaan kutsujen määrän olevan enintään x-määrä. (Grzejszczak 2014, 175–177, 183–184.)

```
//Varmistetaan, että mockatun luokan getSomething()-metodia...

//...kutsuttiin tasan 1 kertaa
verify(mockedClass, times( wantedNumberOfInvocations: 1)).getSomething();
//TAI (times(1) on vakio)
verify(mockedClass).getSomething();

//...kutsuttiin enintään 3 kertaa
verify(mockedClass, atMost( maxNumberOfInvocations: 3)).getSomething();

//...kutsuttiin vähintään kerran
verify(mockedClass, atLeast( minNumberOfInvocations: 1)).getSomething();

//Varmistetaan, että mockatun luokan getPersonByName()-metodia
//kutsuttiin tasan 1 kertaa parametrillä "Matt Michaels"
verify(mockedClass).getPersonByName("Matt Michaels");
```

Kuvio 12. Mock-objektien metodikutsujen verifiointi

Mockitoa käyttäessä kannattaa pitää mielessä, että testauksen päätarkoitus on testata toiminnallisuuden lopputuloksia implementaation sijasta. Implementaation liiallista testausta tulisi vältellä, koska matkittavan komponentin koodin muuttuessa pitää myös testien koodi muuttaa vastaamaan toteutusta. Mockiton verify()-metodin avulla testataan nimenomaan toteutustapaa, mikä voi aiheuttaa virheellisiä testituloksia komponenttien muuttuessa. (Grzejszczak 2014, 41.)

Toinen tärkeä asia mikä kannattaa ottaa huomioon on muiden kuin omien luokkien ja rajapintojen matkiminen. Kolmansien osapuolien toteutukset voivat muuttua, minkä jälkeen niiden matkiminen vanhojen toteutuksien pohjalta voi johtaa virheelliseen positiiviseen tilanteeseen, jossa testit menevät läpi, vaikka ominaisuus ei ole enää toimiva. (Github 2018b.)

5.3.5 Espresso

Espresso on Googlen kehittämä ja ylläpitämä käyttöliittymätestauskirjasto, jonka avulla pystytään simuloimaan käyttöliittymäinteraktioita ja tarkistamaan, että käyttöliittymä on odotetussa tilassa. Espresson suurin vahvuus on testien synkronoiminen, jolloin Espresso osaa ajaa testit oikeaan aikaan varmistaen, että tarvittavat käyttöliittymäkomponentit ovat näkyvissä käyttöliittymässä ja asynkroniset AsyncTask-toimenpiteet ovat suoriutuneet. Muita asynkronisia toimenpiteitä Espresso ei osaa automaattisesti odottaa, mutta synkronoiminen voidaan tällöin toteuttaa itse IdlingResource-luokasta periytyvän luokan avulla. Automaattisen odottamisen avulla nopeutetaan testien kirjoittamista, vältytään manuaalisilta ajoituksilta sekä parannetaan testien luotettavuutta ja luettavuutta. (Android Developers 2018j.)

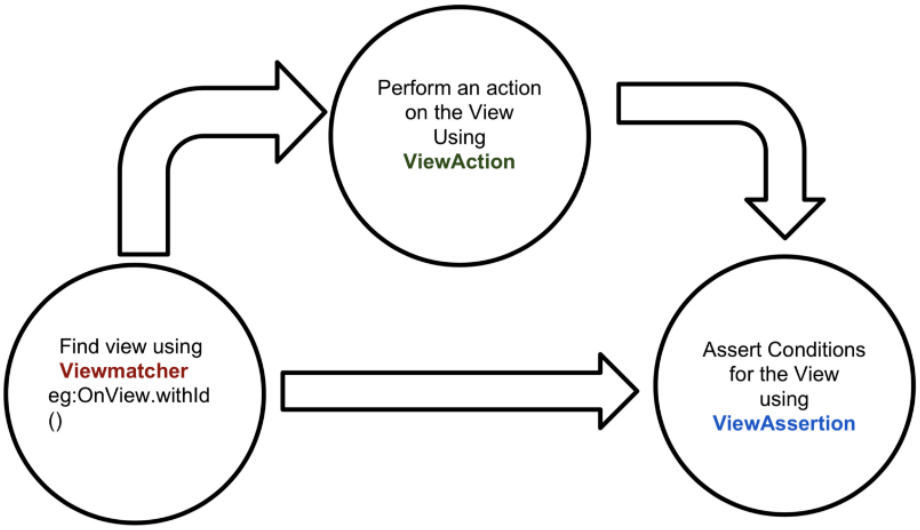
Ennen testien ajamista testauslaitteelta kannattaa ottaa animaatiot pois käytöstä, koska animaatiot voivat vaikuttaa testien synkronointiin aiheuttaen odottamattomia tuloksia tai epäonnistuneita testejä. Animaatioiden asetukset löytyvät asetuksien alta kehittäjäasetuksista (*developer settings*), jotka usein ovat vakiona piilotettuna. (Android Developers 2018j.)

Espresso-testien tarkoituksena on testata käyttöliittymää ja sen adoptointia eri tilanteissa. Useimmiten testeissä simuloidaan yksinkertaisia interaktioita, kuten nappien painalluksia, tekstin syöttämistä ja listakohtien valintoja. Optimaalisessa tilanteessa käyttöliittymätestit kattaisivat kaikki mahdolliset käyttötapaukset, jotka voivat vaikuttaa käyttöliittymään. (Android Developers 2018j.)

Espresson testiluokat kirjoitetaan samalla tavalla kuin normaalit JUnit4-testiluokat, joten testeissä voidaan edelleen käyttää JUnit-kirjaston ominaisuuksia ja tarkistuksia. Esimerkki Espresso-testistä on liitteessä 3, ja yleiset testeissä käytetyt toiminnot ovat kuvattuna liitteessä 4.

Espresso-käyttöliittymätesteissä seurataan kolmivaiheista kaavaa interaktioiden simuloimiseen ja lopputuloksien tarkistamiseen (Kuvio 13). Kaavan eri vaiheet ovat selitettynä alla:

1. Etsitään testattava käyttöliittymäkomponentti aktiviteetista. Staattisen komponentti etsitään onView()-metodilla, ja AdapterView-komponentin sisällä oleva komponentti etsitään onData()-metodin avulla.
2. Simuloidaan käyttäjän interaktioita käyttöliittymäkomponentin kanssa perform()-metodia kutsumalla. Metodille annetaan parametrina yksi tai useampi toiminta, jotka halutaan komponentille suorittaa.
3. Toistetaan 1. ja 2. vaiheet kaikille interaktioille, jotka komponenteille halutaan suorittaa simuloidakseen käyttötapausta.
4. Tarkistusmetodeilla tarkistetaan käyttöliittymän olevan tilassa, jossa sen oletetaan olevan vuorovaikutuksien jälkeen. (Android Developers 2018j.)



Kuvio 13. Espresso-testien kaava (Android Developers 2018j)

Ennen kuin käyttöliittymäkomponentteja voidaan etsiä ja simuloida, käyttöliittymän sisältävä aktiviteetti tulee olla luotuna. Aktiviteettia ei tarvitse manuaalisesti luoda ja käynnistää ennen jokaista testiä, vaan vähentääkseen koodin määrää voidaan käyttää ActivityTestRule-luokkaa, joka on Androidin JUnit-sääntö (Kuvio 14). Ennen jokaista testimetodia ActivityTestRule-luokka luo ja käynnistää aktiviteetin. Testien jälkeen aktiviteetti automaattisesti tuhotaan, jos aktiviteetti on vielä käynnissä. Aktiviteetti voidaan myös käynnistää ja tuhota manuaalisesti launchActivity()- ja finishActivity()-metodien avulla. (Android Developers 2018b.)

```
@Rule
public ActivityTestRule<MainActivity> activityRule =
        new ActivityTestRule<>(MainActivity.class);
```

Kuvio 14. ActivityTestRule-luokan käyttäminen testeissä

Aktiviteetin käynnistämisen jälkeen käyttöliittymäkomponentit ja näkymät etsitään onView()-metodilla (Kuvio 15). Käyttöliittymästä haetaan kerrallaan vain yksi näkymä. (Android Developers 2018e.)

```
//Haetaan näkymä R.id:n perusteella
onView(withId(R.id.welcome_message)).check(matches(isDisplayed()));

//Haetaan näkymä sisältämän tekstin perusteella
onView(withText(R.string.welcome_message)).check(matches(isDisplayed()));
//TAI
onView(withText("Welcome")).check(matches(isDisplayed()));
```

Kuvio 15. Näkymän hakeminen onView()-metodilla

Näkymän haun rajaamisessa käytetään ViewMatchers-luokan matchereita (*matchers*) (Kuvio 16). Matchereita ovat esimerkiksi withId() ja withText(), jotka yrittävät resurssitunnuksen (R.id) tai tekstisisällön perusteella rajata haun vain yhteen näkymään. Haku on virheellinen, jos näkymää ei löytynyt tai matchereita vastaavia näkymiä löytyi useampi kuin yksi. (Android Developers 2018e.)

Kuvio 16. Usein käytetyt Espresson matcherit käyttöliittymäkomponenttien etsimiseen (Android Developers 2018f)

AdapterView-luokasta periytyvissä näkymissä, kuten ListView-listassa ja GridView-ruudukossa, tiedot ladataan dynaamisesti, minkä vuoksi kaikki näkymän lapsielementit eivät välttämättä ole ladattuina. Lapsielementtien hakemiseen käytetään onView()-metodin sijasta onData()-metodia, joka varmistaa, että lapsielementti on ladattuna ja näkyvissä käyttöliittymässä (Kuvio 17). Koska listauksien lapsielementit ovat graafisesti melkein identtisiä ja niillä ei ole uniikkeja resurssitunnuksia, näkymä rajataan näkymän pohjatietojen mukaan. Haun rajaamiseen käytetään matchereista vain objektimatchereita, kuten instanceOf() ja startsWith() (Kuvio 16). (Android Developers 2018e.)

```
//Avataan pudotusvalikko
onView(withId(R.id.spinner_from_unit))
        .perform(click());

//Haetaan pudotusvalikosta oikea kohta ja klikataan sitä
onData(allOf(is(instanceOf(String.class)), is( value: "Inch")))
        .perform(click());
```

Kuvio 17. Listakohdan valitseminen pudotusvalikosta onData()-metodilla

Useimmiten näkymällä on uniikki resurssitunnus, jolloin etsiminen on helpointa withId()-matcherilla, mutta joissakin tapauksissa näkymällä ei ole resurssitunnusta ollenkaan tai useampi näkymä jakaa saman tunnuksen. Näissä tilanteissa joudutaan käyttämään muita matchereita. Matchereita voidaan myös yhdistellä käyttämällä allOf()-metodia, joka rajaa hakua kahden tai useamman matcherin perusteella (Kuvio 18). Hyvänä käytäntönä on käyttää näkymän hakemiseen mahdollisimman vähiten kuvailevaa matcheria. Esimerkiksi näkymän löytyessä pelkästään resurssitunnuksella, rajaamiseen ei tarvitse käyttää muita keinoja. Liian tarkat hakukriteerit hidastavat turhaan testien ajamista sekä pahentavat testien luettavuutta ja ylläpidettävyyttä. (Android Developers 2018e.)

```
onView(allOf(withId(R.id.welcome_message), withText(R.string.welcome_message)));
```

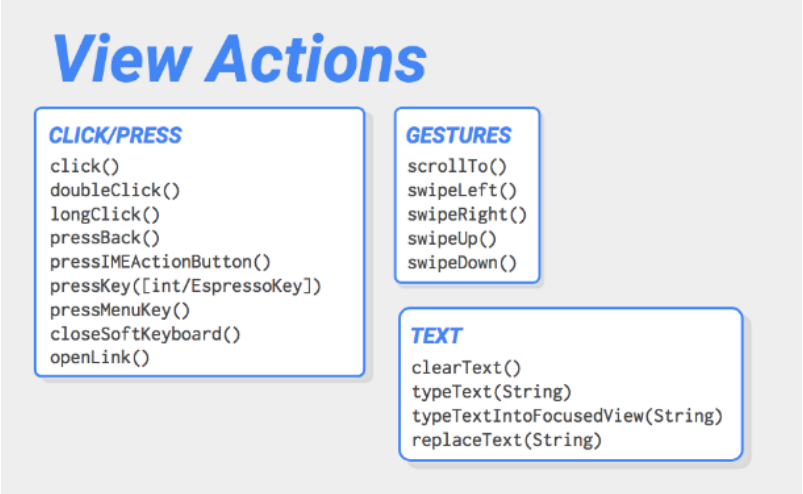
Kuvio 18. Näkymän hakeminen usean matcherin avulla

Käyttöliittymänäkymän löytämisen jälkeen näkymälle simuloidaan toimenpiteitä perform()-metodilla (Kuvio 19). Toimenpiteitä ovat erilaiset klikkaukset, painallukset, tekstikenttien interaktiot ja liike-eleet. (Android Developers 2018e.)

```
//Syötetään tekstikenttään tekstiä ja suljetaan laitteen näppäimistö
onView(withId(R.id.editText_city))
        .perform(typeText( stringToBeTyped: "Rovaniemi"), closeSoftKeyboard());

//Klikataan painiketta
onView(withId(R.id.btn_convert)).perform(click());
```

Kuvio 19. Toimenpiteiden simulointi käyttöliittymäkomponenteille

Toimenpiteet periytyvät ViewActions-luokasta, joita voidaan suorittaa yksi tai useampi peräkkäin (Kuvio 20). Usein käytettyjä toimintoja ovat click() ja typeText(), joilla simuloidaan näkymän klikkausta ja tekstikentän täyttämistä (Kuvio 19). Jos käyttöliittymäkomponentti on ScrollView-komponentin lapsielementti, ennen varsinaisia toimenpiteitä komponentille kannattaa suorittaa scrollTo()-toimenpide, joka varmistaa komponentin olevan näkyvissä. (Android Developers 2018e.)



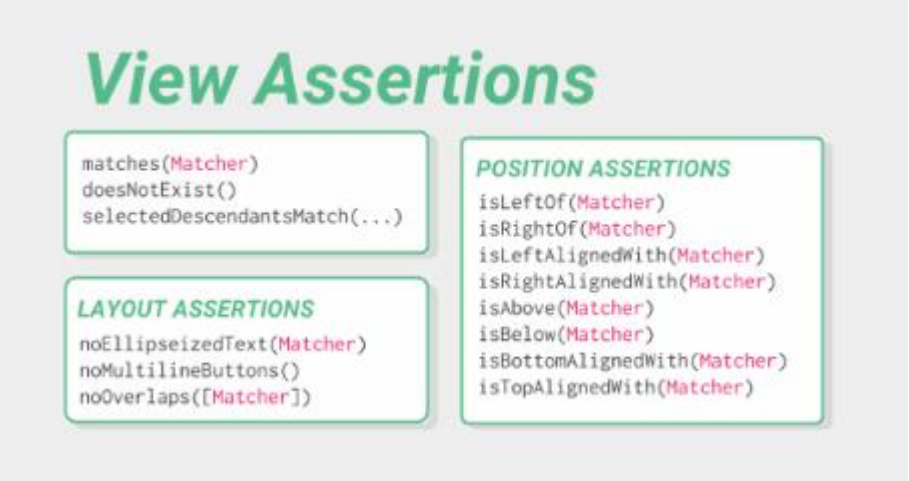
Kuvio 20. Usein käytetyt Espresson käyttöliittymäkomponenttien toimenpiteet (Android Developers 2018f)

Käyttäjäinteraktioiden simuloimisen jälkeen tarkistetaan, että käyttöliittymä on odotetussa tilassa. Tarkistukset käyttöliittymänäkymille suoritetaan käyttämällä check()-metodia (Kuvio 21). (Android Developers 2018j.)

```
//Tarkistetaan, että näkymä sisältää tietyn tekstin
onView(withId(R.id.welcome_message))
        .check(matches(withText("Welcome")));

//Tarkistetaan, että näkymän sisältämä teskti alkaa tietyllä tavalla
onView(withId(R.id.welcome_message))
        .check(matches(withText(startsWith("Welc"))));

//Tarkistetaan, että näkymä sisältää tietyn tekstin JA näkymä on näkyvissä
onView(withId(R.id.welcome_message))
        .check(matches(allOf(withText(R.string.welcome_message), isDisplayed())));
```

Kuvio 21. Näkymän tilan todentaminen check()-metodilla

Tarkistuksien määrittämiseen käytetään ViewAssertion-luokan vertailuja, kuten doesNotExist() ja matches() (Kuvio 22). doesNotExist() tarkastaa, että käyttöliittymässä ei ole tätä näkymää. Yleisin käytetty vertailu on matches(), joka mahdollistaa näkymän tilan tarkistamisen käyttäen aikaisemmin mainittuja ViewMatchers-luokan matchereita. (Android Developers 2018j.)



Kuvio 22. Usein käytetyt Espresson vertailut näkymän tilan tarkistamiseen (Android Developers 2018f)

5.3.6 Monkey

Monkey on komentorivipohjainen testityökalu, joka lähettää Android-laitteelle tai emulaattorille jonon näennäissatunnaisia tapahtumia, kuten painalluksia, koskетuksia ja eleitä, sekä lukuisia järjestelmätason tapahtumia. Työkalun tarkoituksena on rasittaa sovellusta ja tarkkailla kaatuuko sovellus tai ilmeneekö käsittelemättömiä poikkeuksia rasituksen aikana. Jos testin aikana tapahtuu virhe, Monkey pysäyttää testin ja tulostaa virheen tiedot, joiden avulla pystytään paikallistamaan ongelman syy (Kuvio 23). (Android Developers 2018m.)

```
// CRASH: com.testing.arto.greatapplication (pid 3991)
// Short Msg: java.lang.ArithmeticException
// Long Msg: java.lang.ArithmeticException: divide by zero
// Build Label: google/sdk_google_phone_x86/generic_x86:7.0/NYC/3761695:userdebug/dev-keys
// Build Changelist: 3761695
// Build Time: 1487893073000
// java.lang.ArithmeticException: divide by zero
//      at com.testing.arto.greatapplication.MainActivity.onBackPressed(MainActivity.java:53)
```

Kuvio 23. Monkey-testin virheilmoitus

Testityökalun testi voidaan ajaa komentoriviltä tai ohjelmaskriptin avulla. Yksinkertaisimmillaan testi ajetaan komennolla ”adb shell monkey 500”, jossa numero merkitsee haluttua tapahtumien määrää. Testille voidaan syöttää asetuksia, joiden avulla määritellään tarkemmin millä tavalla testit suoritetaan. Yleisin käytetty asetus on testattavan paketin määrittäminen. Vakiona Monkey lähettää tapahtumia kaikkiin laitteelle asennettuihin paketteihin, mutta useimmiten kuitenkin halutaan testien keskittyvän vain kehityksessä olevaan sovellukseen. Tällöin voidaan lisätä tapahtumamäärän eteen ”-p minun.paketin.nimi”, joka rajoittaa testin pääsyn vain määriteltyyn pakettiin/sovellukseen (Kuvio 24). (Android Developers 2018m.)

```
adb shell monkey -p com.testing.arto.greatapplication 500
```

Kuvio 24. Monkey-testin ajaminen komentoriviltä

Läpäisty Monkey-testi ilmaisee, että sovellus osaa käsitellä käyttäjä- ja järjestelmätapahtumat kaatamatta sovellusta. Testit eivät kuitenkaan ota kantaa sovelluksen konkreettiseen toimivuuteen eivätkä varmista, että sovellus ei kaatuisi joissakin erikoistapauksissa. Tästä huolimatta Monkey-testit kannattaa ajaa vähän väliä muiden testien lisäksi, koska se vähentää manuaalisia ”apinatestejä”, joissa kehittäjä tai testaaja tuottaisi samankaltaisia näennäissatunnaisia tapahtumia.

5.3.7 UI Automator

UI Automator on toinen Googlen kehittämä käyttöliittymätestaustyökalu, jonka avulla pystytään testaamaan sovelluksen toimivuutta myös tilanteissa, joissa testattava sovellus navigoi toiseen sovellukseen tai Android-järjestelmän käyttöliittymään. Sovelluksen käytön aikana voi tulla esimerkiksi tilanne, jossa halutaan navigoida tiettyyn Android-asetukseen tai hakea tietoja yhteystiedot-sovelluksesta. UI Automatorilla pystytään myös testaamaan pelkästään itse sovelluksen käyttöliittymää samaan tapaan kuin Espressolla, mutta näihin testeihin Espresso on parempi vaihtoehto nopeuden, käytettävyyden ja Android-versiorajoituksien kannalta. (Android Developers 2018k.)

Google suosittelee käyttämään UI Automatoria vain testatessa kriittisiä käyttötapauksia, joissa sovelluksen tulee olla vuorovaikutuksissa käyttöjärjestelmän käyttöliittymän kanssa. Usean sovelluksen välisen ominaisuuden testit tulisi ensisijaisesti pyrkiä hajottamaan pienempiin osiin ja testata yhtä ominaisuuden osaa kerrallaan. Tällöin yksi testi testaa tietojen lähettämisen ja toinen testi tiedon vastaanottamisen. (Android Developers 2018g.)

5.3.8 Robolectric

Robolectric on Android-järjestelmän emuloimiseen kehitetty testaustyökalu, jonka avulla voidaan helpommin kirjoittaa Android-riippuvuuksia tarvitsevien ominaisuuksien yksikkötestejä. Robolectric-testit ovat lähes yhtä tarkkoja ja täsmällisiä kuin Android-laitteella ajettavat testit, mutta huomattavasti nopeampia, koska testit ajetaan kehittämiskoneella JVM:ssä. Testeissä käytetään Robolectricin omia testausrajapintoja, joten testien kirjoittaminen eroaa huomattavasti muista aikaisemmin esitellyistä testeistä. (Android Developers 2018g.)

Työkalua käytetään erityisesti testivetoisessa kehityksessä, jossa testien nopea suoritus on tärkeätä koodin muuttamisen jälkeen. Robolectricin rinnalla ei tarvitse välttämättä enää käyttää erillistä mockaustyökalua Android-järjestelmän riippuvuuksien matkimiseen. Lisäksi testaustyyli voi olla lähempänä mustalaatikkotestausta, jolloin testit ovat ylläpidettävämpiä ja testit keskittyvät enemmän ominaisuuden toiminnallisuuteen kuin implementaatioon. (Robolectric 2018.)

6 POHDINTA

Android-sovelluksen testaaminen on tärkeä osa sovelluksen kehittämistä ja laadun varmistamista. Testaaminen ohjelmallisten testien avulla on hankala taito, jonka oppiminen vaatii aikaa ja harjoittelua. Testaustyökalujen ja -kirjastojen käyttäminen on suhteellisen suoraviivaista, mutta testien suunnittelu ja oikeiden menetelmien valitseminen ovat hankalia prosesseja.

Työssä kävin aluksi läpi yleisesti sovellustestauksen perusteet ja miten näitä perustietoja voidaan hyödyntää Android-sovelluksen testaamisessa. Toinen osa koostui Android-sovelluksen testaamisesta ja siihen käytetyistä työkaluista. Molemmista osa-alueista olisi voinut kirjoittaa vielä huomattavasti enemmän. Aikataulun ja aiheen laajuuden vuoksi en kyennyt käymään testaustyökalujen käyttämistä läpi yhtä tarkasti kuin olisin halunnut. Tarkoituksena oli kehittää yksinkertainen Android-sovellus, jonka ominaisuuksien testaamiseen olisin suunnitellut ja kirjoittanut testejä, joita olisin voinut käyttää opinnäytetyössä esimerkkeinä. Opinnäytetyössäni olisin halunnut tutkia myös mitä muita tapoja testauksessa voitaisiin käyttää ohjelmallisten testien lisäksi.

Android-sovelluksen testaamisessa tärkeimmät ja oleellisimmat työkalut ovat JUnit, Espresso ja Mockito. Jokaisessa projektissa tulisi näistä olla käytössä ainakin JUnit ja Espresso, jotka riittävät varsinkin pienemmissä sovelluksissa. Mockito ei täysin ole pakollinen, mutta suositeltava eristettyjen testien kirjoittamiseen. Näiden työkalujen lisäksi kannattaa käyttöön ottaa myös Monkey, jonka yksinkertaisien testien avulla voidaan helposti ja nopeasti löytää vakaviakin käytettävyysongelmia.

Mockito voidaan myös korvata kokonaan tai osittain käyttämällä Robolectriciä, varsinkin jos toteutetaan testivetoista kehittämistä tai sovelluksessa käytetään monimutkaisia Android-ominaisuuksia. Android-dokumentaatio sisälsi todella vähän tietoa tästä työkalusta, minkä vuoksi priorisoin muut työkalut sitä korkeammalle. Priorisoinnin myötä Robolectricin toiminnan tutkiminen opinnäytetyössä jäi vähäiseksi, minkä takia voin vain kehottaa kokeilemaan sen käyttämistä.

Käyttöliittymätestit voidaan myös tehdä UI Automatorin avulla, mutta Espresso on ehdottomasti parempi vaihtoehto käytettävyyden ja nopeuden kannalta. UI Automatoria tulisi käyttää vain tilanteissa, joissa muilla työkaluilla ei pystytä hyväksyttävä tasolla todentamaan ominaisuuden toimivuutta.

Yksi vaikeimmista testausvaiheista on varsinaisen ominaisuuden rakentaminen tarpeeksi joustavaksi, että sen toiminnan toimivuutta on helppoa testata. Tämän vuoksi vanhan ohjelmakoodin testaaminen on vaikeata, koska ohjelmakoodia ei ole luultavasti suunniteltu testattavaksi, ja ohjelmakoodin muokkaaminen testausystävälliseksi voi helposti hajottaa ominaisuuden toimivuuden.

Android-sovelluksen testauksen opettelua hidasti materiaalien vähyys. Aiheeseen liittyviä kirjoja on suhteellisen vähän ja sisältävät usein vanhentunutta tietoa. Internetissä olevat artikkelit usein toistavat toisiaan sisältäen samat asiat ja esimerkit. Yksinkertaisien ja vanhentuneiden esimerkkien pohjalta on hankala suunnitella ja kirjoittaa omia testejä. Päälähteenä käytin Androidin dokumentaatiota, josta löytyi ajantasaisin tieto ja parhaat esimerkit. Dokumentaatiosta tietojen etsiminen oli kuitenkin välillä aikaa vievää, koska saman osa-alueen tiedot olivat usein jaoteltu useaan eri paikkaan.

LÄHTEET

Android Developers 2018a. About the platform. Viitattu 26.4.2018 https://developer.android.com/about/.

Android Developers 2018b. ActivityTestRule. Viitattu 14.4.2018 https://developer.android.com/reference/android/support/test/rule/ActivityTestRule.html.

Android Developers 2018c. Automate user interface tests. Viitattu 12.4.2018 https://developer.android.com/training/testing/ui-testing/.

Android Developers 2018d. Building Local Unit Tests. Viitattu 8.4.2018 https://developer.android.com/training/testing/unit-testing/local-unit-tests.html.

Android Developers 2018e. Espresso basics. Viitattu 4.5.2018 https://developer.android.com/training/testing/espresso/basics.

Android Developers 2018f. Espresso Cheat Sheet. Viitattu 4.5.2018 https://developer.android.com/training/testing/espresso/cheat-sheet.html.

Android Developers 2018g. Fundamentals of Testing. Viitattu 11.4.2018 https://developer.android.com/training/testing/fundamentals.html.

Android Developers 2018h. JUnit4 rules with testing support library. Viitattu 3.5.2018 https://developer.android.com/training/testing/junit-rules.

Android Developers 2018i. Kotlin on Android FAQ. Viitattu 26.4.2018 https://developer.android.com/kotlin/faq.

Android Developers 2018j. Testing UI for a Single App. Viitattu 13.4.2018 https://developer.android.com/training/testing/ui-testing/espresso-testing.html.

Android Developers 2018k. Testing UI for Multiple Apps. Viitattu 13.4.2018 https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html.

Android Developers 2018l. Testing Your Content Provider. Viitattu 20.4.2018 https://developer.android.com/training/testing/integration-testing/content-provider-testing.html.

Android Developers 2018m. UI/Application Exerciser Monkey. Viitattu 17.4.2018 https://developer.android.com/studio/test/monkey.html.

Bender, J. & McWherter J. 2011. Professional Test Driven Development with C#. John Wiley & Sons, Incorporated.

Blundell P. & Torres Milano D. 2015. Learning Android Application Testing. Packt Publishing.

Cruz, B. & Niñirola, A. 2014. Testing and Securing Android Studio Applications. Packt Publishing.

Github 2018a. JUnit 4 wiki. Viitattu 8.4.2018
https://github.com/junit-team/junit4/wiki.

Github 2018b. Mockito wiki – How to write good tests. Viitattu 9.4.2018
https://github.com/mockito/mockito/wiki/How-to-write-good-tests.

Grzejszczak, M. 2014. Mockito Cookbook. Packt Publishing.

Homès, B. 2011. Fundamentals of Software Testing. John Wiley & Sons, Incorporated.

IDC 2018. Smartphone OS. Viitattu 26.4.2018
https://www.idc.com/promo/smartphone-market-share/os.

Javadoc 2018. Annotation Type InjectMocks. Viitattu 3.5.2018
http://static.javadoc.io/org.mockito/mockito-core/2.18.3/org/mockito/InjectMocks.html.

JUnit 2018a. Javadoc – Rule. Viitattu 9.4.2018
https://junit.org/junit4/javadoc/4.12/org/junit/Rule.html.

JUnit 2018b. JUnit5. Viitattu 8.4.2018
https://junit.org/junit5/.

Loveland, S., Miller, G., Prewitt, R. & Shannon, M. 2004. Software Testing Techniques : Finding the Defects that Matter. Charles River Media.

Mockito 2018. Viitattu 9.4.2018
http://site.mockito.org/.

Robolectric 2018. Viitattu 8.5.2018
http://robolectric.org/.

Vocke, H. 2018. The Practical Test Pyramid. Viitattu 19.4. 2018
https://martinfowler.com/articles/practical-test-pyramid.html.

LIITTEET

Liite 1

```
public class CustomValidatorUnitTest {

    //Luokka, jonka toimivuutta halutaan testata
    CustomValidator customValidator;

    //Kutsutaan ennen jokaista @Test-annotaatiolla merkittyä metodia
    @Before
    public void setUp() {
        customValidator = new CustomValidator();
    }

    //Validin sähköpostiosoitteen testaaminen
    @Test
    public void testEmailValidation() {
        boolean valid = customValidator.IsValidEmail("valid.email@fake.com");
        assertTrue(valid);
    }

    //Epäkelvon sähköpostiosoitteen testaaminen
    @Test
    public void testInvalidEmailValidation() {
        boolean valid = customValidator.IsValidEmail("not.a.valid.email@.com");
        assertFalse(valid);
    }

    //Validin käyttäjänimen testaaminen
    @Test
    public void testValidUsernameValidation() {
        boolean valid = customValidator.IsValidUsername("Username");
        //Voidaan assertTrue()-metodin sijasta käyttää assertThat
        assertThat(valid, is( value: true));
    }

    //Käyttäjänimessä ei saa olla erikoismerkkejä
    @Test
    public void testUsernameValidationWithSpecialCharacters() {
        boolean valid = customValidator.IsValidUsername("Username!");
        assertThat(valid, is( value: false));
    }

}
```

Esimerkki JUnit4-testiluokasta, jossa testataan sähköpostiosoitteiden ja käyttäjänimien validointiin käytettyä ominaisuutta.

Liite 2

```
@RunWith(MockitoJUnitRunner.class)
public class PeopleFinderUnitTest {

    //Hakutermi
    private static final String QUERY_NAME = "Mike";

    //Merkkijono muodossa, jossa se olisi kun mockedExternalStoragen
    //queryPeople()-metodia kutsuttaisiin oikeasti
    private static final String MOCKED_QUERY_RESULT =
            "[{\"name\": \"Mike Johnson\"}, {\"name\": \"Michael Matthews\"}]";

    //Mock-objektin luominen
    //Luokkaa ei tarvitse enää itse alustaa
    @Mock
    ExternalStorage mockedExternalStorage;

    //Mockito alustaa testattavan muuttujan
    //ja injektoi mockedExternalStorage-muuttujan
    @InjectMocks
    PeopleFinder peopleFinder;

    @Test
    public void testFind() {

        //@InjectMocks-annotaation sijaan voitaisiin käyttää tätä
        //peopleFinder = new PeopleFinder(mockedExternalStorage);

        //Mockataan mockedExternalStoragen queryPersons()
        when(mockedExternalStorage.queryPeople(QUERY_NAME))
                .thenReturn(MOCKED_QUERY_RESULT);

        //Haetaan henkilöiden nimet
        ArrayList<String> people = peopleFinder.findByName(QUERY_NAME);

        //Varmistetaan vielä, että metodia kutsuttiin vain kerran
        verify(mockedExternalStorage).queryPeople(QUERY_NAME);

        //Tarkistetaan, että lista ei ole NULL
        assertNotNull(people);

        //Tarkistetaan, että lista sisältää tasan kaksi (2) merkkijonoa
        assertThat(people.size(), is( value: 2));

        //Tarkistetaan, että lista sisältää odotetut nimet
        assertTrue(people.contains("Mike Johnson"));
        assertTrue(people.contains("Michael Matthews"));

    }
}
```

Esimerkki JUnit4-testiluokasta, jossa on käytetty Mockito-työkalua. Esimerkissä testataan henkilöiden etsimiseen käytettyä luokkaa, jonka riippuvuus on korvattu mock-objektilla.

Liite 3 1(2)

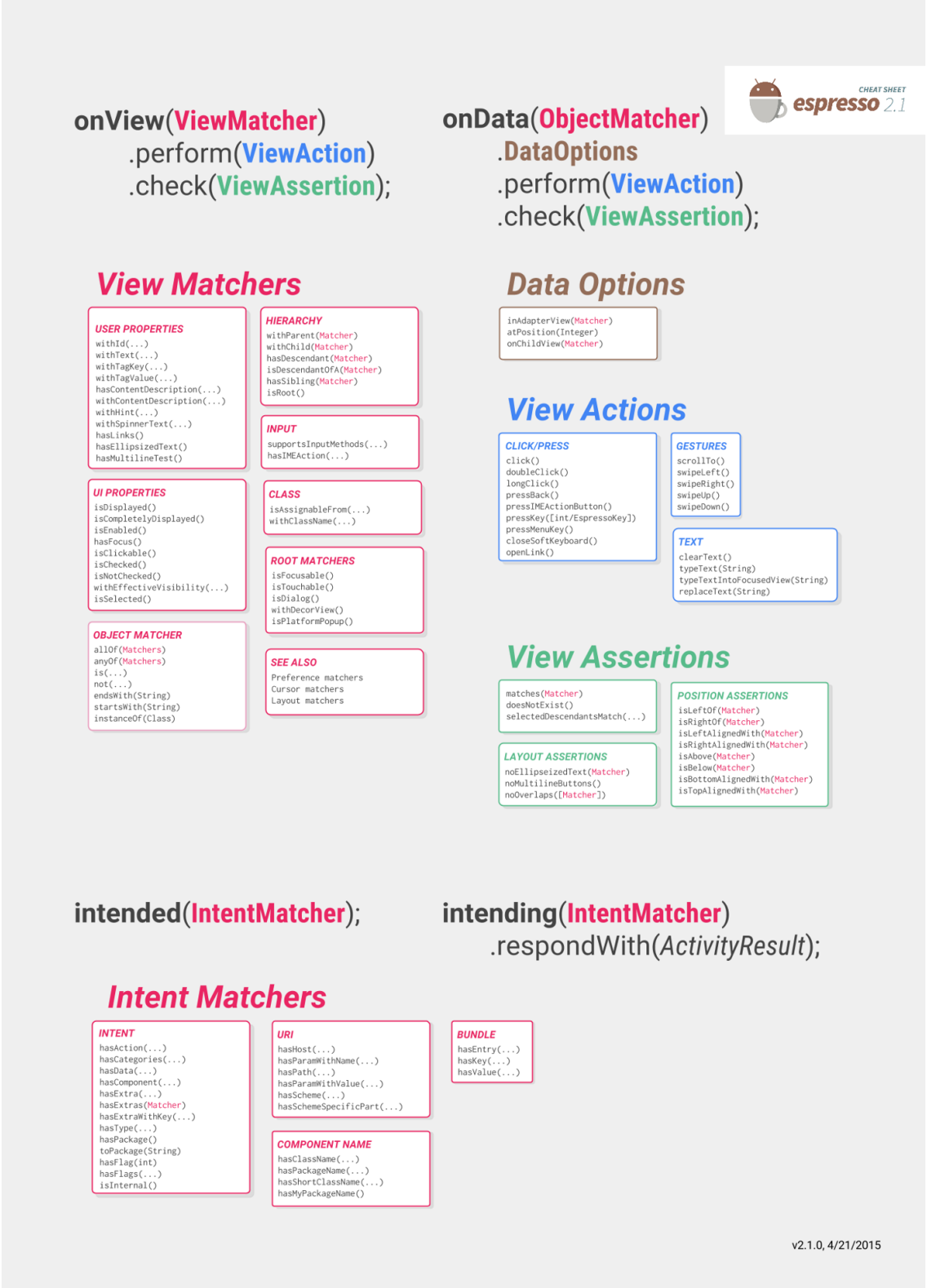
```
@RunWith(AndroidJUnit4.class)
public class UnitConversionUITest {

    private String fromUnit = "Inch";
    private String toUnit = "Centimeter";

    @Rule
    public ActivityTestRule<MainActivity> activityRule =
            new ActivityTestRule<>(MainActivity.class);

    @Test
    public void testConversion() {

        String validValue = "15.25";
        String expectedValue = "38.7";

        //Avataan pudotusvalikko
        onView(withId(R.id.spinner_from_unit))
                .perform(click());

        //Haetaan pudotusvalikosta oikea kohta ja klikataan sitä
        onData(allOf(is(instanceOf(String.class)), is(fromUnit)))
                .perform(click());

        //Avataan toinen pudotusvalikko
        onView(withId(R.id.spinner_to_unit))
                .perform(click());

        //Haetaan pudotusvalikosta oikea kohta ja klikataan sitä
        onData(allOf(is(instanceOf(String.class)), is(toUnit))).perform(click());

        //Haetaan teksikenttä, johon syötetään arvo
        onView(withId(R.id.et_from_unit))
                .perform(typeText(validValue), closeSoftKeyboard());

        //Painetaan muunto-painiketta
        onView(withId(R.id.btn_convert)).perform(click());

        //Tarkastetaan, että tekstikentässä on haluttu arvo
        onView(withId(R.id.et_to_unit)).check(matches(withText(startsWith(expectedValue))));

    }
```

Esimerkki Espresso-testiluokasta. Testissä testataan yksiköiden muuntamiseen käytettyä sovellusta.

```
@Test
public void testConversionWithNoInput() {

    //Varmistetaan, että yksikkö valittuna
    onView(withId(R.id.spinner_from_unit))
            .perform(click());
    onData(allOf(is(instanceOf(String.class)), is(fromUnit)))
            .perform(click());

    //Varmistetaan, että yksikkö valittuna
    onView(withId(R.id.spinner_to_unit))
            .perform(click());
    onData(allOf(is(instanceOf(String.class)), is(toUnit)))
            .perform(click());

    //Haetaan tekstikenttä ja poistetaan sen teksti
    onView(withId(R.id.et_from_unit))
            .perform(clearText(), closeSoftKeyboard());

    //Painetaan muunto-painiketta
    onView(withId(R.id.btn_convert)).perform(click());

    //Tarkastetaan, että virheviesti on näkyvissä ja sisältää odotetun tekstin
    onView(withId(R.id.tv_conversion_error))
            .check(matches(allOf(withText(R.string.conversion_error_invalid_input), isDisplayed())));
}
```

Espresso-testiluokan toinen testi, jossa testataan sovelluksen käyttöliittymän muokkautuvuutta virhetilanteissa.

Liite 4



Espresso-työkalun muistilappu, jossa on kuvattuna testien kirjoittamiseen yleisimmin käytetyt toiminnot. (Android Developers 2018f)