hyväksymispäivä arvosana

arvostelija

Android-sovellusten testaaminen

Juho Niemistö

Helsinki 16.6.2013

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author

Juho Niemistö

Työn nimi — Arbetets titel — Title

Android-sovellusten testaaminen

Oppiaine — Läroämne — Subject

Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	16.6.2013	55 sivua

Tiivistelmä — Referat — Abstract

Ääkkönen

Avainsanat — Nyckelord — Keywords

Android

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Googlen kehittämä Android on noussut viime vuosina markkinaosuudeltaan suurimmaksi mobiililaitteiden käyttöjärjestelmäksi. Kuka tahansa voi kehittää Androidille sovelluksia, joiden kehittämiseen tarvittavat välineet ovat ilmaiseksi saatavilla. Erilaisia sovelluksia onkin kehitetty jo yli 500 000.

Mobiilisovellusten kehittämiseen liittyy monia haasteita. Niitä ovat muunmuassa rajalliset laitteistoresurssit, käytettävyys pienellä näytöllä sekä yksityisyyteen ja tietoturvaan liittyvät kysymykset. Androidilla on lisäksi esimerkiksi Applen iOS-alustaan verrattuna omana haasteenaan Android-laitteiden valtava kirjo. Android-laitteita valmistavat kymmenet eri valmistajat ja ne vaihtelevat kameroista tablettien kautta digibokseihin. Laitteissa on hyvin eritehoisia prosessoreita ja lisälaitteita, kuten gps- tai kiihtyvyysantureita, on vaihtelevasti.

Sovellusten laatu on erityisen tärkeää Android-alustalla, jossa kilpailua on runsaasti ja sovellusten hinta niin alhainen, ettei se muodosta estettä sovelluksen vaihtamista toiseen. Sovelluskauppa on myös aina saatavilla suoraan laitteesta. Tämä asettaa sovellusten testaamisellekin haasteita. Toisaalta sovellukset tulisi saada nopeasti sovelluskauppaan, mutta myös sovellusten laadun pitäisi olla hyvä. Testityökalujen pitäisi siis olla helppokäyttöisiä, ja tehokkaita. Androidille onkin kehitetty lukuisia testaustyökaluja Googlen omien työkalujen lisäksi.

Tutkimuksen tavoitteena on perehtyä Android-sovellusten rakenteeseen, niiden testaamiseen ja Android-alustalla toimiviin automaattisen testauksen työkaluihin. Tutustun kirjallisuudessa esiteltyihin sekä Androidin kehitystyökalujen mukana tuleviin testaustyökaluihin. Kirjallisuuskatsauksen lisäksi vertailen työkaluja testaamalla niiden avulla esimerkkisovellusta ja analysoin testien perusteella niiden puutteita ja vahvuuksia.

Käsittelen tässä tutkimuksessa vain Androidille Javalla kehitettyjä natiivisovelluksia. Androidille voi kehittää sovelluksia myös muunmuassa html5:llä ja erilaisilla työkaluilla, jotka generoivat automaattisesti natiivikoodia useille mobiilialustoille. Osa testaustyökaluista mahdollistaa toki myös tällaisten sovellusten testaamisen. Keskityn vain automaattisiin testaustyökaluihin, joten esimerkiksi manuaaliseen käytettävyystestaukseen liittyvät prosessit ja työkalut on rajattu tämän työn ulkopuolelle.

Keskityn erityisesti yksikkö- ja toiminnallisiin testityökaluihin. Yksikkötestityökaluja käyttävät useimmiten sovelluksen ohjelmoijat itse sovelluksen koodausvaiheessa. Yleistä on myös yksikkötestien kirjoittaminen jo ennen vastaavan ohjelmakoodin

kirjoittamista. Toiminnallisissa testeissä testataan sovellusta hieman korkeammalta tasolta, jolloin pyritään varmistamaan haluttujen toiminnallisuuksien toiminta kokonaisuudessaan sovelluksessa. Nämä työkalut olen valinnut huomion kohteeksi, koska toiminnallinen ja yksikkötestaus ovat yleisimmät testauksen muodot mobiilisovellusta kehitettäessä ja työkaluvalikoima on myös monipuolisin.

Toisessa luvussa esittelen Androidin kehitystyökaluja sekä arkkitehtuuria sovellusten kehittäjän näkökulmasta. Luvussa tutustutaan Androidin tärkeimpiin komponentteihin: aktiviteetteihin, palveluihin ja sisällöntarjoajiin sekä komponenttien väliseen kommunikointiin aikeiden avulla. Lisäksi esittelen Android manifestin sekä Androidin matalan tason tietoturvaratkaisuja.

Kolmannessa luvussa käsitellään ohjelmistojen testaamisen perusasioita, testausta osana ohjelmistotuotantoprosessia sekä mobiilisovellusten testaamisen erityispiirteitä. Lisäksi pohditaan testaustyökalujen arviointia ja laatukriteerejä. Androidin kehitystyökalupakettin mukana tulee runsaasti testaustyökaluja. Esittelen niitä luvussa neljä.

Viidennessä luvussa syvennytään yksikkötestityökaluihin. Vertailen Androidin yksikkötestikehystä avoimen lähdekoodin Robolectriciin, joka pyrkii tarjoamaan Androidin oman yksikkötestikehyksen ominaisuudet tavanomaisessa Java-ympäristössä, jotta testien ajaminen olisi nopeampaa. Selvitän luvussa myös yksikkötestikehysten yhteensopivuutta mock-kehysten kanssa.

Kuudennessa luvussa käsitellään toiminnallisen testauksen työkaluja. Vertailen Androidin Uiautomator-työkalun lisäksi Robotium ja Troyd -testityökaluja. Kukin näistä työkaluista tarjoaa hieman erilaisen lähestymistavan ja abstraktiotason toiminnallisten testien kirjoittamiseen.

2 Android

Tässä luvussa esitellään Androidin historiaa, kehitystyökaluja sekä Android-sovellusten arkkitehtuuria ja pääkomponentit.

2.1 Historia

Androidin kehityksen aloitti Android Inc. -niminen yritys vuonna 2003. Google osti sen vuonna 2005. Kaksi vuotta myöhemmin, marraskuussa 2007 Androidin ensimmäinen versio julkaistiin ja samalla kerrottiin, että sen kehityksestä vastaa Open Handset Alliance, johon kuului Googlen lisäksi puhelinvalmistajia, kuten HTC ja Samsung, operaattoreita, kuten Sprint Nextel ja T-Mobile sekä komponenttivalmistajia, kuten Qualcomm ja Texas Instruments.

Ensimmäinen Androidille julkaistu kaupallinen laite oli HTC Dream -älypuhelin, joka julkaistiin lokakuussa 2008. Loppuvuodesta 2010 Android nousi älypuhelinten markkinajohtajaksi. Syksyllä 2012 Androidilla oli jo tutkimuksesta riippuen 50-70 prosentin markkinaosuus ja laitevalikoima on kasvanut älypuhelimista muunmuassa tablet-tietokoneisiin, digibokseihin ja kameroihin [wik].

2.2 Androidin kehitystyökalut

Android-sovelluksia tehdään Java-ohjelmointikielellä. Google julkaisee Androidille ilmaista ohjelmistokehitystyökalua (Android SDK), joka kääntää sovelluksen ja pakkaa sen kuvien ja muiden resurssien kanssa apk-tiedostoksi (Android Application Package). Apk-tiedosto sisältää kaiken yhden sovelluksen asentamiseen tarvittavat tiedot. Android-sovellusten kehittämiseen tarvitsee käytännössä Javan kehitystyökaluista (SDK) version 5 tai 6, Androidin SDK:n, Eclipsen sekä Android-laajennoksen (Android Development Tools, ADT) Eclipselle.

Androidin SDK:n mukana tulee minimoitu versio Androidin järjestelmäkirjastoista, joiden avulla Eclipse osaa opastaa Androidin rajapintojen käytössä. Rajapinnan takana ei ole kuitenkaan oikeaa toteutusta, joten esimerkiksi yksikkötestit, jotka menevät kirjastoluokkiin asti, eivät toimi Eclipsestä Javalla ajettaessa.

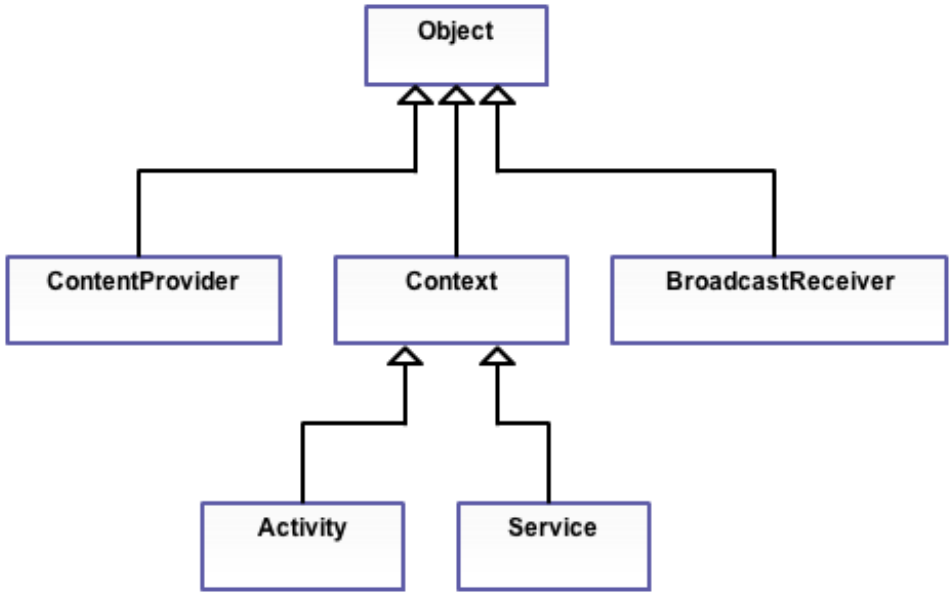
Sovelluksen ja testien ajamista varten SDK:n mukana tulee Android-emulaattori. Emulaattoreita voi ajaa eri Android API:n versioilla ja laitteistoprofiileilla, jotta on mahdollista testata sovelluksen toimivuutta erilaisissa Android-ympäristöissä. Emu-

laattori kykenee myös jossain määrin simuloimaan lisälaitteiden, kuten kiihtyvyysanturin, toimintaa. Suurin puute emulaattorissa on sen heikko suoritusnopeus. Sovelluksia ja testejä voi ajaa myös suoraan tietokoneeseen liitetyssä Android-laitteessa.

Androidin Eclipse-laajennos toimii siltana Android SDK:n ja Eclipsen välillä mahdollistaen SDK:n tarjoamien ominaisuuksien hyödyntämisen suoraan Eclipsestä käsin. ADT:n avulla on myös mahdollista seurata tietokoneeseen kytkettyjen Android-laitteiden tapahtumalogeja ja debug-tietoja [ZG11, 25-50].

2.3 Android-sovellusten rakenne

Android on rakennettu Linuxin ytimen version 2.6 päälle ja koko Androidin järjestelmäkoodi on avointa, mikä tarkoittaa, että mikä tahansa valmistaja voi tehdä Androidin pohjalta oman mobiilikäyttöjärjestelmänsä. Jokainen sovellus on käyttäjänä järjestelmässä. Sovellusten oikeudet on rajattu siten, että ne pääsevät käsiksi vain kyseiseen sovellukseen liittyviin resursseihin. Sovelluksen ollessa käynnissä, se pyörii omana prosessina Linux-prosessien tavoin, jota Android-käyttöjärjestelmä hallitsee. Androidin turvallisuusratkaisu noudattaa vähimmän mahdollisen tiedon periaatetta; sovelluksella on vain ne oikeudet, joita se vähintään tarvitsee toimintaansa. Kaikkia ylimääräisiä oikeuksia varten täytyy erikseen pyytää lupa.



Kuva 1: Androidin tärkeimpien komponenttien luokkahierarkia

Android-sovellukset koostuvat neljästä komponenttityypistä: aktiviteeteista (acti-

vities), palveluista (services), sisällöntarjoajista (content providers) sekä lähetysten vastaanottajista (broadcast receivers). Komponenttien välinen kommunikointi on pääosin tapahtumapohjaista; eri komponentit eivät keskustele suoraan keskenään, vaan kaikki siirtymät komponenttien välillä tapahtuvat käyttöjärjestelmän välittämien tapahtumaviestien perusteella. Tämän vaikutuksesta Android-sovellukset voivat helposti käyttää toiminnassaan järjestelmän ja toisten sovellusten tarjoamia komponentteja.

Kuvassa 1 on esitelty Androidin peruskomponenttien muodostama luokkahierarkia. Vain aktiviteeteilla ja palveluilla on yhteinen yliluokka Context, lähetysten vastaanottajat ja sisällöntarjoajajat perivät vain Javan geneerisen Object-luokan. Kuvaa on yksinkertaistettu siten, että Contextin ja Activityn ja Servicen välillä olevia Wrapper-luokkia on jätetty kuvaamatta perintähierarkiassa. Context-luokka tarjoaa aktiviteettien ja palveluiden käyttöön sovelluksen globaaliin tilaan liittyviä tietoja.

Aktiviteetti kuvaa yhtä sovelluksen käyttöliittymän kerrallaan muodostavaa näkymää. Sovelluksen käyttöliittymä koostuu useista aktiviteeteista, jotka muodostavat yhtenäisen sovelluksen, mutta jokainen aktiviteetti on toisistaan riippumaton. Eri sovellukset voivat myös käynnistää toistensa aktiviteetteja, mikäli vastaanottava sovellus sen sallii. Esimerkiksi kamera-sovellus voi käynnistää sähköposti-sovelluksen sähköpostinkirjoitus-aktiviteetin, jos ottamansa kuvan haluaa jakaa sähköpostilla. Aktiviteetit ovat Androidissa Activity-luokan aliluokkia.

Palvelut ovat taustaprosesseja, jotka suorittavat pitkäkestoisia operaatioita, kuten tiedon lataamista verkosta tai musiikin soittamista taustalla samalla, kun käyttäjä käyttää toista sovellusta. Palvelut eivät tarjoa käyttöliittymää ja toiset komponentit, kuten aktiviteetit, voivat käynnistää niitä. Palvelut ovat Service-luokan aliluokkia.

Sisällöntarjoajat vastaavat sovelluksen tarvitseman tiedon lukemisesta ja kirjoittamisesta pitkäkestoiseen muistiin. Tallennuspaikkana voi olla laitteen tiedostojärjestelmä, SQLite-tietokanta, verkko tai ylipäänsä mikä tahansa kohde, johon sovelluksella on luku- tai kirjoitusoikeudet. Sovellukset voivat käyttää toistensa sisällöntarjoajia, mikäli sovellus julkaisee ne muiden sovellusten käyttöön. Sisällöntarjoajat ovat ContentProvider-luokan aliluokkia.

Lähetysten vastaanottajat reagoivat järjestelmänlaajuisiin viesteihin ja tapahtumiin. Tällaisia ovat esimerkiksi ilmoitus, että akku on lopussa tai että käyttäjä on sulkenut tai avannut näytön. Ne voivat myös lähettää järjestelmänlaajuisia tapahtumaviestejä muille sovelluksille. Lähetysten vastaanottajat ovat BroadcastReceiver-luokan aliluokkia, ja tapahtumat ovat Intent-luokan aliluokkia.

Android-sovellukset käyttävät usein hyväkseen toisten sovellusten komponentteja. Sovellukset eivät pysty suoraan kutsumaan toisiaan, vaan halutessaan hyödyntää toisten sovellusten ominaisuuksia sovellus luo uuden aikeen, jonka järjestelmä välittää tiettyjen sääntöjen perusteella sopivalle vastaanottajalle (katso luku 2.7).

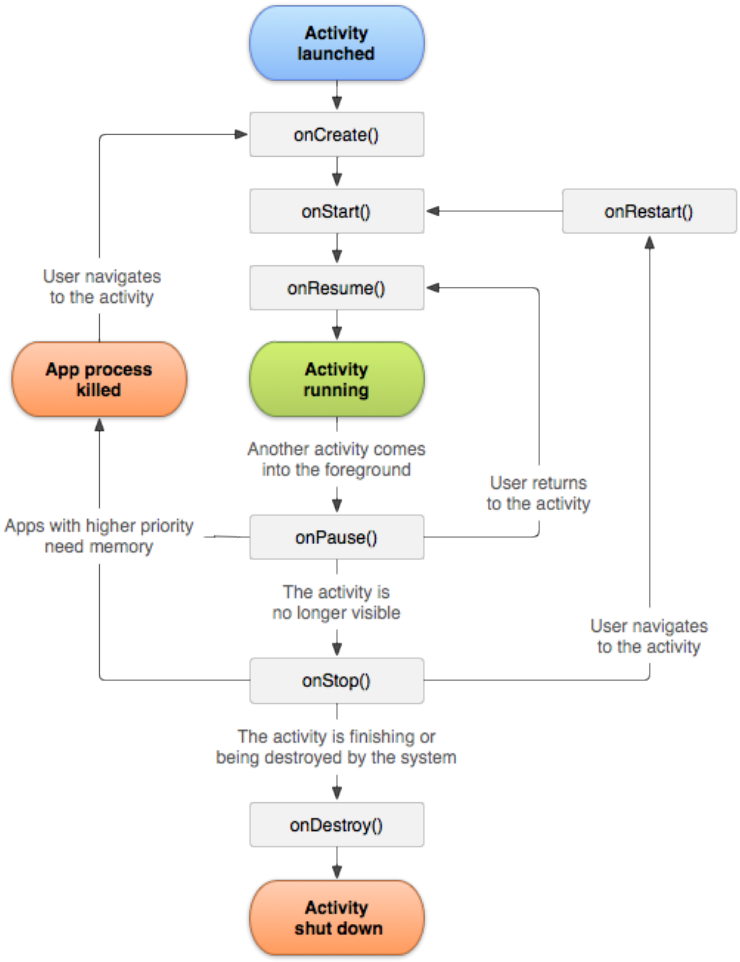
Android-sovelluksilla ei ole yksittäistä main-metodia, joka käynnistäisi ohjelman, kuten usein muissa sovelluksissa on tapana. Sovellus voi sen sijaan käynnistyä vastaanottamansa aikeen johdosta monen eri komponentin kautta. Lisäksi sovellus saatetaan joutua käynnistämään ja sulkemaan useita kertoja esimerkiksi käyttäjän vaihtaessa puhelimen orientaatiota tai vastaanotettaessa puhelua, joten sovelluksen pitää pystyä tehokkaasti palautumaan keskeytyneeseen tilaansa. Sovelluksella on siis lukuisia mahdollisia käynnistymis- ja sulkeutumispolkuja. Tämän takia ohjelmakomponenttien elinkaaren hallinta on tärkeä osa sovelluksen rakentamista.

Android-sovelluksilla on xml-muotoinen Manifest-tiedosto, jossa määritellään sovelluksen komponentit, niiden näkyvyys ja minkälaisia tapahtumia ne osaavat hallita. Manifestissa määritellään myös mitä rajoitteita sovellus asettaa käytettävissä olevalle Android-versiolle, puhelimen ominaisuuksille, kuten lisälaitteiden saatavuudelle ja näyttöresoluutiolle, ja mitä oikeuksia sovellus vaatii toimiakseen. Näin voidaan varmistaa, että sovellusta ei asenneta laitteelle, jossa ei ole sovelluksen välttämättä tarvitsemia ominaisuuksia [andb].

2.4 Aktiviteetit

Aktiviteetti kuvaa yhtä sovelluksen käyttöliittymän näkymää. Lähes aina aktiviteetti on koko näytön kokoinen - eli kaikki, mitä puhelimen ruudulla näkyy yläreunan status-palkkia lukuunottamatta on samaa aktiviteettia - mutta ne voivat olla myös pienempiä tai leijua osittain toisen aktiviteetin päällä. Kuitenkin vain yksi aktiviteetti voi olla aktiivinen, eli reagoida käyttäjän syötteisiin, kerrallaan. Yksi sovelluksen aktiviteeteistä on yleensä pääaktiviteetti, joka käynnistyy silloin, kun käyttäjä avaa sovelluksen.

Aktiviteettien elinkaaren hallinta on Android-sovelluksen kriittisimpiä osia, koska järjestelmän resurssit ovat yleensä hyvin rajalliset ja Android-laitteiden käyttöön liittyy usein tiheä vaihtelu eri sovellusten välillä. Tällöin on tärkeää, että sovellus luovuttaa varaamansa resurssit muiden sovellusten käyttöön, kun sovellus vaihtuu, ja vastaavasti osaa palautua takaisin pysäytettäessä olleeseen tilaan käyttäjän palatessa sovellukseen. Nämä vaihdokset pitäisi lisäksi tapahtua mahdollisimman te-



Kuva 2: Aktiviteetin elinkaari

hokkaasti, jotta järjestelmän toiminta olisi käyttäjän näkökulmasta mahdollisimman sulavaa sovellusten tilojen vaihtamisen yhteydessä.

Aktiviteetilla voi olla pitkäkestoisemmin kolme eri tilaa. Aktiviteetti on aktiivisessa tilassa (resumed) silloin, kun se on näytön etualalla ja käyttäjä käyttää juuri sitä aktiviteettia. Keskeytetyssä (paused) tilassa aktiviteetti on, kun se on osittain näkyvissä, mutta jokin toinen aktiviteetti on aktiivisena sen päällä. Keskeytetyt aktiviteetit ja niiden tilat pysyvät muistissa, joskin jos laitteen muisti on lopussa, järjestelmä saattaa tuhota sen. Aktiviteetti on pysäytetty (stopped) silloin, kun jokin toinen aktiviteetti peittää sen kokonaan näkyvistä. Tällainenkin aktiviteetti säilyy muistissa, jos laitteen resurssit ovat riittävät, mutta järjestelmä voi tuhota sen koska vain, jos resursseja tarvitaan muiden aktiviteettien käyttöön.

Aktiviteetin siirtyminen eri tilojen välillä tapahtuu järjestelmän kutsuessa aktiviteetin takaisinkutsumetodeita. Mahdolliset tilasiirtymäpolut näkyvät kuvassa 2.

Aktiviteetin koko elinkaari tapahtuu onCreate()- ja onDestroy()-kutsujen välillä. Aktiviteetin tulisi tehdä kaikki kerran suoritettavat tilanalustustehtävät kutsuttaessa onCreate()-metodia, kuten ulkoasun määrittely tai koko aktiviteetin elinkaaren ajan tarvittavan tiedonsiirtosäikeen avaus. Vastaavasti onDestroy()-kutsussa aktiviteetin tulisi vapauttaa kaikki loputkin aktiviteetin varaamat resurssit.

Aktiviteetin käyttäjälle näkyvä elinkaari on onStart()- ja onStop()-kutsujen välillä. onStart()-metodia kutsutaan, kun aktiviteetti tulee näkyväksi käyttäjälle, ja onStop()-metodia kutsutaan, kun jokin toinen aktiviteetti on peittänyt kyseisen aktiviteetin kokonaan. Näkyvän elinkaaren aikana tulisi ylläpitää niitä resursseja, joita tarvitaan käyttäjän kanssa kommunikointiin sekä sellaisia, jotka saattavat muuten vaikuttaa käyttäjälle näkyvään käytöliittymään. Esimerkiksi lähetystenvastaanottajaa on hyvä kuunnella tällä välillä mahdollisten järjestelmänlaajuisten käyttöliittymään vaikuttavien tapahtumien varalta. onStart() ja onStop() -kutsuja voi tulla lukuisia aktiviteetin koko elinkaaren aikana. onRestart()-metodia kutsutaan, jos aktiviteetti on jo luotu aiemmin ja pysäytetty sitten onStop()-kutsulla. onRestart()-kutsua seuraa aina onStart()-kutsu.

Aktiviteetti on aktiivisena näytön etualalla onResume() ja onPause() -kutsujen välillä. Kun aktiviteetti on etualalla, käyttäjä käyttää juuri sitä ja se on kaikkien muiden aktiviteettien päällä. onResume() ja onPause() -kutsuja voi tulla tiheästi, esimerkiksi aina kun laitteen näyttö menee lepotilaan tai tulee jokin ilmoitus aktiviteetin päälle, joten niiden toteutus ei saa olla liian raskas.

Androidin järjestelmä voi tuhota sovelluksen prosessin onPausen(), onStopin() tai onDestroyn() jälkeen. Tämän takia pysyväksi tarkoitettu tieto on tallennettava onPause()-kutsun jälkeen. Tallennus voidaan tehdä esimerkiksi toteuttamalla takaisinkutsumetodi onSaveInstanceData(), jota kutsutaan aina, ennen kuin järjestelmä mahdollistaa aktiviteetin tuhoamisen. onSaveInstanceData() saa parametrinaan Bundle-olion, johon voi tallentaa tietoja nimi-arvo-pareina. Sama Bundle-olio tulee aktiviteetille onCreate() ja onRestoreInstanceState() -metodeille. Tiedon palautuksen voi tehdä kummassa tahansa näistä metodeista. Activity-luokka tarjoaa myös oletustoteutuksen onSaveInstanceData() ja onRestoreInstanceState()-metodeista, jotka osaavat monissa tapauksissa suorittaa tiedon tallennuksen ja palautuksen. Aktiviteetin tilanpalautusta tarvitaan usein, esimerkiksi aina kun käyttäjä vaihtaa sovelluksen suuntaa pysty- ja vaakasuuntien välillä.

Aktiviteettien vaihtumisen yhteydessä takaisinkutsujen järjestys on aina sama. Kun aktiviteetti A käynnistää aktiviteetti B:n, ensin kutsutaan aktiviteetti A:n onPause()-

metodia, sitten aktiviteetti B:n onCreate(), onStart() ja onResume()-metodeita peräkkäin. Viimeiseksi kutsutaan aktiviteetti A:n onStop()-metodia, mikäli aktiviteetti B peittää sen kokonaan. Näin esimerkiksi aktiviteetti A:n onPause()-metodissa tietokantaan tallennetut tiedot ovat käytössä aktiviteetti B:tä käynnistettäessä. Jos muutoksia taas tekee onStop()-metodissa, ne tapahtuvat vasta aktiviteetti B:n käynnistyttyä.

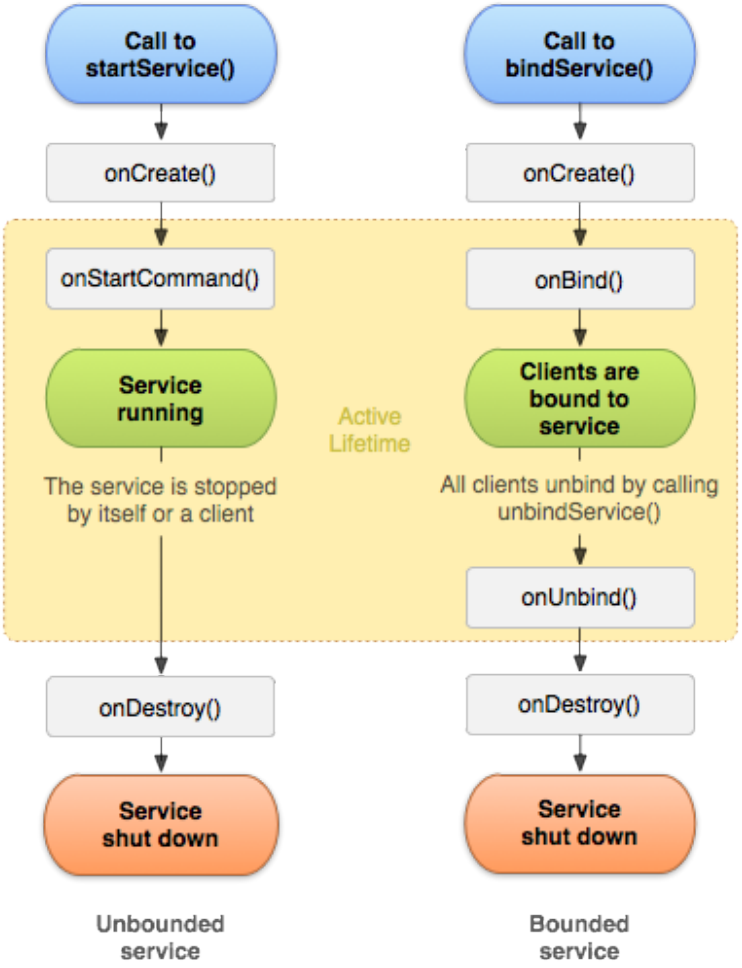
Palat

Androidin versiosta 3 (API-versio 11) asti on ollut mahdollista määritellä aktiviteetteihin paloja (Fragment-luokan aliluokkia). Palat ovat uudelleenkäytettäviä komponentteja, joita voi käyttää osana aktiviteetteja. Niiden avulla on helpompi luoda käyttöliittymiä, jotka skaalautuvat eri kokoisille näytöille. Isommalla näytöllä aktiviteetti voi pitää sisällään useita paloja, jotka pienemmällä näytöllä ovatkin omissa aktiviteeteissaan. Palojen elinkaari on riippuvainen siitä aktiviteetista, johon ne on sisällytetty. Kun aktiviteetti pysähtyy, niin pysähtyy myös aktiviteetin sisältämät palat. Samoin aktiviteetin tuhoutuessa tuhoutuvat myös palat. Aktiviteettien sisällä paloilla voi kuitenkin olla oma elinkaarensa, niitä voi käynnistää ja tuhota vapaasti aktiviteetin ollessa käynnissä.

Palojen elinkaarta hallitaan takaisinkutsumetodeilla, kuten aktiviteettejakin. Monet metoditkin ovat samoja, kuten onCreate(), onStart(), onPause() ja onStop(). Lisäksi paloilla on muutamia takaisinkutsumetoideita, joita aktiviteeteilla ei ole. onCreateView()-metodia kutsutaan, kun palan käyttöliittymä tulee ensimmäistä kertaa näkyviin käyttäjälle. onAttach()-metodia kutsutaan, kun pala on liitetty johonkin aktiviteettiin. Tällöin pala saa itselleen viitteen aktiviteettiin kommunikointia varten. onActivityCreated()-metodia kutsutaan, kun palan liittäneen aktiviteetin onCreated()-metodi on ajettu. onDestroyView()-metodia kutsutaan, kun palan käyttöliittymä tuhotaan, ja onDetach()-metodia kutsutaan kun palaan liitetty aktiviteetti irroitetaan palasta [andb].

2.5 Palvelut

Palvelut ovat pitkäkestoisia taustaoperaatioita. Muut sovelluskomponentit voivat käynnistää niitä, ja ne jatkuvat vaikka käyttäjä lopettaisi kyseisen sovelluksen käyttämisen. Palvelu voi esimerkiksi soittaa musiikkia, suorittaa verkkotransaktioita, kommunikoida sisällöntarjoajien kanssa tai tehdä levykirjoitusta.



Kuva 3: Palvelun elinkaari

Palvelut voivat olla kahdenlaisia. Käynnistettävät (*started*) palvelut suorittavat tehtävänsä kun niiden startService()-metodia kutsutaan. Tällainen palvelu voi jatkaa pyörimistä taustalla, vaikka sovellus suljettaisiin. Tyypillisesti käynnistettävä palvelu tekee jonkin yhden operaation, kuten tiedoston latauksen tai lähettämisen, ja lopettaa sitten itsensä. Käynnistettävät palvelut eivät yleensä palauta palautusarvona kutsujalle mitään. Käynnistettävien palveluiden tulee sulkea itsensä operaation valmistuttua kutsumalla stopSelf()-metodia. Myös muut komponentit voivat sulkea palvelun kutsumalla stopService()-metodia.

Sidotut (*bound*) palvelut ovat sellaisia, että sovelluskomponentit sitovat palvelun niihin kutsumalla bindService()-metoida. Sidotut palvelut tarjoavat asiakas-palvelinrajapinnan sitovalle komponentille. Palvelu voi vastaanottaa pyyntöjä ja palauttaa vastauksia niihin. Palvelun elinkaari on sama kuin sen sitoneen komponentin. Useampi komponentti voi sitoa saman palvelun yhtä aikaa. Tällöin palvelu sulkeu-

tuu kun viimeinenkin niistä lopettaa toimintansa. Sitominen vapautetaan kutsumalla unbindService()-metodia.

Useimmiten käynnistettävät ja sidotut palvelut ovat erillisiä, mutta joissain tilanteissa sama palvelu voi toimia sekä käynnistettävänä että sidottuna palveluna. Käynnistettäviä palveluita käytetään tyypillisesti pitkäkestoisiin taustaoperaatioihin, jotka suoritetaan taustalla ilman että käyttäjä puuttuu niiden toimintaan. Sidotut palvelut taas voivat tarjota sovellukselle minkä tahansa palvelurajapinnan, jonka kanssa sovellus voi kommunikoida palvelun elinkaaren ajan.

Palveluiden elinkaari on esitetty kuvassa 3. Aktiviteettien tavoin koko palvelun elinkaari tapahtuu onCreate() ja onDestroy()-kutsujen välissä ja palvelun alustus tapahtuu onCreate()-metodissa. Sidotun palvelun aktiivinen elinkaari on onBind() ja onUnbind()-kutsujen välillä. Käynnistettävän palvelun elinkaari puolestaan alkaa onStartCommand()-kutsusta kunnes se sulkee itsensä stopSelf()-kutsulla. onBind() ja onStartCommand() -metodit saavat parametrinaan aikeen, jonka niitä kutsunut komponentti antoi bindService() tai startService() -metodille [andb].

2.6 Sisällöntarjoajat

Sisällöntarjoajat tarjoavat pääsyn pysyvästi tallennettuun tietoon. Ne kapsuloivat tiedon ja tarjoavat mekanismit tiedon yksityisyyden hallintaan. Sisällöntarjoajat toimivat rajapintana tiedon ja sovelluskoodin välillä. Kun sisällöntarjoajan tietoon halutaan päästä käsiksi, käytetään ContentResolver-oliota Context-luokassa, joka sitten kommunikoi itse sisällöntarjoajan kanssa.

Sisällöntarjoajat eivät ole välttämättömiä sovelluksessa, jos tietoon ei haluta päästä käsiksi muista kuin samasta sovelluksesta. Sovellustenväliseen kommunikointiin sisällöntarjoajat tarjoavat vakiorajapinnan, joka pitää huolen prosessienvälisestä kommunikoinnista ja tietoturvallisuudesta.

Androidin mukana tulee valmiiksi toteutetut sisällöntarjoajat esimerkiksi musiikille, videotiedostoille ja käyttäjän yhteystiedoille. Muutamia rajoitteita lukuunottamatta nämä sisällöntarjoajat ovat kaikkien sovellusten käytettävissä [andb].

2.7 Aikeet

Suurin osa Android-sovellusten kommunikaatiosta on tapahtumapohjaista. Niin aktiviteetit, palvelut kuin sisällöntarjoajatkin käynnistetään lähettämällä niille aie (*in-*

tent). Tapahtumia käytetään Androidissa, koska niiden avulla komponentit voidaan sitoa toisiinsa ajonaikaisesti ja vasta silloin, kun niitä varsinaisesti tarvitaan. Itse aie-oliot ovat passiivisia tietorakenteita, joissa on abstrakti kuvaus operaatiosta, joka halutaan suoritettavan, tai lähetysten (*broadcast*) tapauksessa kuvaus siitä, mitä on tapahtunut.

Aikeiden kohde voidaan nimetä ekspliittisesti ComponentName-kentässä. Tällöin annetaan kohdekomponentin täydellinen nimi paketteineen, jolloin kohde voidaan tunnistaa yksikäsitteisesti. Tämän muodon käyttäminen vaatii, että kutsuva komponentti tietää kohdekomponentin nimen. Sovelluksensisäisessä kommunikoinnissa tämä onnistuu, mutta sovellustenvälisessä kommunikoinnissa useinkaan ei. Tällöin kohde päätellään implisiittisesti muista aikeelle annetuista kentistä.

Action-kentässä annetaan tapahtuma, joka aikeella halutaan käynnistää, esimerkiksi puhelun aloitus, tai lähetysten vastaanottajien tapauksessa järjestelmässä tapahtunut tapahtuma, kuten varoitus akun loppumisesta. Intent-luokassa määritellään lukuisia vakioita erilaisia tapahtumia varten, mutta niiden lisäksi sovellukset voivat määritellä myös omia tapahtumia.

Data-kentässä annetaan tapahtumaan liittyvän tiedon osoite (URI) ja tyyppi (MIME). Näin vastaanottava komponentti tietää minkätyyppistä tietoa aikeeseen liittyy, ja mistä se löytyy. Category-kentässä kerrotaan, minkä tyyppisen komponentin odotetaan käsittelevän aikeen. Näitäkin Intent-luokka tarjoaa valmiita, mutta omien käyttö on mahdollista.

Aikeen vastaanottava komponentti voidaan päätellä kahdella tavalla. Komponentti valitaan ekplisiittisesti, jos ComponentName-kentässä on arvo. Tällöin muiden kenttien arvoista ei välitetä. Muussa tapauksessa Action-, Data- ja Category-kenttien arvojen perusteella selvitetään, mitä soveltuvia vastaanottavia komponentteja järjestelmään on asennettuna. Tässä käytetään apuna aiesuotimia (Intent filter).

Sovellukset voivat määritellä aiesuotimia, jotta järjestelmä tietää, mitkä sovellukset voivat ottaa vastaan aikeita. Aiesuotimet ovat komponenttikohtaisia, ja ne määrittelevät, mitä tapahtumia, tietotyyppejä ja kategorioita ne tukevat. Aiesuotimia käytetään hyväksi implisiittisessä kohteen määrittelyssä. Jos kohde on määritelty eksplisiittisesti komponentin nimellä, aiesuotimilla ei ole vaikutusta [andb].

2.8 Android manifest

Jokaisella Android-sovellukseen kuuluu AndroidManifest.xml-tiedosto, joka sisältää järjestelmälle välttämätöntä tietoa sovelluksen ajamiseksi. Manifestissa määritellään sovelluksen Java-paketti, joka toimii samalla sovelluksen yksilöllisenä tunnisteena. Manifestissa on myös listattu sovelluksen komponentit, aktiviteetit, palvelut, sisällöntarjoajat ja lähetysten vastaanottajat, joista sovellus koostuu, sekä niiden toiminnallisuus ulkopuolelta tulevien aikeiden kannalta Lisäksi manifestissa ilmoitetaan, mitä oikeuksia sovellus tarvitsee toimiakseen sekä mitä oikeuksia toisilla sovelluksilla pitää olla, jotta ne voivat käyttää kyseisen sovelluksen palveluita. Näiden tietojen lisäksi manifestissa määritellään sovelluksen vaatimuksen ympäristöltään: mikä on sovelluksen vaatima Android APIn minimiversio sekä mitä kirjastoja sovellus tarvitsee toimiakseen. [andb]

Jos sovellus tarvitsee välttämättä laitteelta tiettyjä ominaisuuksia, on mahdollista suodattaa sovellus pois Androidin sovelluskaupan hauista, jos sitä haetaan laitteella, joka ei tue sovelluksen vaatimia ominaisuuksia. Tärkeimmät suotimet ovat androidin API:n minimiversio, tiettyjen lisälaitteiden olemassaolo ja näytön koko.

<uses-sdk>-direktiivillä (directive) määritellään Androidin APIn minimiversio. Jos laitteessa on käytössä pienempi API-versio kuin direktiivillä annettu, sovellusta ei näytetä hakutuloksissa. <support screens>-direktiivi määrittelee, millä näytön koolla sovellus toimii. Tavallisesti määrittelemällä jokin tuettu koko, sovelluskauppa olettaa, että laite tukee sen lisäksi myös isompia näyttökokoja, muttei pienempiä. On myös mahdollista määritellä erikseen kaikki tuetut näyttökoot.

<uses-feature>-direktiivillä voidaan määritellä mitä ominaisuuksia sovellus vaatii. Näitä on sekä laitteistotasolla, kuten kamera, kiihtyvyysanturi tai kompassi, että ohjelmistotasolla, kuten vaikka liikkuvat taustakuvat, joiden pyörittämiseen kaikissa Android-laitteissa ei riitä resursseja. <uses-feature>-direktiiviä käytetään, kun sovellus ei lainkaan toimi ilman kyseistä ominaisuutta [anda].

3 Ohjelmistojen testaaminen ja mobiilisovellusten testaamisen erityispiirteitä

Tässä luvussa esitellään ohjelmistojen testauksen peruskäsitteitä, testauksen asemaa ohjelmistotuotantoprosessissa, mobiilisovellusten testaamisen erityispiirteitä sekä pohditaan miten testaustyökaluja voi arvioida.

3.1 Testaamisen peruskäsitteitä

Testitapaus (test case) on yksittäinen testi, jolle on määritelty syötteet, suoritusehdot ja läpäisykriteerit. Testisarja (test suite) taas on joukko testitapauksia. Testisarja voi myös koostua useasta testisarjasta, jolloin esimerkiksi ohjelman jokaiselle komponentille voi olla oma testisarjansa ja yksi testisarja kattaa sitten kaikki yksittäisten komponenttien testisarjat [PY08, 153].

Yksikkötestaus on useimmiten valkoinen laatikko -testausta (white box testing / structural testing / glass box testing), jolloin testejä voidaan kirjoittaa ohjelmakoodin perusteella. Testeiltä voidaan vaatia esimerkiksi tiettyä koodikattavuutta, jolloin varmistetaan, että mahdollisimman suuri osa ohjelmakoodista tulee suoritettua testien aikana [PY08, 154].

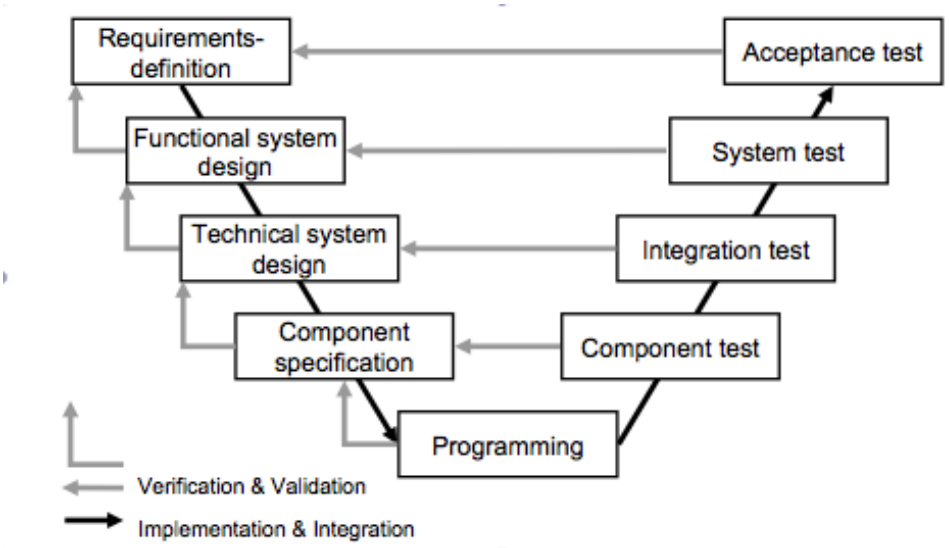
Toiminnallisessa testauksessa (functional testing) testattavan ohjelman sisäistä rakennetta ei tunneta vaan ollaan kiinnostuttu vain syötteistä ja niitä vastaavista tulosteista. Toiminnallista testausta voidaan kutsua myös musta laatikko -testaukseksi (black box testing). Toiminnallisessa testauksessa ollaan kiinnostuttu ohjelman käyttäjälle näkyvästä toiminnasta ja niitä voidaan tehdä esimerkiksi ohjelman määrittelyn pohjalta [PY08, 161-162].

Mallipohjainen testaus (model based testing) on toiminnallisen testauksen alalaji, jossa testitapaukset luodaan automaattisesti ohjelman spesifikaatiosta tehdystä mallista. Jos ohjelma mallinnetaan formaalilla mallilla, kuten äärellisellä automaatilla tai kieliopilla, voidaan testitapaukset generoida täysin automaattisesti. Semiformaaleilla menetelmillä, kuten luokka- tai oliokaavioilla, mallinnetuista ohjelmista generointi saattaa vaatia manuaalista työtä [PY08, 245-250].

Jäljittely (mocking) on tekniikka, joka helpottaa yksikkötestien kirjoittamista. Yksikkötestien ulkoiset riippuvuudet voidaan korvata testeissä kontrolloitavilla jäljittelijäolioilla. Tällöin testit ovat vakaampia, koska ulkopuolisten komponenttien muok-

kaus ei vaikuta testeihin, testin haluttuun lopputulokseen vaikuttava ympäristö on helppo saada haluttuun muotoon. Tällöin testien on helppo testata myös sellaisia olosuhteita, jotka ovat harvinaisia, tai vaikea saada muokattua. Lisäksi jäljittelemällä voidaan korvata vielä tekemättömät ulkoiset riippuvuudet jäljittelijätoteutuksilla [MFC01].

3.2 Testaaminen osana ohjelmistotuotantoprosessia



Kuva 4: Testauksen v-malli

Ohjelmistotuotantoprosessia lähestytään yleensä jonkin elämänkaarimallin kautta. Testauksen näkökulmaa edustaa testauksen v-malli, joka on esitetty kuvassa 4. Mallin yleisessä versiossa lähdetään siitä, että jokaista ohjelmistotuotantoprosessin vaihetta vastaa jokin testausvaihe. Näin testaus tehdään ohjelmistotuotantoprosessissa näkyväksi ja yhtä tärkeäksi osaksi kuin itse sovelluksen kehittäminen. Kuvassa v:n vasemmalla haaralla on vesiputousmallin mukaiset ohjelmistotuotantoprosessin vaiheet. Ensimmäisenä vaiheena on vaatimusmäärittely. Tässä vaiheessa määritellään loppukäyttäjän tai asiakkaan tarpeet ja vaatimukset sovellukselle. Viimeinen vaihe on alimpana oleva sovelluksen ohjelmointi. Oikealla haaralla on testausvaiheet, joissa jokaisessa varmistetaan, että ohjelma täyttää vasemman haaran samalla tasolla olevassa vaiheessa suunnitellut vaatimukset [SLS09, 39-42].

V-mallin alimmalla tasolla on komponentti- eli yksikkötestaus. Siinä testataan ohjelman pienempiä itsenäiseen toimintaan kykeneviä osasia, eli olio-ohjelmoinnin tapauksessa useimmiten luokkia. Android-sovellusten tapauksessa tällä voidaan ajatel-

la yksittäistä Android-komponenttia, esimerkiksi aktiviteettia, tai jopa aktiviteettia pienempiä osia, kuten yksittäistä näkymä-luokkaa tai muuta toimintaa tarjoavaa alikomponenttia. Komponenttitestaus on useimmiten valkoinen laatikko -testausta ja sen suorittaminen vaatii ohjelmointitaitoa. Useimmiten sovelluksen ohjelmoijat suorittavat itse komponenttitestauksen. Komponenttitestauksella pyritään varmistamaan, että komponentti täyttää sen toiminnallisen määritelmän. Komponenttitestejä tehdään usein eri kielille kehitetyillä yksikkötestikehyksillä, kuten Javan JUnitilla. Oikean toiminnallisuuden testaamisen lisäksi on tärkeää testata myös toiminta väärillä syötteillä ja poikkeustilanteissa. Moderni tapa tehdä komponenttitestausta on kirjoittaa testikoodi ennen sovelluskoodia [SLS09, 43-50].

V-mallin seuraavalla tasolla on integraatiotestaus. Siinä vaiheessa oletetaan, että komponenttitestaus on jo tehty ja yksittäisten komponenttien omaan toimintaan liittyvät virheet on löydetty. Integraatiovaiheessa yksittäisten komponentit yhdistetään toisiinsa ja testataan, että niiden yhteistoiminta on oikeanlaista. Tavoitteena on löytää mahdolliset virheet komponenttien rajapinnoista ja komponenttien välisestä yhteistyöstä. Testauksessa voidaan käyttää apuna tynkiä (stub) sellaisista komponenteista, jotka eivät vielä ole valmiita [SLS09, 50-52].

V-mallin kolmannella testaustasolla on järjestelmätestaus (system testing). Järjestelmätestauksessa testataan, että täysin integroitu järjestelmä toimii kokonaisuutena kuten pitäisi. Alemmista tasoista poiketen järjestelmätestauksessa näkökulma on järjestelmän tulevan käyttäjän, kun alemmilla tasoilla testaus on luonteeltaan enemmän teknistä. Järjestelmätestauksessa varmistetaan, että järjestelmä kokonaisuudessaan täyttää sille asetetut toiminnalliset ja ei-toiminnalliset vaatimukset. Järjestelmätestauksessa ei tulisi käyttää enää tynkiä, vaan järjestelmän kaikki komponentit tulisi asentaa esimerkiksi erilliseen testiympäristöön testausta varten. Itse sovelluksen lisäksi järjestelmätestauksen piiriin kuuluu ohjelmiston dokumentaatio ja konfiguraatio [SLS09, 58-61].

V-mallin viimeinen vaihe on hyväksyntätestaus. Alemmilla tasoilla sovellus on ollut kehittäjän vastuulla, mutta hyväksyntätestauksessa järjestelmä testataan asiakkaan näkökulmasta. Tällöin pyritään varmistamaan, että tuotettu järjestelmä täyttää mahdollisen sopimuksen, että sen käyttäjät hyväksyvät sen käytön sekä mahdollisesti alfa tai beta -testauksen oikeilla järjestelmän loppukäyttäjillä.[SLS09, 62-63]

Testauksen jokaisessa vaiheessa ollaan kiinnostuneita validoinnista ja verifioinnista. Validoinnissa varmistetaan, että toteutus täyttää järjestelmän alkuperäisen tehtävän, eli että rakennetaan oikeaa sovellusta. Verifioinnissa taas varmistetaan, että

kehitysvaiheessa tuotettu sovelluksen osa vastaa sille tehtyä spesifikaatiota, eli että sovellus on tehty oikein. V-mallin eri vaiheissa validoinnin ja verifikaation painotukset vaihtelevat. Alemman tason testauksessa on kyse enemmän verifikaatiosta ja ylemmän tason testauksessa taas validoinnista [SLS09, 41-42].

3.3 Mobiilisovellusten testaamisen erityispiirteitä

Mobiilisovellusten kehittämiseen liittyy haasteita, jotka liittyvät mobiiliympäristön rajoituksiin. Näitä on muunmuassa päätelaitteiden rajallinen kapasiteetti ja jatkuva kehitys, erilaiset standardit, protokollat ja verkkotekniikat, päätelaitteiden monimuotoisuus, mobiililaitteiden käyttäjien erikoistarpeet sekä tiukat aikavaatimukset sovellusten saamiseksi markkinoille.

Sovelluksia rajoittaa myös mobiililaitteiden fyysinen koko ja laitteiden monimuotoinen koko, paino ja näytön koko. Myös käyttöliittymissä on eroja, joskin kosketusnäyttöpuhelimet ovat vallanneet suurimman osan markkinoista viime vuosina.

Mobiilisovelluksen on oltava laadultaan hyvä, jotta se on helppo saada toimimaan oikein erilaisissa laiteympäristöissä. Lisäksi julkaisunopeus voi olla kriittinen tekijä markkinaosuuden valtaamisessa. Jos kilpaileva sovellus julkaistaan viikkoa aikaisemmin, voi markkina olla jo täytetty sovelluksen julkaisuhetkellä.

VTT on kehittänyt mobiilisovellusten kehittämiseen ketterän Mobile-D-lähestymistavan. Testauksen kannalta olennaista on testilähtöisen kehityksen käyttäminen (test driven development, tdd). Mobile-D:hen kuuluu testien kirjoittaminen ennen tuotantokoodin kirjoitusta, yksikkötestien automatisointi ja kaikkien ominaisuuksien hyväksyntätestaus asiakkaan kanssa [AHH+04].

Testilähtöisessä kehityksessä on kaksi merkittävää periaatetta: ohjelmakoodia saa kirjoittaa vain automaattisen testin korjaamiseksi ja duplikaattikoodin poistamiseksi. Näistä periaatteista seuraa tunnettu tdd-sykli: punainen, vihreä, refaktoroi. Ensin kirjoitetaan testi, joka ei mene läpi, koska testin toteuttavaa ohjelmakoodia ei ole vielä olemassa. Vaiheen nimi on punainen, koska useimmilla yksikkötestityökaluilla lopputuloksena näkyy punainen palkki, jos jokin testi ei mene läpi. Toinen vaihe on kirjoittaa juuri sen verran koodia, mitä tarvitaan testin läpäisemiseksi. Tässä vaiheessa ei välitetä miten rumaa koodi on. Vaiheen nimi on vihreä, koska useimmillaa yksikkötestityökaluilla lopputuloksena näkyy vihreä palkki, kun testit menevät läpi. Viimeisessä vaiheessa refaktoroidaan toisessa vaiheessa mahdollisesti syntynyt duplikaatti- tai muuten ruma koodi. Testit auttavat varmistamaan, ettei

refaktoroidessa hajoiteta vanhaa toiminnallisuutta. Jotta tällainen ohjelmointisykli olisi mahdollinen, ohjelmistoympäristön täytyy tarjota mahdollisuus saada nopeasti palaute pienestä testijoukosta, jottei ohjelmoidessa jouduta jatkuvasti odottamaan testien ajautumista [Bec03].

3.4 Testityökalujen arviointikriteereistä

Androidille on tehty Androidin mukana tulevien testaustyökalujen lisäksi monia muita testaustyökaluja. Nämä työkalut erottautuvat Androidin työkaluista joko pyrkimällä toteuttamaan jonkin asian paremmin kuin vastaava Androidin oma testaustyökalu tai sitten tarjoamalla sellaisen lähestymistavan testaamiseen, mitä Androidin omat testaustyökalut eivät tarjoa.

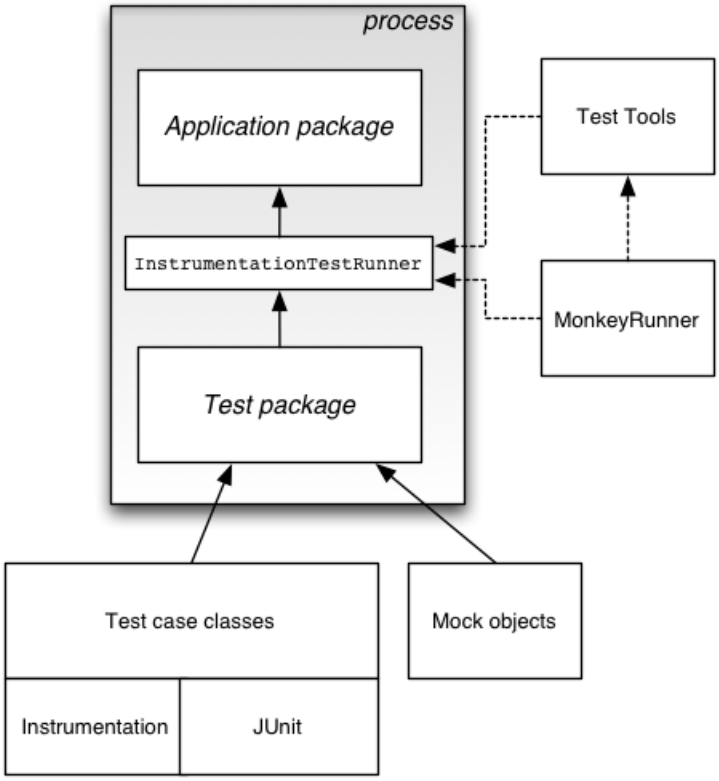
Testaustyökalujen arviointia käsittelevät muunmuassa Poston ja Sexton [PS92]. Hyvän testaustyökalun kriteerit ovat osin kontekstista riippuvia, mutta Poston ja Sexton määrittelevät myös yleisempiä kriteerejä testityökalujen arviointiin. Työkalun toiminnallisten ominaisuuksien arviointi on yleensä kontekstiriippuvaista, mutta usein myös helposti määriteltävissä. Jollain työkalulla pystyy testaamaan asioita, joita toisella ei voi. Ei-toiminnallisia olennaisia ominaisuuksia he luettelevat työkalun tehokkuuden, miten nopeasti sen käytön oppii, miten nopeaa testien tekeminen sillä on ja miten luotettava työkalu on.

Michael et al. [MBS02] ovat kehittäneet joukon metriikoita testityökalun tehokkuuden arviointiin. Heidän käyttämänsä metriikat ovat saaneet innoituksensa tavallisille sovelluksille kehitetyistä erilaisista kompleksisuusmetriikoista, kuten koodirivien määrä tai syklomaattinen kompleksisuus. He esittävät listan vastaavia metriikoita testityökalujen tehokkuuden arviointiin. Näitä arvoja voi sitten painottaa testityökalujen valinnassa haluamallaan tavalla. Osa metriikoista ei ole enää relevantteja nykyisten testaustyökalujen kannalta. Metriikoita on muunmuassa työkalun kypsyys ja käyttäjäkunnan koko, helppokäyttöisyys, kustomointimahdollisuudet, automaattinen testitapausten generointi, muiden ohjelmointityökalujen tarjoama tuki työkalulle, luotettavuus sekä suoritusnopeus. Osalle metriikoista tarjotaan täsmällisiä laskukaavoja, jotka helpottavat kvantitatiivisen analyysin tekemistä.

4 Android-sovellusten testaaminen ja Androidin mukana tulevat testityökalut

Androidin sovelluskehityspaketin mukana tulee monia Googlen kehittämiä testaustyökaluja. Esittelen niitä tässä luvussa. Samalla tutustutaan Android-sovelluksen testaamisen perusteisiin.

4.1 Android-testien ajoympäristö



Kuva 5: Android-testien ajoympäristö

Androidin-sovelluksen testejä voi ajaa joko emulaattorilla tai suoraan puhelimessa. Androidin testisarjat perustuvat JUnit-testikehikkoon. Puhdasta JUnitia voi käyttää sellaisen koodin testaamiseen, joka ei kutsu Androidin rajapintaa tai sitten voi käyttää Androidin JUnit-laajennusta Android-komponenttien testaukseen. Laajennus tarjoaa komponenttikohtaiset yliluokat testitapausten kirjoittamista varten. Nämä luokat tarjoavat apumetodeita jäljittelijöiden (mock object) luomiseen ja komponentin elinkaaren hallintaan. Androidin JUnit-toteutus mahdollistaa JUnitin versio

3:n mukaisen testityylin, ei uudempaa versio 4:n mukaista.

Kuvassa 5 on esitetty Androidin testien ajoympäristö. Testattavaa sovellusta testataan ajamalla testipaketissa olevat testitapaukset MonkeyRunnerilla (ks. luku 4.3). Testipaketti sisältää testitapausten lisäksi Androidin instrumentaatiota, eli apuvälineitä sovelluksen elinkaaren hallintaan ja koukkuja, joilla järjestelmän lähettämiä callback-metodikutsuja pääsee muokkaamaan, sekä jäljittelijäolioita korvaamaan järjestelmän oikeita luokkia testin ajaksi jäljittely-toteutuksella. [andb]

4.2 Komponenttikohtaiset yksikkötestiluokat

Android tarjoaa aktiviteeteille, palveluille ja sisällöntarjoajille jokaiselle oman testiyliluokkansa, joka mahdollistaa komponenttikohtaisten testien helpomman toteutuksen.

Aktiviteettien testauksessa Androidin JUnit-laajennus on merkittävä, koska aktiviteeteilla on monimutkainen elinkaari, joka perustuu paljolti takaisinkutsumetodeihin, joiden suora kutsuminen ei ole mahdollista. Aktiviteettien testauksen pääyliluokka on InstrumentationTestCase. Sen avulla on mahdollista käynnistää, pysäyttää ja tuhota testattavana oleva aktiviteetti halutuissa kohdissa. Lisäksi sen avulla voi jäljitellä järjestelmäolioita, kuten Context- ja Applications-luokan instansseja. Tämä mahdollistaa testin eristämisen muusta järjestelmästä ja aikeiden luomisen testejä varten. Lisäksi yliluokassa on metodit käyttäjän vuorovaikutusten, kuten kosketus- ja näppäimistötapahtumien lähettämiseen suoraan testattavalle luokalle.

Aktiviteettien testaamiseen on kaksi välitöntä yliluokkaa, ActivityUnitTestCase ja ActivityInstrumentationTestCase2. ActivityUnitTestCase on tarkoitettu luokan yksikkötestaamiseen siten, että se on eristetty Android-kirjastoista. Näitä testejä voi ajaa suoraan kehitystyökalu ja tarvittaessa Android-kirjaston mockaamiseen on käytössä MockApplication-olio. ActivityInstrumentationTestCase2 taas on tarkoitettu toiminnalliseen testaukseen tai useamman aktiviteetin testaamiseen. Ne ajetaan normaalissa suoritusympäristössä emulaattorilla tai Android-laitteessa. Aikeiden jäljittely on mahdollista, mutta testin eristäminen muusta järjestelmästä ei ole mahdollista.

Palveluiden testaaminen on paljon yksinkertaisempaa kuin aktiviteettien. Ne toimivat eristyksessä muusta järjestelmästä, joten testattaessakaan ei tarvita Androidin instrumentaatiota. Android tarjoaa ServiceTestCase-yliluokan palveluiden testaamiseen. Se tarjoaa jäljittelijäoliot Application- ja Context-luokille, joten palvelun

saa testattua eristettynä muusta järjestelmästä. Testiluokka käynnistää testattavan palvelun vasta kutsuttaessa sen startService() tai bindService()-metodia, jolloin jäljittelijäoliot voi alustaa ennen palvelun käynnistymistä. Jäljittelijäolioiden käyttö palveluiden testaamisessa paljastaa myös mahdolliset huomaamatta jääneet riippuvuudet muuhun järjestelmään, koska Jäljittelijäoliot heittävät poikkeuksen, mikäli niihin tulee metodikutsu, johon ei ole varauduttu.

Sisällöntarjoajien testaaminen on erityisen tärkeää, jos sovellus tarjoaa sisällöntarjoajiaan muiden sovellusten käyttöön. Tällöin on myös olennaista testata niitä käyttäen samaa julkista rajapintaa, jota muut sovellukset joutuvat käyttämään kommunikoidessaan sisällöntarjoajien kanssa. Sisällöntarjoajien testauksen yliluokka on ProviderTestCase2, joka tarjoaa käyttöön jäljittelijäoliot ContentResolveriosta ja Contextista, jolloin sisällöntarjoajia voi testaja eristyksissä muusta sovelluksesta. Yliluokka tarjoaa myös metodit sovelluksen oikeuksien testaamisen. Contextin jäljittelijäolio mahdollistaa tiedosto- ja tietokantaoperaatiot, mutta muut Androidin kirjastokutsut on toteutettu tynkinä (stub). Lisäksi tiedon kirjoitusosoite on uniikki testissä, joten testien ajaminen ei yliaja varsinaista sovelluksen tallentamaa tietoa. Sisällöntarjoajatestit ajetaan emulaattorissa tai Android-laitteella. [andb]

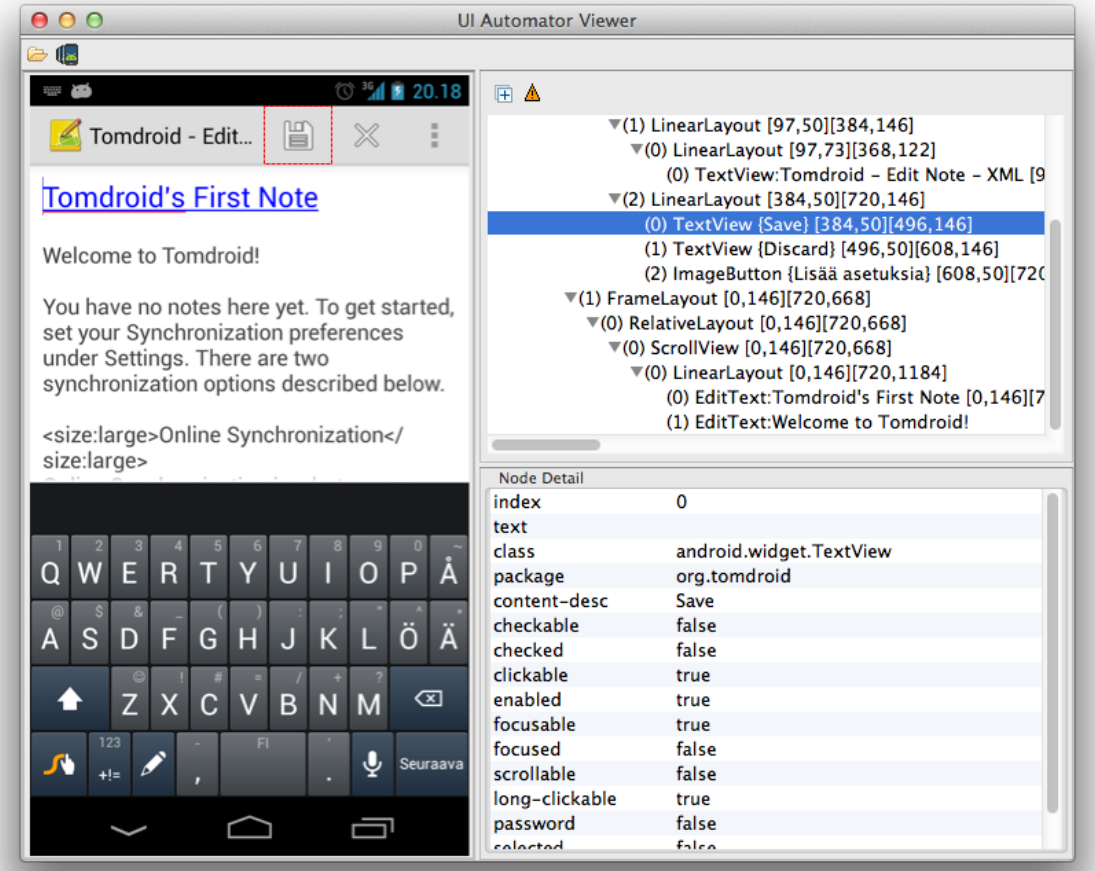
4.3 MonkeyRunner

MonkeyRunner tarjoaa rajapinnan, jolla Android-sovellusta voi ohjata laitteessa tai emulaattorissa. Se on lähinnä tarkoitettu toiminnallisten testien sekä yksikkötestien ajamiseen, mutta soveltuu myös muihin tarkoituksiin. Sen avulla voi esimerkiksi asentaa sovelluksia, ajaa testisarjoja ja sovelluksia ja lähettää niihin syötteitä. Lisäksi monkeyrunnerilla voi ottaa eri kohdista kuvakaappauksia ja verrata niitä referenssikuviin. Tällä tavalla voidaan tehdä esimerkiksi regressiotestausta.

MonkeyRunnerilla voidaan testata yhtä aikaa monia eri emulaattoreita tai useita laitteita, jolloin voidaan tehdä fragmentaatiotestausta. MonkeyRunner on myös laajennettavissa, jolloin sitä voi käyttää muihinkin tarkoituksiin. MonkeyRunneria ohjataan pythonilla ja se on toteutettu jythonilla, joka on Javan virtuaalikoneessa pyörivä python-toteutus [andb].

4.4 Uiautomator ja uiautomatorviewer

Uiautomator tarjoaa mahdollisuuden kirjoittaa toiminnallisia mustalaatikko-testejä Android-sovelluksille Monkeyrunneria paremmat assert-mahdollisuudet ja mahdollisuuden toiminallisten mustalaatikko-testien kirjoittamiseen Javalla tarjoaa uiautomator. Siinä on kaksi komponenttia: uiautomatorviewer ja uiautomator. Uiautomatorviewer on graafinen työkalu, jolla voi analysoida ja skannata Android-sovelluksen käyttöliittymää ja uiautomator työkalu, joka tarjoaa rajapinnan ja moottorin toiminnallisten mustalaatikkotestien ajamiseen.



Kuva 6: UIAutomatorViewerin käyttöliittymä

Uiautomatorviewer (kuvassa 6) analysoi sovelluksen näkymän komponentit. Vasemmalla näkymässä punaisella ympyröitynä on oikeasta valikosta valittu komponentti. Oikeassa alareunassa näytetään komponentin tietoja, joita voi käyttää testeissä esimerkiksi komponentin valitsijoissa, esimerkiksi siten, että valitaan save-nappi tekstin

perusteella, jotta siitä voidaan testissä painaa.

UIAutomator-testit kirjoitetaan Javalla JUnitiin pohjautuen perimällä UIAutomatorTestCase-yliluokka, joka tarjoaa käyttöön käyttöliittymäelementtien valitsemiseen tarvittavan rajapinnan ja mahdollisuuden vuorovaikutukseen niiden kanssa. UISelector-luokassa on metodeita käyttöliittymäelementtien valitsemiseen ja UIObject-luokassa niiden kanssa kommunikointia varten. UIObject-oliolta voi kysyä assertointia varten elementin olemassaoloa ja erilaisia tilaan liittyviä tietoja, kuten onko elementti käytössä tai painettavissa.

Uiautomator on uusi työkalu ja testejä voi ajaa vain laitteilla, jotka tukevat API:n versiota 16 tai uudempaa. Tämä vastaa Androidin versiota 4.1, joka on julkaistu kesäkuussa 2012. Uiautomatorilla ei voi siis testata sovellusta vanhemmilla Android-laitteilla, joita on tällä hetkellä vielä reilusti yli puolet Androidin laitekannasta [andb].

4.5 Monkey

Monkey on Androidin mukana tuleva työkalu, jota voi ajaa emulaattorissa tai Android-laitteessa ja joka tuottaa sovellukselle pseudosatunnaisia syötteitä. Näitä voi olla esimerkiksi painallukset, eleet sekä järjestelmätason viestit. Monkeytä voi käyttää esimerkiksi sovelluksen stressitestaukseen tai fuzz-testaukseen.

Monkeylle voi antaa jonkin verran sen toimintaa ohjaavia parametreja. Ensinnäkin testisyötteiden määrää ja tiheyttä voi rajoittaa. Toiseksi erityyppisten syötteiden osuutta voi säätää. Kolmanneksi testauksen voi rajoittaa tiettyyn pakettiin sovelluksessa. Tällöin Monkey pysäyttää testauksen, jos se on ajautuu muihin kuin haluttuun osaan sovelluksesta. Neljänneksi Monkeyn tulosteiden määrää ja tarkkuutta voi säätää.

Monkey pystäyttää testin, jos ohjelmasta lentää käsittelemätön poikkeus tai jos järjestelmä lähettää sovellus ei vastaa -virheviestin. Näissä tapauksissa Monkey antaa raportin virheestä ja miten se syntyi. Monkey voi myös tehdä profilointiraportin testistä.[andb]

5 Yksikkötestityökalujen vertailua

Androidin oma yksikkötestaustapa on tehdä JUnit-testejä Androidin oman AndroidUnitTestCase-luokan aliluokkana. Nämä testit ajetaan emulaattorissa Dalvik-ympäristössä. Tässä tavassa on kaksi heikkoutta, jonka takia on olemassa myös kolmansien osapuolien yksikkötestityökaluja Android-ympäristöön. Ensinnäkin testien ajaminen emulaattorissa on hitaampaa kuin jos niitä voisi ajaa suoraan standardissa Javan virtuaalikoneessa JUnit-testeinä. Toiseksi muutkaan testauksen apuna käytettävät työkalut, kuten jäljittelijä-työkalut, eivät välttämättä toimi ongelmitta Dalvik-ympäristössä. Tässä luvussa vertailen Android-sovelluksen yksikkötestausta AndroidUnitTestCasella ja suosituimmalla Javan virtuaalikoneessa pyörivällä vaihtoehdolla: Robolectricillä. Tavoitteena on selvittää, voiko Robolectricillä helposti korvata Androidin oman yksikkötestikehyksen ja pitääkö Robolectricin lupaus nopeammasta testien suoritusajasta paikkansa.

Yksikkötesteissä kiinnostavaa on nimenomaan Androidin keskeisten komponenttien testaus, koska ohjelmassa mahdollisesti olevat muut kuin Androidin kirjastoluokista perivät luokat on helppo testata tavanomaisilla Javan yksikkötestityökaluilla ilman erityisesti Androidille tarkoitettuja työkaluja.

5.1 Robolectric

Robolectric on yksikkötestaustyökalu, jonka tarkoitus on mahdollistaa Android-koodin yksikkötestaus suoraan ohjelmointiympäristössä Javan virtuaalikoneessa ilman emulaattoria. Tarkoitus on mahdollistaa nopea TDD-sykli ja helpompi integrointi jatkuvan integroinnin palveluihin. Normaalisti Android-kirjaston luokat palauttavat kutsuttaessa ohjelmointiympäristöstä ajoaikaisen poikkeuksen, mutta Robolectric korvaa nämä luokat varjototeutuksilla, jotka palauttavat poikkeuksen sijaan tyhjän oletusvastauksen, kuten null, 0 tai false, tai jos Robolectricissa on kyseistä metodia varten olemassa varjototeutus, se palauttaa toteutuksen määrittelemän oikeamman paluuarvon.

Robolectricin varjoluokkien käytön vaihtoehtona on jonkin jäljittelijäkehyksen käyttäminen Androidin kirjaston korvaamiseen, mutta tämä on hyvin työläs ja verboosi tapa kirjoittaa testejä. Lisäksi tällöin testejä kirjoittaessa täytyy tuntea testattavan metodin toiminta hyvin tarkasti, jotta jäljittelijätoteutukset saadaan kirjoitettua. Robolectricin varjoluokat mahdollistavat enemmän musta laatikko -tyyppisen testauksen [roba].

5.2 Aiempaa tutkimusta Robolectricistä

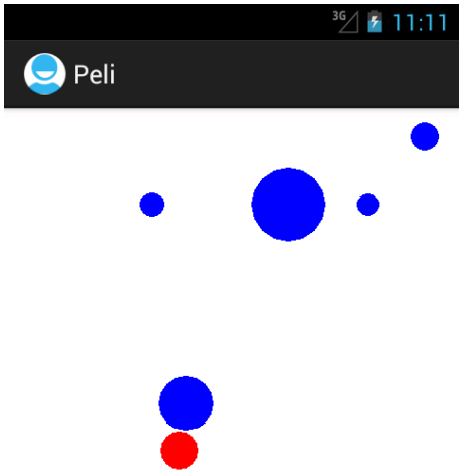
Sadeh et al. vertasivat Androidin aktiviteettien yksikkötestausta JUnitilla, Androidin omilla yksikkötestityökaluilla sekä Robolectricilla. JUnitilla testattaessa ongelmaksi muodostui, että ohjelmakoodia jouduttiin melko rajusti muokkaamaan, jotta luokkien yksikkötestaus onnistui. Tämä tekee ohjelman ylläpidon vaikeaksi. JUnitin hyvä puoli oli erittäin nopea testien ajonopeus. Robolectricillä testien tekeminen taas oli lähes yhtä helppoa kuin Androidin omilla työkaluilla. Androidin työkaluihin verrattuna Robolectricin vahvuudet olivat virhepaikkojen paikantamisen helppous ja nopea ajonopeus. Robolectric-testit ajautuivat viisi kertaa nopeammin kuin Androidin työkaluilla ajetut testit, koska Androidin työkaluilla kirjoitetut testit ajetaan emulaattorilla Dalvikilla, Robolectric taas suoraan Javalla. Androidin omien työkalujen suurin vahvuus oli testien kirjoittamisen helppous [S+11].

Sadeh et al. eivät käyttäneet JUnit-testeissään mitään jäljittelijätyökalua, joten testeissä testiluokan riippuvuudet jäljiteltiin tekemällä niistä staattisia sisäluokkia testiluokan sisään. Tämä on erittäin verboosi tapa ja vaikutti osaltaan siihen, miksi puhdas JUnit-testi näytti niin hankalalta. Androidin kirjastoluokat on pakko eristää testattavasta luokasta Javan virtuaalikoneella testattaessa, koska Androidin kehitysympäristössä käytettävä paketti ei sisällä varsinaisesti luokkien sisältöä, vaan vain niiden luokkien julkiset rajapinnat, jolloin kehitystyökalut osaavat auttaa niiden käytössä, mutta varsinaista toteutusta ei ole.

Jeon & Foster mainitsevat Robolectricin vahvuudeksi sen, että se pyörii Javan virtuaalikoneessa ja näin ohittaa hitaan vaiheen testeistä, kun sovellus pitää kääntää emulaattorille tai laitteelle testattavaksi. Robolectric ei heidän mielestään kuitenkaan sovellu kokonaisten sovellusten testaamiseen, koska sen varjoluokat eivät toteuta Androidin komponenteista kuin osan. Ylipäänsä Robolectric vertautuu Jeonin ja Fosterin mielestä paljolti Robotiumiin [JF12].

Allevato & Edwards käyttivät Robolectriciä opetuskäyttöön tarkoitetun RoboLIFT-työkalun kehitykseen. Robolectric auttoi heitä ohittamaan emulaattorin käytön ja nopeuttamaan opiskelijoiden testisykliä ja automaattista arviointialgoritmia. Tässä käytössä Robolectricin ongelma oli, että se ohittaa käyttöliittymän piirtämisen kokonaan ja muunmuassa näkymien onDraw()-metodia ei kutsuta ollenkaan. Tämän seurauksena esimerksi näkymän leveys on aina 0 pikseliä, jolloin sellaiset testit, joilla haluttiin klikata näytöllä johonkin suhteelliseen kohtaan (vaikkapa keskelle) eivät toimineet oikein [AE12].

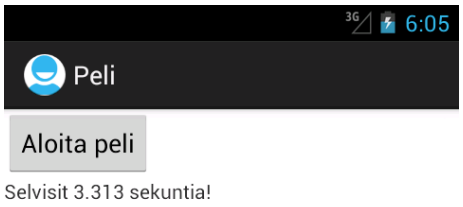
5.3 Testiprojekti



Kuva 7: Ruutukaappaus testiohjelman pelinäkymästä

Yksikkötestaustyökaluja testasin itse tehdyllä demoprojektilla. Kyseessä on yksinkertainen peli, jonka pelinäkymästä on ruutukaappaus kuvassa 7. Pelissä ohjataan kosketusnäytöllä painamalla punaista palloa ja pyritään väistämään ympäriinsä pomppivia sinisiä palloja. Kun peli päättyy, palataan takaisin päänäkymään, josta on ruutukaappaus kuvassa 8. Tästä näkymästä voi aloittaa uuden pelin ja lisäksi näkee, montako sekuntia edellinen peli kesti.

Peli koostuu kahdesta aktiviteetista, yksinkertaisemmasta MainActivitysta sekä hieman monimutkaisemmasta GameActivitysta, jonka yksikkötestaukseen keskityn. Itse peliä ohjaa GameView-luokka, joka on yhtä aikaa näkymä ja kontrolleri MVC-suunnittelumallin mukaisesti. Malleja ovat GameClock, joka kuvaa pelikelloa, sekä Circle, joka kuvaa yhtä ruudulla näkyvää ympyrää. GameActivity toteuttaa lisäksi OnGameEndListener-rajapinnan, jonka avulla GameView ilmoittaa pelin päättymisestä ja pistemäärästä. Pelin luokkakaavio on esitetty kuvassa 9.

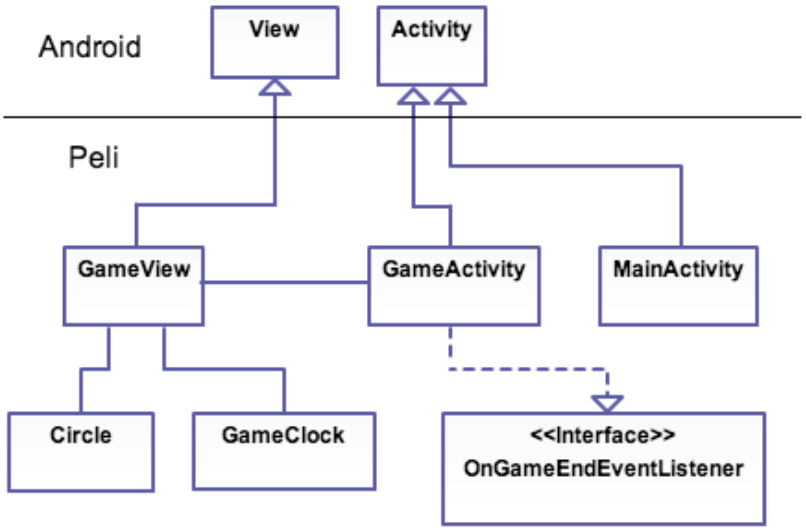


Kuva 8: Testiohjelman päänäkymä

5.4 Robolectricin asentaminen

Robolectricin suositeltu asennustapa on Maven, joka on yleisesti käytössä oleva kääntämis- ja riippuvuuksienhallintatyökalu [mav]. Koska testattava projekti ei ollut Maven-projekti, jouduin tekemään Robolectric-testitkin ilman Mavenia. Ilman Mavenia asennus osoittautui haastavaksi: eri kirjastopakettien riippuvuudet eivät tahtoneet mitenkään toimia yhteen. Lopulta asennus kuitenkin onnistui tekemällä Robolectricin TestRunnerista oma aliluokka. Listauksessa 1 on muokattu Robolectricin aliluokka testejä varten lähteenä olleesta esimerkistä [sam]. Olennaista on, että yliluokalle syötetään konstruktoriparametrina RobolectricConfig-olio, jolle annetaan parametrina testattavan Android-sovelluksen AndroidManifest.xml sekä resurssi-hakemiston osoite.

Robolectric-testejä varten luodaan oma tavallinen Java-projekti Eclipsessä, joka laitetaan viittaamaan Android-projektin koodiin. Testiprojektin build pathiin tarvitaan Robolectricin jar-paketti, sekä Androidin android.jar sekä maps.jar -paketit. Robolectricin paketin pitää olla riippuvuuslistassa ennen Android-paketteja, jotta se toimii. Tämän jälkeen testejä voi ohjelmoida kuten tavallisia JUnit-testejä.



Kuva 9: Testiohjelman luokkakaavio

5.5 Perusominaisuudet

Aktiviteetin elinkaarimetodien testaus on olennaisin osa Android-sovellusten yksikkötestauksesta.

Listauksessa 2 on yksinkertainen aktiviteettiyksikkötesti Robolectricillä toteutettuna. Robolectricin testit ovat yhteensopivia JUnitin 4. version kanssa, mikä mahdollistaa annotaatioiden käytön testeissä. @RunWith-annotaatiolla määritellään testien ajossa käytettävä testiajuri (runner). Testissä käytetään luvussa 5.4 esiteltyä ajuria. @Before-annotaatiolla ilmaistaan metodit, jotka on ajettava ennen testejä ja @Test-annotaatiolla ajettavat testit.

setUp()-metodissa alustetaan testattava aktiviteetti ja kutsutaan sen onCreate()-metodia. Robolectric havaitsee automaattisesti Androidin kirjastometodien kutsun ja korvaa ne Robolectricin toteutuksella, jotta ne toimivat testissä. Siksi aktiviteetti voidaan alustaa suoraan konstruktorilla ja kutsua sen onCreate()-metodia toisin kuin AndroidUnitTestissä, jossa aktiviteetti käynnistetään AndroidUnitTestCase-luokan tarjoamien apumetodien avulla.

Ensimmäisessä testissä testataan, että aktiviteetin onCreate()-metodista alustetaan GameView-olio Running-tilaan. Robolectricin toteutus findViewById()-metodista mahdollistaa View-olion löytämisen Android-sovelluksen resursseissa määritellyn tunnisteen perusteella. Tämän jälkeen varmistetaan, että GameViewin tila on todella vaihtunut Running-tilaan.

Listing 1: CustomRobolectricTestRunner

```
public class CustomRobolectricTestRunner
        extends RobolectricTestRunner {
  public CustomRobolectricTestRunner(
        @SuppressWarnings("rawtypes") Class testClass)
        throws InitializationError {
         super(testClass, new RobolectricConfig(
                        new File("../Demo/AndroidManifest.xml"),
                        new File("../Demo/res")));
  }
}
```

Toinen testi on rakenteeltaan hyvin samanlainen: siinä testataan, että onPause()-elinkaarimetodin kutsuminen siirtää GameViewin pause-tilaan. Testimekaniikka on sinänsä täsmälleen samanlainen kuin ensimmäisessä testissä.

Nämä testit eivät vaadi mitenkään ottamaan huomioon, että testejä tehdään Robolectriciä, eikä oikeaa Androidia vastaan. Robolectric toimii taustalla automaattisesti ja mahdollistaa testien vaatimat Androidin kirjastokutsut.

Listauksessa 3 on toteutettu vastaavat testit kuin listauksessa 2. Itse testit ovat täysin identtisiä Robolectric-testien välillä, erot ovat alustuksessa. Androidin aktiviteettiyksikkötestit perivät yliluokan ActivityUnitTestCase, jolle annetaan konstruktoriparametrina testattavan aktiviteetin luokka. Tämän jälkeen yliluokka instrumentoi luokan testausta varten.

setUp()-metodissa kutsutaan yliluokan setUp()-metodia ja käynnistetään testattava aktiviteetti yliluokan tarjoamalla startActivity()-metodilla. Tämän jälkeen viite testattavaan aktiviteettiin saadaan yliluokan getActivity()-metodilla.

Robolectricistä poiketen Androidin yksikkötestit ovat JUnit3-pohjaisia, joten annotaatioita ei käytetä, vaan metodien nimien perusteella päätellään setUp()-metodi sekä testimetodit siitä, että niiden nimi alkaa sanalla test. Koska JUnit3-testit kutsuvat testien alustamiseksi nimenomaan setUp()-nimistä metodia, on metodin alussa muistettava kutsua yliluokan setUp()-metodia, tai yliluokan suorittamat alustukset jäävät tekemättä.

Listauksessa 4 on hieman monimutkaisempi testitapaus samasta testiluokasta kuin

Listing 2: Yksinkertainen aktiviteettiyksikkötesti Robolectricilla

```
@RunWith(CustomRobolectricTestRunner.class)
public class GameActivityTest {

  private GameActivity activity;

  @Before
  public void setUp() {
    activity = new GameActivity();
    activity.onCreate(null);
  }

  @Test
  public void testActivityInitializesViewWithRunningState()
                  throws Exception {
    GameView gameView =
          (GameView) activity.findViewById(R.id.gameview);
    assertThat(gameView.getState(),
                equalTo(GameState.RUNNING));
  }

  @Test
  public void testOnPauseStopsTheGame() {
        activity.onPause();
        GameView gameView =
              (GameView) activity.findViewById(R.id.gameview);
        assertThat(gameView.getState(),
                    equalTo(GameState.PAUSED));
  }

}
```

Listing 3: Yksinkertainen aktiviteettiyksikkötesti ActivityUnitTestCasen avulla

```
public class GameActivityTest
        extends ActivityUnitTestCase<GameActivity> {

  private GameActivity activity;

  public GameActivityTest() {
        super(GameActivity.class);
  }

  public void setUp() throws Exception {
        super.setUp();
        startActivity(
            new Intent(getInstrumentation().getTargetContext(),
                       GameActivity.class), null, null);
    activity = (GameActivity)getActivity();
  }

  public void testActivityInitializesViewWithRunningState() {
    GameView gameView =
        (GameView) activity.findViewById(R.id.gameview);
    assertThat(gameView.getState(),
                equalTo(GameState.RUNNING));
  }

  public void testOnPauseStopsTheGame() {
        activity.onPause();
        GameView gameView =
            (GameView) activity.findViewById(R.id.gameview);
        assertThat(gameView.getState(),
                    equalTo(GameState.PAUSED));
  }

}
```

Listing 4: Aikeen tilatietojen tarkastelu Robolectricin varjo-olioilla

```
@Test
public void testMainActivityIsCalledAfterLostGame() {
  GameView gameView =
      (GameView) activity.findViewById(R.id.gameview);
  gameView.setState(GameState.LOST);

  ShadowActivity shadowActivity = shadowOf(activity);
  Intent startedIntent =
      shadowActivity.getNextStartedActivity();
  assertThat(startedIntent.getComponent().getClassName(),
          equalTo(MainActivity.class.getName()));
  assertThat(startedIntent.getStringExtra(GameActivity.SCORE),
          equalTo("0.0"));
}
```

listaus 2. Tässä testissä testataan tapahtumia pelin loppuessa. Jos aktiviteetti toimii oikein, se rekisteröi GameView-oliolle havainnoija-suunnittelumallin (observer) mukaisen havainnoijan, jota kutsutaan, kun peli päättyy. Aktiviteetin pitäisi tämän jälkeen pyytää GameView-oliolta pistemäärä ja lähettää uusi aie, jolla käynnistetään MainActivity-aktiviteetti ja annetaan aikeelle ylimääräisenä tietona saatu pistemäärä.

Androidin Activity-luokka ei tarjoa suoraa julkista metodia, jolla voitaisiin kysyä, minkä aikeen aktiviteetti on lähettänyt. Tällaisia tapauksia varten Robolectricilla on valmiina varjototeutus, joka kopioi oikean Android-luokan tilan ja tarjoaa laajemman mahdollisuuden aktiviteetin tilan kyselyyn. Tätä varten testissä tehdään varjo-olio testattavasta aktiviteetista, jolloin varjo-oliolta voidaan kysyä getNextStartedActivity()-metodilla, mikä on seuraava käynnistettävä aktiviteetti. Tältä aikeelta voidaan sitten tarkistaa, että aktiviteetti lähetti aikeen MainActivity-aktiviteetin käynnistämiseksi ja ylimääräisenä tietona on pistemäärä. Tässä tapauksessa pisteiden oletetaan olevan 0.0, koska pelikelloa ei missään testin vaiheessa käynnistetty.

Listauksessa 5 on toteutettu vastaava testi AndroidUnitTestCase:n avulla. Alustus tehdään testissä kuten Robolectric-testissäkin, mutta toiminnan varmistus on yksin-

Listing 5: Aikeen tilatietojen tarkastelu ActivityUnitTestCasen avulla

```
public void testMainActivityIsCalledAfterLostGame() {
  GameView gameView = (GameView) activity.findViewById(R.id.gameview);
  gameView.setState(GameState.LOST);

  Intent startedIntent = getStartedActivityIntent();
  assertThat(startedIntent.getComponent().getClassName(),
             equalTo(MainActivity.class.getName()));
  assertThat(startedIntent.getStringExtra(GameActivity.SCORE),
             equalTo("0.0"));
}
```

kertaisempaa kuin Robolectricillä, koska AndroidUnitTestCase tarjoaa getStartedActivityIntent()-metodin, jolla saadaan aktiviteetin viimeisin lähettämä aie palautettua samalla tavalla kuin Robolectricin varjo-oliolta kysyttiin edellisessä listauksessa. Tämän jälkeen testin läpipääsyn varmistus tapahtuu täsmälleen samalla tavalla kuin Robolectric-testissä.

5.6 Toiminta jäljittelijäkehysten kanssa

Yksikkötestausta pyritään useimmiten tekemään niin, että testiluokka on eristetty riippuvuuksistaan. Apuvälineenä eristämisessä käytetään usein jäljittelijäkehyksiä. Näissä testeissä jäljittelijäkehyksenä käytetään Mockitoa, koska se toimii myös Androidissa sellaisenaan [moc].

Luvussa 5.5 esitetyt yksikkötestit ovat riippuvaisia GameView-luokan toteutuksesta. Testit voivat kuitenkin palauttaa virheellisen tuloksen, jos GameView-luokan setState()-metodi ei muutakaan onnistuneesti tilaa. Tällöin kyse on kuitenkin GameView-luokan, eikä GameActivity-luokan toiminnasta. GameActivityn osalta testissä ollaan oikeastaan vain kiinnostuneita siitä, että aktiviteetin käynnistyessä pelin tilaa yritetään muuttaa RUNNING-tilaan.

Aktiviteetti on testissä eristettävä GameView:stä niin, että oikean näkymän sijaan se saa käyttöönsä jäljitellyn version näkymästä. Aktiviteetti käyttää findViewById()-metodia näkymän lataamiseen, koska se on alustettu layout-tiedostossa. Jotta tähän metodikutsuun pääsee väliin, täytyi aktiviteetille tehdä testiä varten aliluokka Mock-

Listing 6: Jäljittelijäaliluokka

```
private class MockGameActivity extends GameActivity {

        private GameView gameView;

  public MockGameActivity(GameView gameView) {
          setView(gameView);
  }

  @Override
  public View findViewById(int id) {
          return gameView;
  }

  public void setView(GameView gameView) {
    this.gameView = gameView
  }
}
```

Listing 7: Jäljittelyä käyttävä testi Robolectrcicillä

```
@Test
public void testActivityInitializesGameWithRunningStateWithMock() throw
        GameView gameView = mock(GameView.class);
        activity = new MockGameActivity(gameView);
        activity.onCreate(null);
        verify(gameView).setState(GameState.RUNNING);
}
```

GameActivity, joka on esitetty listauksessa 6. Aliluokka toteuttaa findViewById()-metodista version, joka palauttaa aina sille parametrina annetun näkymän, joka voi olla esimerkiksi jäljitelty näkymä. Tämä on yleinen tapa käyttää jäljittelijäkehyksiä riippuvuuksien eristämiseksi.

Tämän voisi Robolectricillä tehdä vaihtoehtoisesti siten, että toteuttaa oman Shadow-luokan Activity-luokasta, joka on kaikkien aktiviteettien yliluokka. Toteutuksen voi tehdä siten, että käyttää Robolectricin oletustoteutusta kaikkeen muuhun paitsi findViewById()-metodin toteutukseen [roba]. Jäljittelijänäkymän injektointia varten tälle voisi tehdä myös oman set-metodin, jolla jäljitelty näkymä sijoitettaisiin palautettavaksi. Yllä esitetty testattavan luokan aliluokka on kuitenkin helpompi tapa toteuttaa sama asia, koska Robolectriciä ei tarvitse erikseen käskeä käyttämään shadow-luokkana itse tehtyä toteutusta. Lisäksi aliluokkatoteutuksella on helpompi vaihdella, käytetäänkö testeissä oman aliluokan ilmentymää, vai varsinaisen testattavan luokan ilmentymää.

Listauksessa 7 on esitetty edellisen listauksen aliluokkaa käyttävä Robolectric-testi. Ensimmäisellä rivillä luodaan Mockiton jäljittelijäolio GameView-luokasta. Toisella rivillä luodaan testattavan aktiviteetin aliluokka, jolle syötetään jäljitelty näkymä konstruktoriparametrina. Sitten kutsutaan onCreate-metodia, kuten listauksen 2 vastaavassa testissä. Testin läpäisy testataan Mockiton verify-metodilla, joka varmistaa, että parametrina annettua jäljittelijäolion annettua metodia kutsuttiin annetulla parametrillä.

AndroidUnitTestin avulla tehty vastaava testi listauksessa 8 on jäljittelijänäkymän luomisen ja testin läpäisyn varmistamisen kannalta täsmälleen samanlainen kuin Robolectric-testi. Testissä käytetään apuna vastaavaa aliluokkaa kuin Robolectric-testissäkin, mutta näkymä pitää syöttää aliluokalle setView()-metodissa, koska Androi-

Listing 8: Jäljittelyä käyttävä testi AndroidUnitTestillä

```
@Test
public void testActivityInitializesGameWithRunningStateWithMock() throw
        GameView gameView = mock(GameView.class);
        MockGameActivity.setView(gameView);
        startActivity(
          new Intent(getInstrumentation().getTargetContext(),
                     MockGameActivity.class), null, null);
        verify(gameView).setState(GameState.RUNNING);
}
```

dUnitTestille annetaan testattavan luokan nimi jo luokan määrittelyssä ja aktiviteetti käynnistetään yliluokan avulla jo ennen testeihin pääsyä.

5.7 Testisyklin nopeus

Robolectricin vahvuudeksi mainitaan toistuvasti sen testien ajonopeus. Testien ajonopeudella on merkitystä kahdesta syystä: ensinnäkin laajojen projektien tapauksessa testitapauksia voi olla hyvin paljon ja kaikkien testien ajo voi kestää hyvin pitkään, mikä hidastaa koodin jakamista tai ohjelmistotuotantoprosessia, kun muutosten regressiotestaus aiemmin tehtyjen yksikkötestien avulla kestää pitkään. Toiseksi kehitettäessä sovellusta esimerkiksi Mobile-D-prosessin mukaisesti testilähtöisesti osaa testejä ajetaan jatkuvasti. Prosessi pysähtyy aina testien ajoajaksi, joten jos yksittäisten testien ajaminen on kovin hidasta, ei testilähtöinen kehittäminen ole järkevää.

	Keskiarvo (s)	Max (s)	Min (s)
Robolectric	1,59	1,61	1,577
AndroidUnitTest	44,10	44,271	43,868

Taulukko 1: Testikestot

Testasin ensin isomman testisetin ajonopeutta. Kopioin aiemmin luvussa esitellyn testiluokan mock-testin kanssa 32 erilliseksi testiluokaksi niin, että testimetodeita kertyi yhteensä 128. Ajoin kaikki testit yhtenä sarjana ja annoin Eclipsen ottaa aikaa testien suorituksesta. Tähän aikaan ei sisälly sitä aikaa, joka kuluu JUnitin käyn-

nistymiseen, vain itse testien suoritusaika. Toistin testit viisi kertaa ja testikestojen keskiarvo, maksimi ja minimi on esitetty taulukossa 1. Testit ajoin Macbook Pro:lla OS X versiolla 10.7.5, joka oli varustettu 2.7Ghz Intel core i7 -tuplaydinprosessorilla ja 8 gigatavun muistilla.

Robolectricin testit kestivät keskimäärin 1,59 sekuntia, AndroidUnitTestilla 44,10 sekuntia. Testien ajaminen emulaattorissa oli siis noin 27 kertaa hitaampaa kuin Robolectricilla JVM:llä. Koska testattava sovellus ja itse testit ovat hyvin yksinkertaisia, on aikaero todennäköisesti vielä suurempi laajempaa sovellusta testattaessa. Robolectricin lupaus nopeammista yksikkötesteistä vaikuttaa siis toteutuvan.

Testisarjan keston lisäksi tutkin yksittäisten testien kestoja. Erityisen kauan emulaattorissa ajetuista testeistä kesti ensimmäisen testiluokan mock-testi. Sen ajo kesti jokaisella ajokerralla yli 12 sekuntia, eli yli 25% koko testisarjan kestosta. Tämä johtuu siitä, että Mockito muokkaa ohjelman tavukoodia toimintaansa varten ja sen ensimmäinen alustus on hyvin hidas. Sama hitaus oli havaittavissa myös Robolectric-testeissä: ensimmäinen mock-testi kesti noin 0.3 sekuntia, mikä on noin 19% koko Robolectric-testisarjasta. Suhteellinen hidastuminen on siis verrattavissa AndroidUnitTestillä ajettuihin testeihin.

Robolectricillä myös kaikkein ensimmäinen testi on suhteessa hyvin hidas, noin 0.5 sekuntia. Robolectric toimii tavukooditason muokkauksessa Mockiton tavoin instrumentoidessaan testattavaa sovellusta toimimaan Robolectricin varjototeutusten kanssa, joten ensimmäisen testin suhteellinen hitaus johtunee samasta syystä kuin Mockitolla. Nämä seikat eivät kuitenkaan muuta kokonaiskuvaa siitä, että testien ajaminen on todella paljon nopeampaa Robolectricillä kuin AndroidUnitTestillä.

	Ensimmäisen testin alkuun (s)	Koko testisarjan loppuun
Robolectric	2.7	4.3
AndroidUnitTest	36	80

Taulukko 2: Testisarjan kesto alustus mukaanlukien

Toiseksi testasin, kuinka kauan kestää testikehyksen alustus siihen pisteeseen, että ensimmäinen testi lähtee ajautumaan koodimuutoksen jälkeen. Tämä tarkoittaa Android-tapauksessa sovelluksen asentamista emulaattoriin ja testikehyksen alustusta. Tulokset on esitetty taulukossa 2 Alustusajat olivat merkittäviä. Robolectricillä testikehyksen alustus kestää jopa kauemmin kuin koko 128 testin sarjan ajo ja AndroidUnitTestilläkin lähes yhtä kauan. Yhdenkin testin ajaminen AndroidUnitTestillä kestää yli 30 sekuntia.

5.8 Analyysi

Ohjelmakoodin yksikkötestaus onnistui hyvin sekä Androidin mukana tulevalla yksikkötestikehyksellä, että Robolectricillä. Itse testikoodi ei poikennut kovin merkittävästi toisistaan eri kehyksille kirjoitetulla koodilla ja testien kirjoitukseen ei tarvinnut kovin paljoa Android-spesifiä osaamista. Robolectric-koodi oli jopa yksinkertaisempaa testattavien komponenttien alustuksen ostalta, koska konstruktoreja saattoi käyttää suoraan yliluokan tarjoamien alustusmetodien sijaan. Toisaalta joskus oli vaikea tietää, mitä metodeita eri luokkien valmiit varjototeutukset tarjoavat. Näissä tapauksissa AndroidUnitTestCasen yliluokkametodit olivat käytössä selkeämpiä. Toisaalta Robolectric tarjoaa mahdollisuuden kirjoittaa itse omia varjoluokkia, joissa voi toteuttaa tynkiä Androidin kirjastoluokkien toiminnalle, kuten itse haluaa.

Jäljittelijäkehyksen käyttö onnistui ongelmitta sekä Robolectricillä että AndroidUnitTestCasella. Mockito toimi suoraan yhdessä Robolectricin kanssa ja Android-emulaattorissa ajaminenkin vaati vain erillisen Dalvik-käännöspaketin.

Suurin ero testityökalujen välillä oli testien suoritusnopeudella. Robolectric lupaa nopeita testejä ja toteuttaa lupauksensa; Robolectric-testit ajautuivat yli 25 kertaa nopeammin kuin emulaattorissa ajetut AndroidUnitTestit. Lisäksi AndroidUnitTestillä aika ensimmäiseen testitulokseen pienelläkin testiohjelmalla oli yli puoli minuuttia, joten testilähtöinen ohjelmointi AndroidUnitTestCasea käyttäen on käytännössä toivottoman hidasta.

Tässä luvussa testattu sovellus oli hyvin yksinkertainen, joten jotkin tulokset eivät välttämättä skaalaudu suoraan suurempien sovellusten testaamiseen. Toisaalta yksikkötestauksessa pyritään yleensä riippuvuuksien rajaamiseen ja mahdollisimman pienten osien testaamista erikseen, joten periaatteessa suurempien sovellusten yksikkötestaaminen ei ole juuri vaikeampaa kuin yksinkertaisten. Koodin laadun suhteen yksikkötestaus tosin on herkkää: tarkoitukseni oli tehdä yksikkötestaus luvussa 6.4 esiteltyä sovellusta vasten kuten toiminnallinen testauskin, mutta koska sovellusta ei ollut mitenkään tehty yksikkötestaus mielessä, osoittautui tämä mahdottomaksi.

6 Toiminnallisen testauksen työkalujen vertailua

Toiminnallisia testejä voi Androidilla tehdä monella eri työkalulla ja eri abstraktion tasolla. Androidin omista työkaluista toiminnallisten testien kirjoittamiseen soveltuvat AndroidInstrumentationTestCase-yliluokan avulla tehdyt JUnit-testit, Monkeyrunnerilla kirjoitetut testit sekä Uiautomator-testit. Näistä AndroidInstrumentationTestCase-luokkaa käyttävät testit ovat matalimmalla tasolla ja niiden kirjoittaminen vaatii tietoa ohjelmakoodin toiminnasta. Monkeyrunner-testit kirjoitetaan Pythonilla, eivät vaadi tietoa ohjelman sisäisestä rakenteesta, mutta koska Monkeyrunnerista puuttuvat kunnon assertointimahdollisuudet, en käsittele sitä tässä. Käsittelen sen sijaan Uiautomatorilla kirjoitettuja toiminnallisia testejä. Tämän lisäksi käsittelen Robotiumia, joka on Javalla käytettävä testityökalu sekä Troydia, joka käyttää Rubya testien tuottamiseen.

6.1 Robotium

Robotium on suosittu testityökalu Android-sovellusten testauksessa. Robotium pyrkii tarjoamaan Android-sovelluksille vastaavan toiminnallisuuden kuin suosittu Selenium-testikehys selaintestaukseen. Selenium on toiminnallisessa ja integraatiotestauksessa käytetty työkalu, joka mahdollistaa selaimen toimintojen automatisoimisen, kuten linkkien klikkauksen ja lomakekenttien täyttämisen koneellisesti [sel].

Robotium on tarkoitettu Android-sovellusten toiminnalliseen, järjestelmä- ja hyväksyntätestaukseen. Se on musta laatikko -työkalu, eli testin kirjoittajan ei tarvitse päästä käsiksi tai tuntea testattavan sovelluksen koodia. Robotium-testit voivat testata samassa testitapauksessa useita aktiviteetteja. Robotium-testeissä annetaan ohjeita, missä järjestyksessä käyttöliittymäelementtejä klikataan tai syötetään tekstiä.

Robotiumtestejä voi ajaa niin emulaattorissa kuin puhelimessakin. Testit eivät kuitenkaan voi käsitellä kahta eri sovellusta, eli yksi testitapaus voi käsitellä vain yhtä sovellusta. Sovellustenvälinen integraatiotestaus ei ole mahdollista.

Robotiumin kotisivuilla sille esitellään useita vahvuuksia Android SDK:n mukana tuleviin työkaluihin verrattuna. Testit vaativat vain vähäistä tuntemusta testattavasta sovelluksesta, Robotium tukee usean aktiviteetin testaamiseta samassa testissä, testien kirjoittamisen nopeus, testikoodin selkeys ja sitkeys, joka johtuu ajoaikaisesta sidonnasta käyttöliittymäkomponentteihin, nopea suoritusnopeus ja helppo

integrointi jatkuvan integroinnin työkaluihin Antin tai Mavenin avulla [robb].

6.2 Troyd

Troyd on Robotiumia käyttäen tehty integraatiotestaustyökalu, jonka tavoite on yhdistää Monkeyrunnerin skriptausominaisuudet ja Robotiumin tarjoama korkean tason API. Troyd-testit käyttävät korkean tason komentoja, kuten p̈aina nappia nimeltä ẍtai ẗarkista, että ruudulla näkyy teksti y;̈ joten testien kirjoituksen pitäisi olla nopeaa. Lisäksi Troyd tarjoaa nauhoitus-toiminnon, jolla testiä voidaan kirjoittaa siten, että testiä kirjoittaessa ohjelma etenee aina seuraavaan tilaan testin mukaisesti. Lopuksi testi tallentuu testitapauksiksi [JF12].

Troyd-testejä kirjoitetaan Rubylla käyttäen Rubyn Test::Unit-työkalua, joka on Rubyn vakiokirjaston mukana tuleva yksikkötestityökalu. [tes] Troydin komennot sisältävä TroydCommands-moduli sisällytetään testiluokkaan käyttämällä Rubyn mixin-toiminnallisuutta. Testitapauksia voi kirjoittaa kuten tavallisia Test::Unit-testejä tai sitten voi käyttää nauhoitusmahdollisuutta.

Troydin heikkouksia on sen tekijöiden mielestä mahdollisuus testata vain yhtä sovellusta kerrallaan. Esimerkiksi, jos sovellus aukaisee selainikkunan, Troyd menettää sovelluksen kontrollin. Tämä johtuu Androidin testi-instrumentaation rajoituksista. Toinen Troydin heikkous on hidas suoritusnopeus, koska testiskripti odottaa jokaisen komennon jälkeen, että sovellus on varmasti oikeassa tilassa ennen testin jatkamista. [JF12]

Troydin lähdekoodi on avoin ja se löytyy GitHubista [tro].

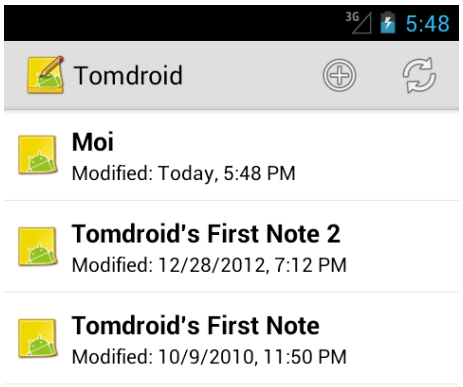
6.3 Aiempaa tutkimusta

Jeon & Foster mainitsevat Robotiumin vahvuudeksi Androidin omaa Instrumentatiota rikkaamman APIn. Esimerkiksi nappien painamiseen voidaan käyttää nappien nimeä, josta Robotium laskee napin sijainnin. He myös vertaavat Robotiumia omaan Troyd-työkaluunsa ja sanovat sen heikkoudeksi, että testit pitää määritellä etukäteen, eikä niitä pysty muokkaamaan ajonaikaisesti. Muulta toiminnallisuudeltaan Troyd ja Robotium ovat suunnilleen samankaltaisia, koska Troyd on tehty Robotiumin päälle [JF12].

Benli et al. tutkivat valkoinen laatikko ja musta laatikko -testaustapojen suhteellista tehokkuutta Android-alustalla. Musta laatikko -testeissä tutkimuksessa käytettiin

Robotiumia, koska Androidin mukana tulevat testaustyökalut eivät mahdollistaneet järkevää JUnit-pohjaista musta laatikko -testausta. Valkoinen laatikko -testit tehtiin Androidin yksikkötestityökaluilla. Tuloksena oli, että valkoinen laatikko -testien kirjoittaminen kesti 89% kauemmin, mutta testien ajaminen oli 43% nopeampaa kuin musta laatikko -testien. Testiohjelmaan istutetut bugit löytyivät valkoinen laatikko -testeillä, mutta ei Robotium-testeillä. Testiajojen nopeuteen liittyen on huomattava, että Robotium-testit ajettiin visuaalisessa moodissa niin, että jokaisen komennon välissä oli yksi sekunti, jotta käyttöliittymän tila ehdittiin havaita manuaalisesti [BHH+12].

6.4 Testiprojektista

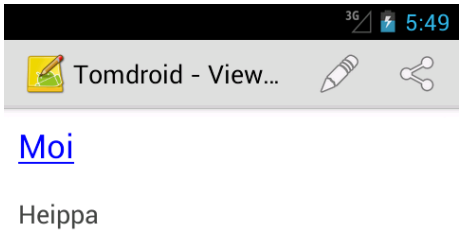


Kuva 10: Muistikirjalista Tomdroidissa

Toiminnallisten testityökalujen vertailussa käytän testattavana ohjelmana Tomdroidia. Tomdroid on GPL-lisenssillä julkaistu avoimen lähdekoodin muistikirjasovellus [tomb]. Tomdroid on valittu testattavaksi sovellukseksi, koska sen lähdekoodi on saatavilla, se on riittävän monimutkainen, jotta sille tehdyt testit kuvaisivat oikeassa Android-kehityksessä kohdattavia testaushaasteita, se on vasta beta-vaiheessa, joten

sovelluksesta pitäisi löytyä myös bugeja ja lisäksi sovelluksen oma testaus on lähes olematonta.

Tätä tutkimusta varten tein kopion tomdroidin lähdekoodista versiosta 0.7.2 ja kopioin sen githubiin. Sieltä löytyy myös kaikki sovellukselle tekemäni testit [toma].



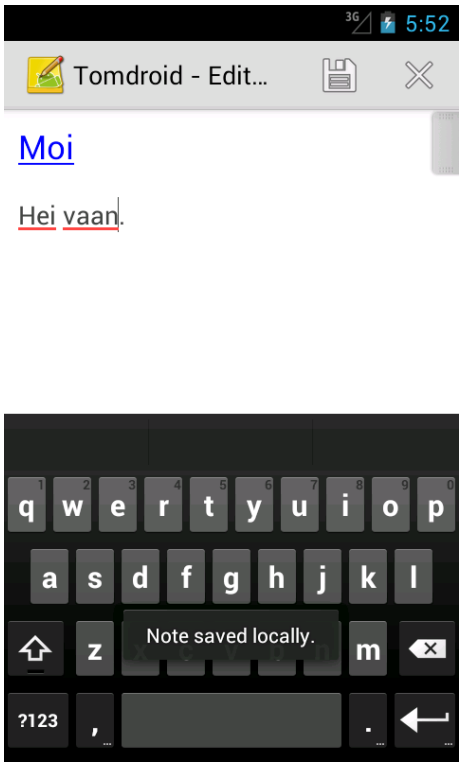
Kuva 11: Muistikirjan luku Tomdroidissa

Tomdroidissa olennaisimmat näkymät ovat muistikirjalista, josta on ruutukaappaus kuvassa 10, yksittäisen muistikirjan selaaminen (kuvassa 11) ja sen editointi (kuvassa 12).

Listanäkymässä näkyvät kaikki käyttäjän muistikirjat päivitysajan mukaan järjestettynä. Muistikirjan koskeminen avaa kyseisen muistikirjan selausnäkymän. Yläpalkin +-symboli luo uuden muistikirjan ja avaa sen muokkausnäkymään. Yläpalkin oikeassa reunassa on synkronointi-symboli, josta muistikirjojen tila päivitetään palvelimen kanssa.

Selausnäkymässä voi lukea yksittäistä muistikirjaa. Jos tekstiä on enemmän kuin ruudulle mahtuu kerrallaan, sitä voi selata raahaamalla. Kynä-ikoni yläpalkissa avaa muistikirjan editointinäkymään. Yläpalkin oikean reunan ikonista voi jakaa muistikirjan toisiin sovelluksiin. Lisäksi vasemman yläreunan ikonista pääsee takaisin listaan.

Editointinäkymässä yläpalkissa on kuvakkeet muutosten tallentamista ja perumista varten. Tallennettaessa ruudulla näkyy hetken aikaa leijuke, jossa kerrotaan muutosten tallennuksesta. Painettaessa peru-nappia ruudulle tulee dialogi, jossa pitää vahvistaa peruminen edelliseen tallennettuun versioon. Lisäksi vasemman yläreunan ikonista pääsee takaisin listaan.



Kuva 12: Muistikirjan editointi Tomdroidissa

6.5 Asennukset

Robotiumin asennus on yksinkertaista. Projektin build pathiin tarvitsee vain lisätä Robotiumin jar-paketti, jossa tulee kaikki tarvittava mukana. Itse Robotium-testit perivät Androidin omasta ActivityInstrumentationTestCase2-yliluokasta. Listauksessa 9 on esitetty Robotium-testin runko ilman varsinaisia testejä. setUp()-metodissa alustetaan Solo, joka on Robotiumin testit suorittava olio. Se ottaa konstruktoriparametreina ActivityInstrumentationTestCase2:n tarjoaman instrumentaation ja testattavan aktiviteetin. tearDown()-metodissa kutsutaan finishOpenedActivities()-metodia, joka lopettaa kaikki testin aikana aktiivisena olleet aktiviteetit.

Robotium-testeissä on huomattava, että jos sovellus muuttaa muistikortille tai muualle tallennettua tilaansa, on testeissä manuaalisesti pidettävä huolta, että sovellus

Listing 9: Robotium-testirunko

```
public class RobotiumTest
        extends ActivityInstrumentationTestCase2<Tomdroid> {

        private Solo solo;

        public RobotiumTest() {
                super(Tomdroid.class);
        }

        @Override
        public void setUp() {
                solo = new Solo(getInstrumentation(),
                                getActivity());
        }

        @Override
        public void tearDown() throws Exception {
                solo.finishOpenedActivities();
        }
}
```

Listing 10: Muistikirjojen poisto

```
private void removeAllNotes() {
        NoteManager.deleteAllNotes(getActivity());
}
```

Listing 11: Troyd-nauhoitusskriptin ja testien ajaminen

```
ruby bin/rec.rb apks/org.tomdroid.apk
ruby bin/trun.rb
```

Listing 12: Uiautomator-testien ajaminen

```
#!/bin/sh
ant build
adb push bin/TomdroidUiAutomatorTest.jar /data/local/tmp/
adb shell uiautomator runtest TomdroidUiAutomatorTest.jar
    -c org.tomdroid.test.UiAutomatorTest
```

resetoidaan takaisin alkuperäiseen tilaan, jotta testejä voi toistaa useita kertoja luotettavasti. Tomdroidin tapauksessa tämä tehtäisiin kutsumalla tearDown():ssa apumetodia removeAllNotes(), joka on esitetty listauksessa 10. Metodi kutsuu Tomdroidin NoteManagerin deleteAllNotes()-metodia, joka poistaa kaikki sovelluksen tallentamat muistikirjat.

Troyd vaatii toimiakseen Rubyn version 1.8.7 sekä nokogiri-xml-kirjaston. Lisäksi se tukee vain Androidin versiota 2.3.6, joten uudemman Android-version vaativien sovellusten testaaminen ei ole sillä mahdollista. Troydin asennus kesti huomattavasti Robotiumin asennusta kauemmin, vaikka koneellani oli valmiiksi asennettuna Rubyn versionhallintatyökalu rvm [rvm], jolla oikean ruby-version asennus onnistuu yhdellä komennolla. Rubyn versio 1.8.7 on julkaistu kesällä 2008 [rub], ja on siten varsin vanha Rubyn versio, joten joissain järjestelmissä näin vanhan version asennus voi olla haasteellista. Asennusten jälkeen Troydin testin kirjoittaminen onnistui kuitenkin nopeasti nauhoitusskriptin avulla, joka hoiti kaiken muun kuin itse testikoodin kirjoittamisen. Nauhoitus ja ajoskriptit on esitetty listauksessa 11. rec.rb on nauhoitusskripti, jolla testit kirjoitetaan ja run.rb ajoskripti, joka ajaa oletuksena kaikki testcases-hakemistosta löytyvät testit. Testit voivat olla myös eri sovelluksille, jolloin Troyd asentaa kunkin testattavan sovelluksen erikseen.

Uiautomator vaatii vähintään Androidin API-version 16, jonka mukana tulee uiautomator.jar-paketti, joka täytyy asettaa testiprojektin riippuvuuksiin Androidin ja JUnitin ohella. Lisäksi testien ajaminen vaatii tietokoneessa kiinni olevan Android-laitteen, johon testattava sovellus (tai sovellukset) on asennettuna. Testien kääntäminen ja

Listing 13: Muistikirjan luontitesti Robotiumilla

```
public void testCreateNoteAddsNote() {
        solo.assertCurrentActivity(
            "Testi alkoi väärästä aktiviteetista",
            Tomdroid.class);
        assertFalse(solo.searchText("new note"));
        solo.clickOnActionBarItem(R.id.menuNew);
        solo.assertCurrentActivity(
            "Uuden muistikirjan luonti ei avannut editointinäkymää",
            EditNote.class);
        solo.enterText(0, "new note");
        solo.clickOnActionBarItem(R.id.edit_note_save);
        solo.clickOnActionBarHomeButton();
        solo.assertCurrentActivity(
            "Koti-näppäimen painaminen ei vienyt takaisin muistikirjali
            Tomdroid.class);
        assertTrue(solo.searchText("new note"));
}
```

ajaminen on esitetty listauksessa 12. Testit käännetään Apache Antilla [ant], joka on avoimen lähdekoodin käännöstyökalu Java-sovelluksille. Toisella rivillä käytetään Androidin mukana tulevaa adb-työkalua (Android debug bridge) käännetyn testisovelluksen siirtämiseen puhelimeen ja kolmannella rivillä ajetaan testisovelluksesta luokka org.tomdroid.test.UiAutomatorTest.

6.6 Robotium-testit

Robotium-testi, jossa testataan uuden muistikirjan luonti, on esitetty listauksessa 13. Robotiumilla testiä ohjataan Solo-luokan instanssin kautta, jossa on sovelluksen kanssa kommunikointiin tarkoitettuja metodeja, sovelluksen tilasta kertovia metodeja, sekä assertteja. Testin ensimmäisellä rivillä käytetään assertCurrentActivity()-metodia asserttia varmistamaan, että testi alkaa muistikirjalistasta. Toisella rivillä varmistetaan, että testissä luotavaa muistikirjaa ei vielä löydy listasta. Ilman tätä testissä ei voisi olla varma, että muistikirja on luotu onnistuneesti juuri testin aikana. Seuraavalla rivillä painetaan yläpalkin uuden muistikirjan luovaa nappia

Listing 14: Muistikirjan luontitesti Troydilla

```
def test_"create␣note␣adds␣note"
  click "OK"
  assert_not_text "new␣note"
  clickImg 1
  edit 0, "new␣note"
  clickImg 1
  clickImg 0
  assert_text "new␣note"
end
```

clickOnActionBarItem()-metodilla. Se ottaa parametrina komponentin id:n, johon ollaan painamassa. Tämän jälkeen pitäisi avautua uusi muistikirja editointinäkymään, mikä varmistetaan seuraavalla rivillä. Sitten syötetään enterText()-metodilla uuden muistikirjan otsikoksi "new note". Ensimmäinen parametri kertoo, monenteenko ruudulla näkyvään tekstinmuokkauskomponenttiin teksti syötetään. Tämän jälkeen klikataan yläpalkin tallennus-nappia ja sitten muistikirjalistaukseen vievää nappia. Lopuksi vielä varmistetaan, että palattiin takaisin muistikirjalistaan ja listasta löytyy nyt juuri luotu aktiviteetti.

6.7 Troyd-testit

Troyd-testi, jossa luodaan muistikirja, on esitetty listauksessa 14. Troyd-testit asentavat aina ensimmäisenä uuteen emulaattoriin koko sovelluksen, joten Robotium-testissä tarvittuja vanhojen muistikirjojen poistoa ei tarvita. Tein testin Troydin nauhoitusskriptin avulla, joten itse testi on automaattisesti kirjoitettu tämän pohjalta. Testitiedosto sisältää testimetodin lisäksi metodit sovelluksen alustamiselle ja lopettamiselle sekä testeissä käytetyt Ruby-metodit, mutta nämä kaikki Troyd tuotti automaattisesti nauhoitusskriptin avulla. Itse testimetodin sisältö on sama kuin testiskriptissä kirjoittamani komennot. Lisäksi näin sovelluksen testiä vastaavassa tilassa emulaattorissa testiä kirjoittaessa, mikä helpotti testin kirjoittamista. Listauksessa on lisäksi huomautettava, että Rubyssa metodien sulut ovat vapaaehtoiset, mikäli sekaantumisen vaaraa ei ole, siksi testissä metodien parametrit eivät ole sulkujen sisällä. Kun nauhoitusskriptissä on saanut haluamansa testitapauksen valmiiksi, se tallennetaan sofar-metodilla, jonka parametrina on testin nimi, kuten

Listing 15: Testin tallennus nauhoitusskriptistä Troydilla

```
sofar "create note adds note"
finish
```

on tehty listauksessa 15. Finish-komento lopettaa nauhoitusskriptin.

Testin ensimmäisellä rivillä painetaan OK-nappia, koska sovellus näyttää ensimmäisellä käynnistyskerralla ohje-tekstin. Troydilla ei voi varmistaa, missä aktiviteetissa ollaan, kuten Robotium-testissä tehtiin, joskin listan tähän mennessä vierailluista aktiviteeteista saisi getActivities-metodilla. Troyd ei myöskään tue Robotium-testissä käytettyä R-luokan id:n perusteella elementtien etsimistä, vaan elementit etsitään indeksin perusteella järjestyksessä ruudulta. Sen takia testi luettuna ei ole yhtä selkeä, kuin Robotium-testit. Tässä testissä ensimmäinen clickImg-metodi painaa uuden muistikirjan luovaa nappia, toinen clickImg tallennus-nappia ja kolmas clickImg palaa takaisin listaukseen -nappia.

Testien kirjoittaminen sujui Troydilla nopeasti, mutta toisaalta osaan Rubya yhtä hyvin kuin Javaakin, joten erilainen syntaksi ei häirinnyt kirjoitusta. Nauhoitusskripti on kuitenkin vielä hieman raakile, esimerkiksi ruudulta löytymättömän indeksin klikkaaminen kaatoi sovelluksen ja pakotti aloittamaan nauhoituksen alusta.

6.8 Uiautomator-testit

Listauksessa 16 on esitetty Uiautomator -testi, jossa luodaan uusi muistikirja. Testi perii yliluokan UiAutomatorTestCase. Testeissä ajetaan JUnit3-tyylin mukaisesti test-alkuiset metodit. Testimetodi heittää UiObjectNotFoundExceptionin, jos yritetään tehdä interaktiota komponentin kanssa, jota ei lyödetty selectorilla. Testissä käytetään pääosin kahta UiAutomatorin mekaniikkaa. UiSelectorin eri metodeilla etsitään käyttöliittymäelementtejä, joiden avulla konstruoidaan UiObject-olioita, joiden kanssa sitten kommunikoidaan tai kysytään niiden tilaa.

Testissä käytetään kolmea eri UiSelectorin hakumetodia: text()-metodi hakee käyttöliittymäelementtiä, johon liittyy parametrina annettu näkyvä teksti, description()-metodi käyttää hakemiseen elementille liitettyä selitettä, esimerkiksi resurssi-tiedoston kautta annettua. className()-metodi etsii kaikki elementit, jotka ovat annetun luokan ilmentymiä, ja palauttaa niistä ensimmäisen.

Listing 16: Muistikirjan luontitesti Uiautomatorilla

```
public class UiAutomatorTest extends UiAutomatorTestCase {
  public void testCreateNoteAddsNote()
              throws UiObjectNotFoundException {
    getUiDevice().pressHome();
    UiSelector selector = new UiSelector().text("Tomdroid")
    UiObject targetApp = new UiObject(selector);
    targetApp.clickAndWaitForNewWindow();
    selector = new UiSelector().text("new␣note")
    UiObject newNoteText = new UiObject(selector);
    assertFalse(newNoteText.exists());
    selector = new UiSelector().description("New")
    UiObject newNoteButton = new UiObject(selector);
    newNoteButton.clickAndWaitForNewWindow();
    selector =
        new UiSelector().className("android.widget.EditText")
    UiObject titleEditText = new UiObject(selector);
    titleEditText.setText("new␣note");
    selector = new UiSelector().description("Save")
    UiObject saveButton = new UiObject(selector);
    saveButton.click();
    selector =
        new UiSelector().description("Siirry␣etusivulle")
    UiObject homeButton = new UiObject(selector);
    homeButton.clickAndWaitForNewWindow();
    selector = new UiSelector().text("new␣note")
    newNoteText = new UiObject(selector);
    assertTrue(newNoteText.exists());
    removeCreatedNote(newNoteText);
  }
}
```

Listing 17: Luodun muistikirjan poisto Uiautomatorilla

```
private void removeCreatedNote(UiObject noteText)
            throws UiObjectNotFoundException {
  noteText.clickAndWaitForNewWindow();
  UiSelector selector =
      new UiSelector().description("Lisää asetuksia")
  UiObject moreOptionsButton = new UiObject(selector);
  moreOptionsButton.click();
  selector = new UiSelector().text("Delete")
  UiObject deleteButton = new UiObject(selector);
  deleteButton.click();
  selector = new UiSelector().text("Yes")
  UiObject confirmButton = new UiObject(selector);
  confirmButton.clickAndWaitForNewWindow();
}
```

UiObjectin metodeista testeissä käytetään seuraavia: clickAndWaitForNewWindow() painaa käyttöliittymäelenettiä ja odottaa seuraavan aktiviteetin latautumista, exists() palauttaa true, jos elementti löytyy ruudulta, setText()-kirjoittaa elementtiin parametrina annetun tekstin. click()-metodi painaa elementtiä, mutta ei jää odottamaan uutta aktiviteettia.

Ui automator -testit alkavat siitä tilasta, missä puhelin on testin käynnistyessä, joten on hyvä tapa aloittaa kaikki testit painamalla kotinäppäintä, jolla puhelin palautuu aloitusruutuun. Testissä oletin, että Tomdroid löytyy aloitusnäytöltä, joten seuraavaksi testissä etsitään ruudulta elementtiä, johon liittyy Tomdroid-teksti, eli sovelluksen käynnistysikoni. Yleisempi ratkaisu olisi avata lista kaikista sovelluksista ja etsiä testisovellus sitä kautta.

Kun sovellus on avattu, testi etenee kuten Robotium-testissäkin. Ensin varmistetaan, että "new note-nimistä muistikirjaa ei löydy valmiiksi listalta, sitten painetaan uusi muistikirja -nappia. Seuraavaksi etsitään ensimmäinen EditText-elementti ruudulta, ja syötetään siihen muistikirjan nimi. Sitten tallennetaan muistikirja, palataan muistikirjalistaan ja varmistetaan, että muistikirja tallentui.

Lopuksi testissä kutsutaan removeCreatedNote()-metodia, joka on esitetty listauksessa 17. Metodi poistaa juuri luodun muistikirjan. Koska Uiautomator-testit eivät

pääse käsiksi ohjelmakoodiin kuten Robotium-testit, joudutaan tehdyt muutokset perumaan käyttöliittymän kautta. Poistometodi saa parametrinaan valmiiksi viitteen luotuun muistikirjaan etusivulla, joten, sitä ei tarvitse hakea uudestaan.

6.9 Testien suoritusnopeudet

Ui automator: 15.980, 15.883, 16.320, 16.506, 16.240

Troyd: 2:22.776, 2m24.027s, 2m25.961s, 2m23.822s, 2m22.304s

	Keskiarvo (s)	Max (s)	Min (s)
Robotium	?	?	?
Troyd	?	?	?
UI Automator	16,186	16,506	15,883

Taulukko 3: Testikestot

Eri testityökalujen testien ajoajat on esitetty taulukossa 3. Ajoin kaikilla työkaluilla testisarjan viidesti ja mittasin testiaikojen keskiarvon ja hitaimman ja nopeimman testiajon.

Troyd-testit ajoin emulaattorissa trun.rb-skriptillä ja ajastin unixin time-työkalulla. Testiajoa hidastaa se, että joka ajokerralla emulaattori käynnistetään ja testattava sovellus asennetaan emulaattoriin. Toisaalta tästä syystä testien kirjoittaminen on helpompaa, koska sovellusta ei tarvitse palauttaa testejä edeltävään tilaan kuten Robotiumilla ja Ui automatorilla ajaessa. (Tsekkaa myös, olisiko nopeampaa (olisi) ajaa troyd-testit suoraan puhelimessa)

Uiautomator-testit ajoin listauksessa 12 esitetyllä shell-skriptillä ja ajoitin testit unixin time-työkalulla. Testiajon kestoon sisältyy testien kääntäminen, siirtäminen puhelimeen ja itse testiajo. Uiautomator-testin kestoon sisältyy myös luodun muistikirjan poisto käyttöliittymästä. Testeissä oletetaan, että sovellus on valmiiksi asennettuna koneeseen kytkettyyn puhelimeen.

7 Yhteenveto

Android-sovellusten testaamiseen on kehitetty runsaasti testaustyökaluja. Jo Androidin mukana tulevat työkalut tarjoavat varsin kattavan työkaluvalikoiman Android-sovellusten testaamiseen ohjelmistotuotantoprosessin eri vaiheissa. Tämän lisäksi kolmannen osapuolen kehittämät testaustyökalut, kuten Robolectric ja Robotium, täydentävät Googlen kehittämiä testaustyökaluja.

Lähteet

AE12 Allevato, A. ja Edwards, S. H., Robolift: engaging cs2 students with testable, automatically evaluated android applications. *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 2012, sivut 547–552.

AHH+04 Abrahamsson, P., Hanhineva, A., Hulkko, H., Ihme, T., Jäälinoja, J., Korkala, M., Koskela, J., Kyllönen, P. ja Salo, O., Mobile-d: An agile approach for mobile application development. *OOPSLA '04 Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004, sivut 174–175.

anda *Android 4.1 Compatibility Definition.* URL `http://static.googleusercontent.com/external\_content/untrusted\_dlcp/source.android.com/en//compatibility/4.1/android-4.1-cdd.pdf`.

andb Android developer documentation. URL `http://developer.android.com/`.

ant Apache ant. URL `http://ant.apache.org/`.

Bec03 Beck, K., *Test-driven development: by example.* Addison-Wesley, 2003.

BHH+12 Benli, S., Habash, A., Herrmann, A., Loftis, T. ja Simmonds, D., A comparative evaluation of unit testing techniques on a mobile platform. *Proceedings of the Ninth International Conference on Information Technology - New Generations*, 2012, sivut 263–268.

JF12 Jeon, J. ja Foster, J. S., Troyd: Integration testing for android. Tekninen raportti, University of Maryland, August 2012. URL `http://drum.lib.umd.edu/handle/1903/12880`.

mav Apache mavenin kotisivut. URL `http://maven.apache.org/`.

MBS02 Michael, J. B., Bossuyt, B. J. ja Snyder, B. B., Metrics for measuring the effectiveness of software-testing tools. *Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002, sivut 117–128.

MFC01 Mackinnon, T., Freeman, S. ja Craig, P., Endo-testing: unit testing with mock objects. Teoksessa *Extreme Programming Examined*, Addison-Wesley, 2001, sivut 287–301.

moc Mockiton kotisivut. URL `https://code.google.com/p/mockito/`.

PS92 Poston, R. ja Sexton, M., Evaluating and selecting testing tools. *IEEE Software*.

PY08 Pezzè, M. ja Young, M., *Software Testing And Analysis*. John Wiley Sons, 2008.

roba Robolectricin kotisivut. URL `http://pivotal.github.com/robolectric/`.

robb Robotiumin kotisivut. URL `http://code.google.com/p/robotium/`.

rub Ruby 1.8.7 has been released. URL `http://www.ruby-lang.org/en/news/2008/05/31/ruby-1-8-7-has-been-released/`.

rvm Rvm:n kotisivut. URL `https://rvm.io`.

sam Robolectric-testiprojekti. URL `https://github.com/jmschultz/Eclipse-Robolectric-Example`.

sel Seleniumin kotisivut. URL `http://seleniumhq.org/`.

SLS09 Spillner, A., Linz, T. ja Schaefer, H., *Software Testing Foundations*. Rocky Nook, 2009.

S+11 Sadeh, B., Ørbekk, K., Eide, M. M., Gjerde, N. C., Tønnesland, T. A. ja Gopalakrishnan, S., Towards unit testing of user interface code for android mobile applications. Teoksessa *Software Engineering and Computer Systems*, Zain, J. M., Maseri, W., Mohd, W. ja El-Qawasmeh, E., toimittajat, Springer, 2011, sivut 163–175.

tes Test::unitin kotisivut. URL `http://test-unit.rubyforge.org/`.

toma Tomdroidille tehdyt testit. URL `https://github.com/jniemisto/tomdroid`.

tomb Tomdroidin kotisivut. URL `https://code.launchpad.net/tomdroid`.

tro Troydin lähdekoodi. URL `https://github.com/plum-umd/troyd`.

wik Android (operating system). URL `http://en.wikipedia.org/wiki/Android\_(operating\_system)`.

ZG11 Zechner, M. ja Green, R., *Beginning Android 4 Games Development*. 2011.