

Android-sovellusten testaaminen

Juho Niemistö

Helsinki 17.11.2013

Pro gradu -tutkielma

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

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	17.11.2013	65 sivua + 10 liitesivua	
Tiivistelmä — Referat — Abstract			
<p>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 miljoona.</p> <p>Sovellusten laatu on erityisen tärkeää Android-alustalla, jossa kilpailua on runsaasti ja sovellusten hinta niin alhainen, ettei se muodosta estettä sovelluksen vaihtamiselle 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.</p> <p>Tässä tutkielmassa tutustutaan Android-sovellusten rakenteeseen, niiden testaamiseen ja Android-alustalla toimiviin automaattisen testauksen työkaluihin. Erityisesti keskitytään yksikkö- ja toiminnallisen testauksen työkaluihin. Yksikkötestityökaluista vertaillaan Androidin omaa yksikkötestikehystä Robolectriciin. Toiminnallisen testauksen työkaluista vertaillaan Uiautomatoria, Robotiumia ja Troydia.</p> <p>ACM Computing Classification System (CCS):</p> <p>-Software and its engineering~Software testing and debugging</p> <p>-Software and its engineering~Operating systems</p>			
Avainsanat — Nyckelord — Keywords			
Android, testaus, yksikkötestaus, toiminnallinen testaus, mobiilisovellukset			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

Sisältö

1	Johdanto	1
2	Android-sovellusten rakenne ja kehityssykli	3
2.1	Historia	3
2.2	Androidin kehitystyökalut	3
2.3	Android-sovellusten rakenne	4
2.4	Aktiviteetit	7
2.5	Palvelut	10
2.6	Sisällöntarjoajat	11
2.7	Aikeet	12
2.8	Android manifest	13
2.9	Android-sovelluksen julkaiseminen	14
3	Ohjelmistojen testaaminen ja mobiilisovellusten testaamisen erityispiirteitä	15
3.1	Testaamisen peruskäsitteitä	15
3.2	Testaaminen osana ohjelmistotuotantoprosessia	16
3.3	Mobiilisovellusten testaamisen erityispiirteitä	18
3.4	Testityökalujen arviointikriteereistä	19
4	Android-sovellusten testaaminen ja Androidin mukana tulevat testityökalut	21
4.1	Android-testien ajoympäristö	21
4.2	Komponenttikohtaiset yksikkötestiluokat	22
4.3	MonkeyRunner	23
4.4	Uiautomator ja Uiautomatorviewer	24
4.5	Monkey	25
5	Yksikkötestityökalujen vertailua	27

5.1	Robolectric	27
5.2	Aiempaa tutkimusta	28
5.3	Testiprojekti	29
5.4	Robolectricin asentaminen	30
5.5	Perusominaisuudet	31
5.6	Toiminta jäljittelijäkehysten kanssa	36
5.7	Testisyklin nopeus	39
5.8	Yksikkötestauksen haasteita	40
5.9	Analyysi	43
6	Toiminnallisen testauksen työkalujen vertailua	44
6.1	Robotium	44
6.2	Troyd	45
6.3	Aiempaa tutkimusta	45
6.4	Testiprojekti	46
6.5	Testitapaus	48
6.6	Asennukset	49
6.7	Robotium-testit	52
6.8	Troyd-testit	53
6.9	Uiautomator-testit	54
6.10	Testien suoritusnopeudet	57
6.11	Analyysi	57
7	Yhteenveto	59
	Lähteet	62
	Liitteet	

1 Yksikkötesteissä testatun sovelluksen lähdekoodi

1 Johdanto

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

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

Sovellusten laatu on erityisen tärkeää Android-alustalla, jossa kilpailua on runsaasti ja sovellusten hinta useimmiten niin alhainen, ettei se muodosta estettä sovelluksen vaihtamiselle toiseen. Sovelluskauppa on myös aina saatavilla suoraan laitteesta. Nämä kilpailutekijät asettavat sovellusten testaamisellekin haasteita. Toisaalta sovellukset tulisi saada nopeasti sovelluskauppaan, mutta myös sovellusten laadun pitäisi olla hyvä.

Sovellusten laatua voidaan parantaa automaattisilla testaustyökaluilla. Testien automatisointi tarkoittaa, että kerran kirjoitettua testisarjaa voidaan ajaa helposti uudelleen. Tämä mahdollistaa esimerkiksi testivetoisen kehityksen yksikkötestityökalujen avulla ja regressiotestauksen jatkuvan integraation työkaluissa. Jotta testaus ei hidastaisi liikaa sovelluksen ohjelmointia, testaustyökalujen pitäisi olla helppokäyttöisiä ja tehokkaita. Androidille onkin kehitetty monia 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 tutkielmassa vain Androidille Javalla kehitettyjä natiivisovelluksia. Androidille voi kehittää sovelluksia myös muun muassa 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 käytettävyydestäukseen liittyvät prosessit ja työkalut on rajattu tämän työn ulkopuolelle.

Keskityn 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 ja pyritään varmistamaan haluttu toiminnallisuus kokonaisuudessaan sovelluksessa. Nämä työkalut olen valinnut huomion kohteeksi, koska toiminnallinen ja yksikkötestaus ovat yleisimmät testauksen muodot mobiilisovelluksia kehitettäessä ja työkaluvalikoima on myös monipuolisin. Toiminnallinen ja yksikkötestaus ovat myös useimmiten automatisoituja testaamisen vaiheita.

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 ja Android-sovellusten julkaisun.

Kolmannessa luvussa käsitellään ohjelmistojen testaamisen perusasioita, testausta osana ohjelmistotuotantoprosessia sekä mobiilisovellusten testaamisen erityispiirteitä. Lisäksi pohditaan testaustyökalujen arviointia ja laatukriteerejä. Neljännessä luvussa käsitellään Android-sovellusten testauksen perusteita sekä esitellään Androidin kehitystyökalupaketin mukana tulevia testityökaluja, kuten Monkeyrunner, Monkey ja UiAutomator.

Viidennessä luvussa syvennyttään yksikkötestityökaluihin. Vertailen Androidin yksikkötestikehystä avoimen lähdekoodin Robolectriciin, joka pyrkii tarjoamaan Androidin oman yksikkötestikehyksen ominaisuudet Javan omassa virtuaalikoneessa, jotta testien ajaminen olisi nopeampaa. Selvitän luvussa myös yksikkötestikehysten yhteensopivuutta jäljittelijäkehysten kanssa.

Kuudennessa luvussa käsitellään toiminnallisen testauksen työkaluja. Vertailen Androidin kehitystyökalujen mukana tulevan UiAutomator-työkalun lisäksi Roboium ja Troysd -testityökaluja. Kukin näistä työkaluista tarjoaa hieman erilaisen lähestymistavan ja abstraktiotason toiminnallisten testien kirjoittamiseen. Viimeisessä luvussa on yhteenveto.

2 Android-sovellusten rakenne ja kehityssykli

Tässä luvussa esitellään Androidin historiaa, kehitystyökaluja ja sovelluksen julkaisua 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 älypuhelisten 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 pääsääntöisesti 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 kaikki yhden sovelluksen asentamiseen tarvittavat tiedot. Android-sovelluksia ohjelmoidaan yleensä Eclipsellä, joka on suosittu ohjelmointiympäristö erityisesti Java-ohjelmistojen tekemiseen [ecl]. Android-sovellusten kehittämiseen tarvitsee käytännössä Javan kehitystyökaluista (SDK) version 5 tai uudemman, Androidin SDK:n, Eclipsen sekä Android-laajennoksen (Android Development Tools, ADT) Eclipselle [ZG11, 25].

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

Sovelluksen ja testien kehitysaikaista ajamista varten SDK:n mukana tulee Android-emulaattori. Emulaattoreita voi ajaa Androidin eri järjestelmäversioilla ja laitteisto-profiileilla, jotta on mahdollista testata sovelluksen toimivuutta erilaisissa Android-ympäristöissä. Emulaattori kykenee myös jossain määrin simuloimaan lisälaitteiden, kuten kiihtyvyysanturin, toimintaa. Suurin puute emulaattorissa on sen heikko suoritusnopeus. Sovelluksia ja testejä voi ajaa kehitystyökalusta myös suoraan tietokoneeseen liitetystä Android-laitteesta.

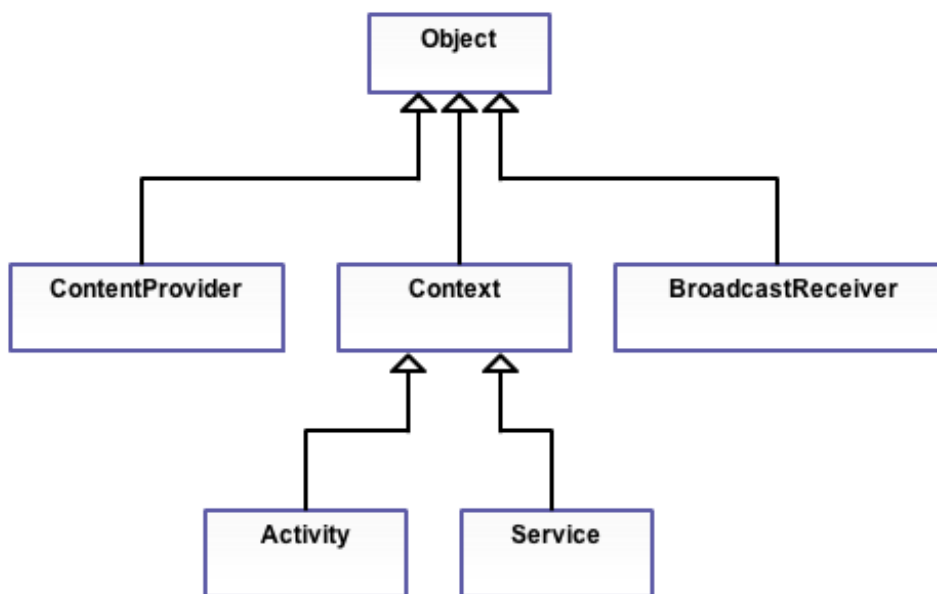
Androidin Eclipse-laajennos toimii siltana Android SDK:n ja Eclipsen välillä mahdollistaen SDK:n tarjoamien ominaisuuksien hyödyntämisen suoraan Eclipsestä käsin. ADT:n avulla on myös mahdollista seurata tietokoneeseen kytkettyjen Android-laitteiden tapahtumalogeja ja debug-tietoja [ZG11, 25-50].

Eclipsessä Android-projektin aloittaminen luo sovelluksen perusrakenteen valmiiksi, joten sovellusta pääsee suoraan ohjelmoimaan ilman konfigurointia. Itse sovelluksen ohjelmointi taas on pääosin kuin tekisi mitä tahansa Java-ohjelmaa. Ohjelmaa voi ajaa suoraan Eclipsestä emulaattorissa tai Android-laitteesta. Tällöin Eclipse kääntää ja paketoi sovelluksen ja asentaa sen kohdelaitteeseen. Testejä ajettaessa prosessi on sama: sovellus käännetään, paketoidaan ja asennetaan kohdelaitteeseen, jonka jälkeen testit ajetaan ja testien tulokset näkyvät Eclipsessä, kuten tavallisesakin Java-ohjelmassa. Etenkin testejä ajaessa asennussyklejä tulee siis jatkuvasti, mikä hidastaa testausta.

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. Esimerkiksi Amazon on julkaissut Kindle Fire -tabletteja oman Android-johdannaisen käyttöjärjestelmän päälle [kin].

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 vähemmän 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 (*activities*), palveluista (*services*), sisällöntarjoajista (*content providers*) sekä lähetyksen 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 ylikuokka *Context*, lähetyksen vastaanottajat ja sisällöntarjoajat perivät vain Javan *Object*-luokan. Kuvaa on yksinkertaistettu siten, että *Contextin* ja *Activityn* ja *Service*n välillä olevia *Wrapper*-luokkia on jätetty kuvaamatta perintahierarkiassa. *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 Intent-luokan aliluokkia.

Android-sovellukset käyttävät usein hyväkseen toisten sovellusten komponentteja. Sovellukset eivät pysty suoraan kutsumaan toisiaan. 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 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 suuntausta tai vastaanotettaessa puhelua, joten sovelluksen pitää pysyä tehokkaasti palautumaan keskeytyneeseen tilaansa. Sovelluksella on siis lukuisia mahdollisia käynnistymis- ja sulkeutumispolkuja. Tämän takia ohjelmakomponenttien elinkaaren hallinta on tärkeä osa sovelluksen rakentamista.

Android-sovelluksissa pyritään erottamaan sovelluksen Java-toteutus ja lokalisointi mahdollisimman hyvin toisistaan. Tämä tapahtuu resurssien (*resource*) avulla. Androidin resurssi-luokka tarjoaa sovellukselle pääsyn lokalisoituihin resursseihin siten, ettei Java-koodissa tarvitse ottaa kantaa lokalisointiin. Tärkein lokalisoitu resurssi on sovelluksen tekstit, mutta myös grafiikan ja layoutin voi määritellä kielen tai laitteen näytön koon perusteella. Itse resurssitekstit määritellään xml-

ti 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 kerrallaan voi olla aktiivinen - eli reagoida käyttäjän syötteisiin. Yksi sovelluksen aktiviteeteista 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 tyypillisesti 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. Näiden vaihdosten pitäisi lisäksi tapahtua mahdollisimman tehokkaasti, jotta järjestelmän toiminta olisi käyttäjän näkökulmasta 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.

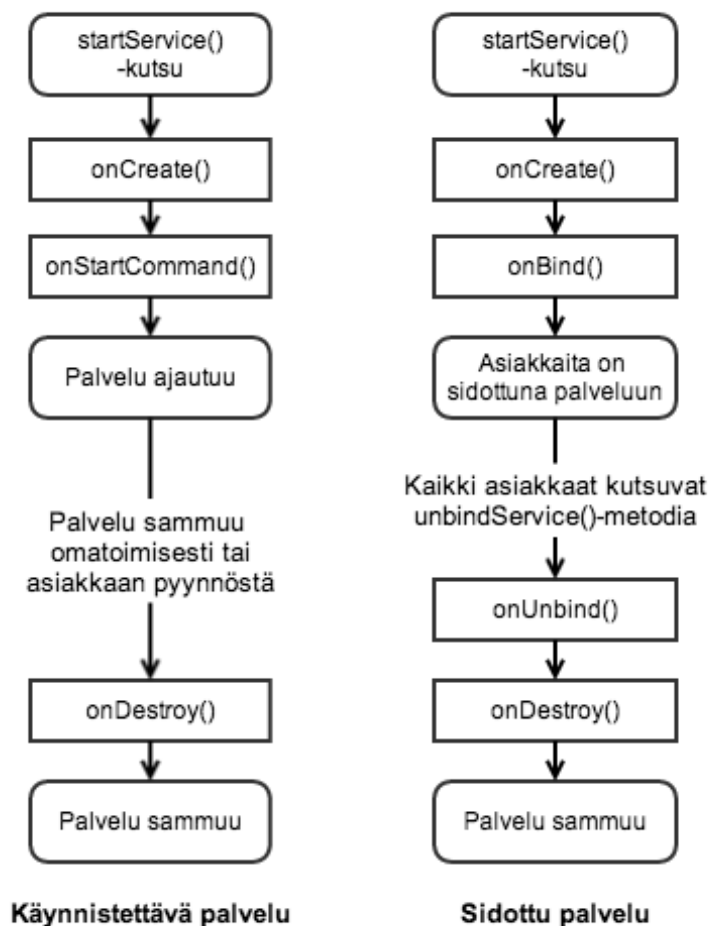
Aktiviteetin käyttäjälle näkyvä elinkaari on onStart()- ja onStop()-kutsujen välillä. onStart()-metodia kutsutaan, kun aktiviteetti tulee näkyväksi käyttäjälle, ja onStop()-metodia kutsutaan, kun jokin toinen aktiviteetti on peittänyt kyseisen aktiviteetin kokonaan. Näkyvän elinkaaren aikana tulisi ylläpitää niitä resursseja, joita tarvitaan käyttäjän kanssa kommunikointiin sekä sellaisia, jotka saattavat muuten

vaikuttaa käyttäjälle näkyvään käytöliittymään. Esimerkiksi lähetystenvastaanottajaa on hyvä kuunnella tällä välillä mahdollisten järjestelmänlaajusten 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 `onPause()`-, `onStop()`- tai `onDestroy()`-kutsun jälkeen. Tämän takia pysyväksi tarkoitettu tieto on tallennettava `onPause()`-kutsun jälkeen. Tallennus voidaan tehdä esimerkiksi toteuttamalla takaisinkutsuun `onSaveInstanceState()`, jota kutsutaan aina, ennen kuin järjestelmä mahdollistaa aktiviteetin tuhoamisen. `onSaveInstanceState()` 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 `onSaveInstanceState()`- 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 [andb].



Kuva 3: Palvelun elinkaari [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.

Palveluita on 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 palautusarvoa 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()`-metodia. Sidotut palvelut tarjoavat asiakas-palvelin-rajapinnan 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 sulkeutuu 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 pitkäkestoiisiin 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 sovelluksen säilömiseen tietoon ei tarvitse päästä käsiksi sovelluksen ulkopuolelta. Sovellustenväliseen kommunikointiin sisällöntarjoajat tarjoavat vakiorajapinnan, joka pitää huolen prosessienvälisestä kommunikoinnista ja tietoturvasuudesta.

Androidin mukana tulee valmiiksi toteutetut sisällöntarjoajat esimerkiksi musiikille, videotiedostoille ja käyttäjän yhteystiedoille. 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 (*intent*). 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 eksplisiittisesti `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, tietotyyppisiä 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

Jokaiseen 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, sisälöntarjoajat 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 vaatimukset ympäristöltään: mikä on sovelluksen vaatima Android API:n 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 API:n 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].



Kuva 4: Android-sovelluksen kehityskaari [andb]

2.9 Android-sovelluksen julkaiseminen

Kuvassa 4 on kuvattu yksinkertaistettu Android-sovelluksen kehityskaari. Kun sovellus on kehitetty ja testattu, on viimeisenä vaiheena sen julkaisu. Android-sovellukset julkaistaan yleensä Google Play -sovelluskaupassa [goo]. Sovelluksia voi julkaista kuka tahansa. Julkaistakseen sovelluksia julkaisijan täytyy rekisteröityä ja maksaa 25 dollarin rekisteröitymismaksu. Sovelluksen voi julkaista halutessaan myös vaikka omilla verkkosivuillaan tai muissa sovelluskaupoissa, mutta parhaan näkyvyyden Android-laitteissa saa tällä hetkellä Google Playssa.

Julkaisua varten sovellus täytyy konfiguroida julkaisua varten ja kääntää julkaistava APK-tiedosto, joka ladataan sovelluskauppaan. Lisäksi tarvitaan sovelluskauppaa varten muutama ruutukaappaus sovelluksesta sekä esittelyteksti. Sovellukselle pitää myös valita nimi. Sovellus voi olla joko ilmainen tai maksullinen ja siinä voi olla lisäksi sovelluksen sisäisiä maksumekanismeja. Ainut rajoitus on, että kerran ilmaisen julkaistua sovellusta ei saa muuttaa maksulliseksi. Ilmaiseksi ladattavissakin sovelluksissa voi kuitenkin käyttää sovelluksen sisäisiä maksumekanismeja.

Sovelluksen voi julkaista vain haluamissaan maissa ja saatavuutta voi rajoittaa myös laitekohtaisesti tai Androidin version perusteella. Google Play mahdollistaa myös alfa- ja betatestauksen pienille ryhmille. Kehittäjän hallintanäkymästä näkee tilastotietoa sovelluksen kaatumisista ja virhetilanteista sekä mitä versioita sovelluksesta on käytössä ja missä Androidin versioissa niitä ajetaan. Sovelluksesta voi tehdä milloin tahansa päivitysversion lataamalla päivitetyn APK-tiedoston sovelluskauppaan. Google Play ilmoittaa sovelluksen ladanneille automaattisesti uuden version ilmestymisestä [pub].

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 lasilaatikkotestausta (*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].

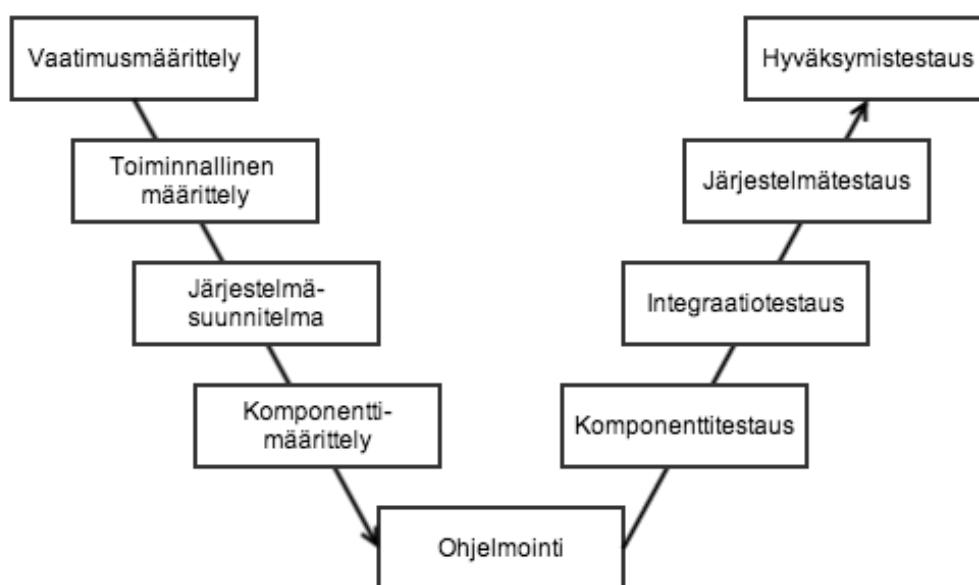
Toiminnallisessa testauksessa (*functional testing*) testattavan ohjelman sisäistä rakennetta ei tunneta, vaan ollaan kiinnostuneita vain syötteistä ja niitä vastaavista tulosteista. Toiminnallista testausta voidaan kutsua myös musta laatikko -testaukseksi (*black box testing*). Toiminnallisessa testauksessa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

Fuzz-testaus on satunnaistestauksen muoto, jossa ohjelmaan kohdistetaan satunnaisia syötteitä ja seurataan ohjelman tulosteita. Syötteitä voidaan myös luoda kieloppien avulla, jolloin voidaan käyttää hyödyksi tietoa testattavasta ohjelmasta. Fuzz-testaus on osoittautunut tehokkaaksi tavaksi löytää sovelluksista virheitä, joita muut testaustavat eivät paljasta. Toisaalta satunnaisuus tarkoittaa, että testejä pitää suorittaa lukemattomia kertoja, jos fuzz-testaukselle halutaan kattava testi-peitto. Fuzz-testaus on havaittu erityisen tehokkaaksi turvallisuusaukkoja etsittäessä [GLM⁺08].

Jäljittely (*mocking*) on tekniikka, joka helpottaa yksikkötestien kirjoittamista. Yk-

sikkötestien ulkoiset riippuvuudet voidaan korvata testeissä kontrolloitavilla jäljitte-
lijäolioilla. 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 vaikeita saada muokattua. Lisäksi jäljittele-
mällä voidaan korvata vielä toteuttamattomat ulkoiset riippuvuudet jäljittelijäto-
teutuksilla [MFC01].

3.2 Testaaminen osana ohjelmistotuotantoprosessia



Kuva 5: Testauksen v-malli [SLS09, 40]

Ohjelmistotuotantoprosessia lähestytään yleensä jonkin elinkaarimallin kautta. Testauksen näkökulmaa edustaa testauksen v-malli, joka on esitetty kuvassa 5. 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 ovat testausvaiheet, joissa jokaisessa varmistetaan, että ohjelma täyttää vasemman haaran samalla ta-

solla 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 ajatella 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 automaattisilla yksikkötestikehyksillä, kuten Javan JUnitilla. Oikean toiminnallisuuden testaamisen lisäksi on tärkeää testata myös toiminta väärillä syötteillä ja poikkeustilanteissa. Moderni tapa tehdä komponenttitestauksia 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 kuuluvat 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 toimitussopimuksen ja sen käyttäjät hyväksyvät järjestelmän. Hy-

väksyntätestaus voi sisältää myös alfa tai beta -testauksen oikeilla järjestelmän lopukäyttäjillä [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].

Automaattiset testityökalut ovat tyypillisesti sitä enemmän käytössä, mitä alemmalla v-mallin testaustasolla testausta tehdään. Varsinkin komponenttitestaus on tyypillisesti täysin automatisoitua. Testauksen automatisoinnilla pyritään testausprosessin tehostamiseen, kun testien suorittaminen on nopeaa. Lisäksi testien luotavuus paranee, kun epäluotettavat manuaaliset vaiheet korvataan automaattisilla [SLS09, 201].

3.3 Mobiilisovellusten testaamisen erityispiirteitä

Mobiilisovellusten kehittämiseen liittyy haasteita, jotka liittyvät mobiiliympäristön rajoituksiin. Näitä ovat muun muassa päätelaitteiden rajallinen kapasiteetti ja jatkuva kehittyminen, erilaiset standardit, protokollat ja verkkotekniikat, päätelaitteiden monimuotoisuus, mobiililaitteiden käyttäjien erikoistarpeet sekä tiukat aikavaatimukset sovellusten saamiseksi markkinoille. Sovellusmarkkinat ovat myös globaalit, joten sovellusten lokalisoinnissa eri kieli- ja kulttuuriympäristöihin aiheuttaa haasteita.

Sovelluksia rajoittaa myös mobiililaitteiden pieni fyysinen koko ja laitteiden erot koossa, painossa ja näytön koossa. 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 kirjoitta-

minen 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, refaktori. 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 luettavaa ja eleganttia koodi on. Vaiheen nimi on vihreä, koska useimmissa yksikkötestityökaluissa lopputuloksena näkyy vihreä palkki, kun testit menevät läpi. Viimeisessä vaiheessa refaktoroidaan toisessa vaiheessa mahdollisesti syntynyt duplikaatti- tai muuten vaikealukuinen 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 muun muassa Poston ja Sexton [PS92]. Hyvän testaustyökalun kriteerit ovat osin kontekstista riippuvia, mutta Poston ja Sexton määrittelevät myös yleisempiä kriteerejä testityökalujen arviointiin. Usein kriteerit ovat myös helposti määriteltävissä: jollain työkalulla pystyy testaamaan asioita, joita toisella ei voi. Ei-toiminnallisista olennaisista ominaisuuksista 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. Metriikoita on muunmuassa työkalun kypsyys ja käyttäjäkunnan koko, helppokäyttöisyys, kustomointimahdollisuudet, automaattinen testitapausten generointi, muiden ohjelmointityökalujen tarjoama tuki työkalulle, luotettavuus sekä suoritusnopeus. Osalle metriikoista esitetään myös täsmällisiä laskukaavoja, jotka helpottavat kvantitatiivisen analyysin tekemistä.

Spillner et al. [SLS09, 218] listaavat testityökalujen valintaan vaikuttavia tekijöitä:

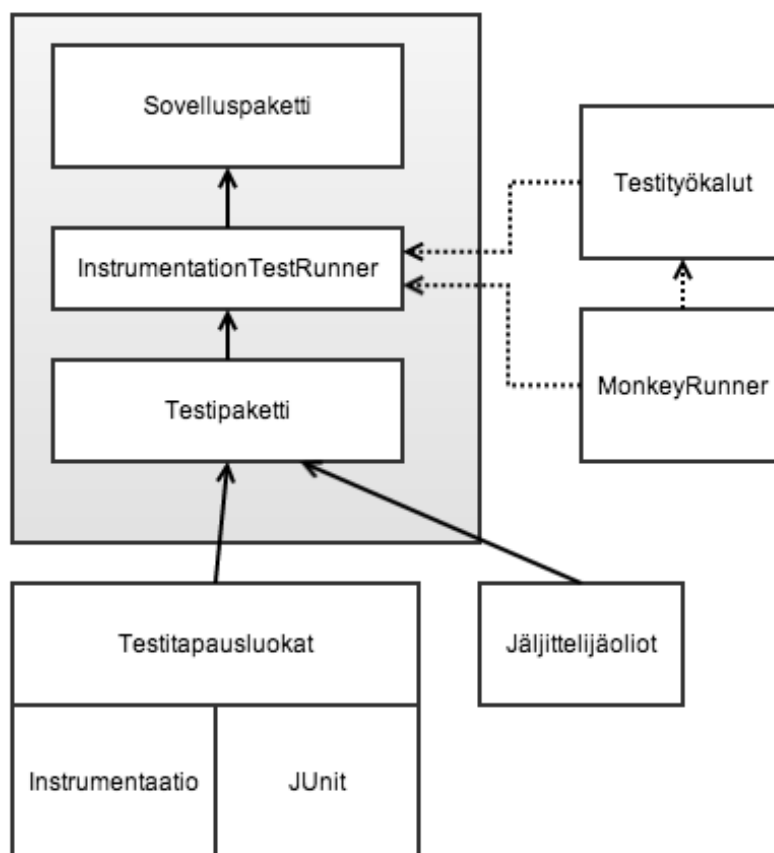
- Miten hyvin testityökalu saa tietoa ja pystyy vaikuttamaan
- Testaajien tietotaito työkalun käytöstä
- Miten hyvin työkalu voidaan integroida käytössä oleviin kehitystyökaluihin
- Miten hyvin työkalu voidaan integroida muihin käytettäviin testaustyökaluihin
- Millä alustalla työkalu toimii
- Työkalun valmistajan tarjoama tuki, luotettavuus ja markkina-asema
- Lisenssiehdot, hinta ja ylläpitokustannukset

Osa kriteereistä on hyvin kontekstiriippuvaisia, mutta myös niitä voi soveltaa yleisemmän arvioinnin tekemisessä. Esimerkiksi testaajien tietotaidon sijaan voidaan arvioida, miten helppoa työkalun käyttö on oppia.

4 Android-sovellusten testaaminen ja Androidin mukana tulevat testityökalut

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

4.1 Android-testien ajoympäristö



Kuva 6: Android-testien ajoympäristö [andc]

Android-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. Nä-

mä 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 6 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 takaisinkutsumetodikutsuja pääsee muokkaamaan, sekä jäljittelijäolioita korvaamaan järjestelmän oikeita luokkia testin ajaksi jäljittelytoteutuksella [andb].

4.2 Komponenttikohtaiset yksikkötestiluokat

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

Aktiviteettien testauksessa Androidin JUnit-laajennus on merkittävä, koska aktiviteeteilla on monimutkainen elinkaari, joka perustuu paljolti takaisinkutsumetodeihin. Näiden suora kutsuminen ei ole mahdollista ilman Androidin omia testiluokkia. Aktiviteettien testauksen päätyyliluokka 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 näppäimistötapahdumien 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 kehitystyökalussa ja tarvittaessa Android-kirjaston jäljittelyyn 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 toimi-

vat 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 ylliluokka on `ProviderTestCase2`, joka tarjoaa käyttöön jäljittelijäoliot `ContentResolver`istä ja `Context`istä, jolloin sisällöntarjoajia voi testaja eristyksissä muusta sovelluksesta. Ylliluokka tarjoaa myös metodit sovelluksen oikeuksien testaamisen. `Context`in 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 [andc].

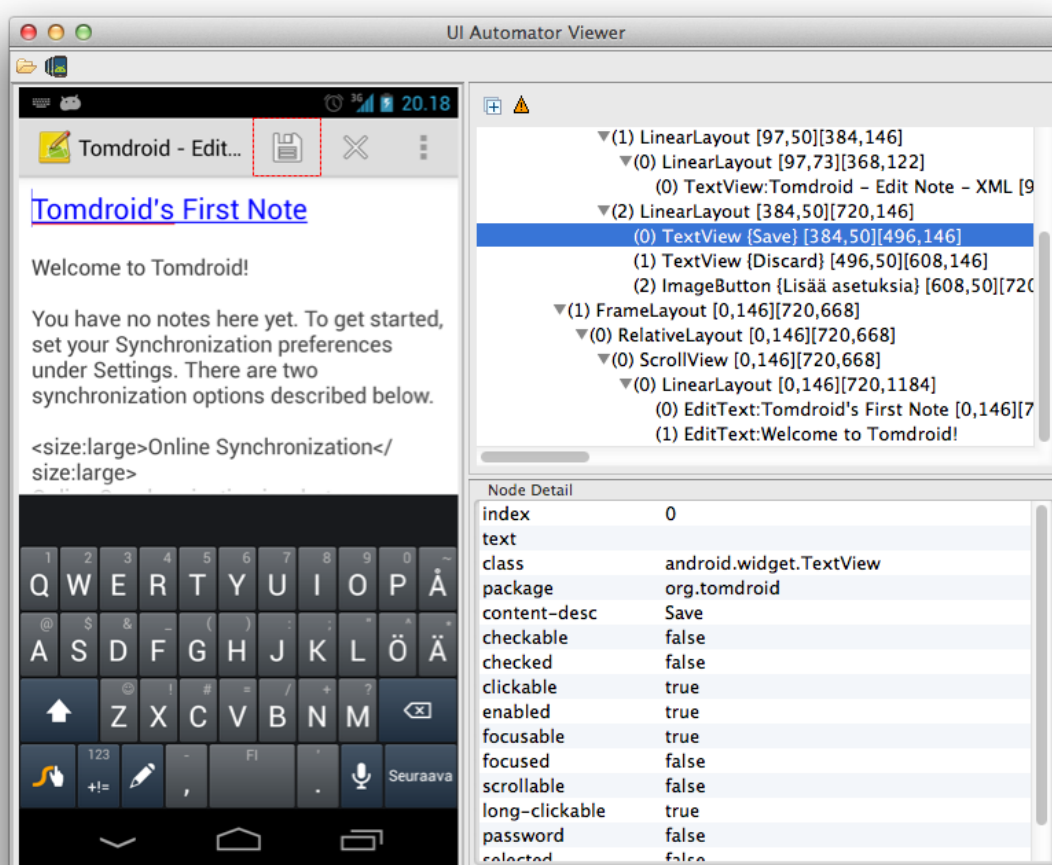
4.3 MonkeyRunner

MonkeyRunner [monb] on Androidin sovelluskehiksen mukana tuleva työkalu, joka tarjoaa rajapinnan, jolla Android-sovellusta voi ohjata laitteessa tai emulaattoris-
sa. 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 referenssikuvuihin. Tällä tavalla voidaan tehdä esimerkiksi regressiotestausta.

MonkeyRunnerilla voidaan testata yhtä aikaa monia eri emulaattoreita tai useita laitteita, jolloin voidaan testata nopeasti sovelluksen toimivuutta eri Androidin versioilla ja laitteistoprofiileilla. 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.

4.4 Uiautomator ja Uiautomatorviewer

Uiautomator [uia] tarjoaa mahdollisuuden toiminallisten musta laatikko -testien kirjoittamiseen Javalla. Uiautomator on Monkeyrunneria monipuolisempi, koska se tarjoaa laajemman rajapinnan sovelluksen tilatietojen kyselyyn ja sitä kautta mahdollisuuden monipuolisempien testitapausten tekemiseen. Siinä on kaksi komponenttia: Uiautomatorviewer ja Uiautomator. Uiautomatorviewer on graafinen työkalu, jolla voi analysoida Android-sovelluksen käyttöliittymää. Uiautomator taas on työkalu, joka tarjoaa rajapinnan ja moottorin toiminnallisten musta laatikko -testien ajamiseen.



Kuva 7: Ruutukaappaus Tomdroidin muokkausnäymästä Uiautomatorilla analysoituina

Uiautomatorviewer (kuvassa 7) analysoi sovelluksen näkymän komponentit. Vasemmassa osassa näytetään sovelluksesta ruutukaappaus, josta voi valita käyttöliittymä-

mäkomponentteja analysoitavaksi. Valittu komponentti näytetään punaisella viivalla ympyröitynä (kuvassa valittuna tallennus-painike yläpalkissa). Oikea puoli jakautuu kahteen osaan. Yläpuolella näytetään valitun komponentin sijainti näkymän komponenttihierarkiassa, elementin tyyppi ja selite, sekä sen pikselikoordinaatti näytöllä. Alapuolella taas on tarkemmat tiedot komponentista: sen järjestysluku hierarkiatasollaan (*index*), komponenttiin mahdollisesti liittyvä teksti (*text*), komponentin luokkahierarkia (*class*), paketti (*package*), siihen liitetty komponenttiselite (*content-desc*) sekä komponentin interaktioon liittyviä tietoja, kuten onko se painettavissa tai valittuna. Näitä tietoja voi käyttää hyväkseen Uiautomator-testeissä esimerkiksi komponentin valintaan.

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, joka tulee uusimpien Androidin sovelluskehysten mukana. Uiautomator-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.

4.5 Monkey

Monkey [mona] on Androidin sovelluskehysten mukana tuleva työkalu, jota voi ajaa emulaattorissa tai Android-laitteessa ja joka tuottaa sovellukselle pseudosatunnaisia syötteitä. Näitä voivat 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 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 .

5 Yksikkötestityökalujen vertailua

Androidin yksikkötestaustapa on tehdä JUnit-testejä Androidin oman `AndroidUnitTestCase`-luokan aliluokkana. Nämä testit ajetaan emulaattorissa Dalvik-ympäristössä. Tässä tavassa on kaksi heikkoutta, jonka takia on kehitetty myös kolmansien osapuolien yksikkötestityökaluja Android-ympäristöön. Ensinnäkin testien ajaminen emulaattorissa on hitaampaa, kuin jos niitä voisi ajaa suoraan tavallisessa 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 `AndroidUnitCasella` 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.

Android-sovellusten yksikkötestauksessa kiinnostavaa on nimenomaan Androidin kirjastoluokista perivien keskeisten komponenttien testaus, koska sovelluksessa mahdollisesti olevat muut kuin Androidin kirjastoluokista perivät luokat on helppo testata tavanomaisilla Javan yksikkötestityökaluilla ilman erityisesti Androidille tarkoitettuja työkaluja samaan tapaan kuin muutkin Java-sovellukset.

Yksikkötestaustyökaluissa tärkeintä on nopeus, jotta tdd-kehityssykli toimisi mahdollisimman nopeasti, työkalun käytön helppous ja koodin ytimekkyys, jotta testien kirjoittaminen sujuu nopeasti, sekä yhteensopivuus jäljittelijäkehysten kanssa, jotta komponenttien yksikkötestaus muusta sovelluksesta eristettynä olisi mahdollista.

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 paluuarvon. `Robolectricin` varjoluokkien käytön vaihtoehtona on jonkin jäljittelijäkehysten käyt-

täminen Androidin kirjaston korvaamiseen, mutta tämä tapa kirjoittaa testejä on hyvin työläs, koska koodirivejä kertyy väistämättä paljon. Lisäksi tällöin testejä kirjoitettaessa täytyy tuntea testattavan metodin toiminta hyvin tarkasti, jotta jäljittelijätoteutukset saadaan kirjoitettua. Robolectricin varjoluokat mahdollistavat enemmän musta laatikko -tyyppisen testauksen [roba].

5.2 Aiempaa tutkimusta

Sadeh et al. vertasivat Androidin aktiviteettien yksikkötestausta JUnitilla, Androidin omilla yksikkötestityökaluilla sekä Robolectricilla [S⁺11]. 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 testien suoritusnopeus. Robolectric-testit ajautuivat viisi kertaa nopeammin kuin Androidin työkaluilla ajatut testit, koska Androidin työkaluilla kirjoitetut testit ajetaan Dalvik-emulaattorilla, Robolectric taas suoraan Javalla. Androidin omien työkalujen suurin vahvuus oli testien kirjoittamisen helppous.

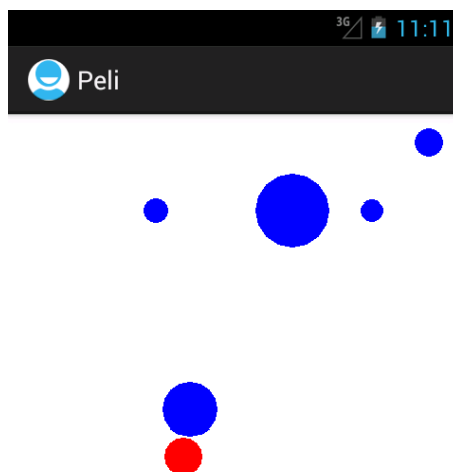
Sadeh et al. eivät käyttäneet JUnit-testeissään mitään jäljittelijätyökalua, joten testeissä testiluokan riippuvuudet jäljiteltiin tekemällä niistä staattisia sisäluokkia testiluokan sisään. Tämä vaatii paljon koodia ja vaikutti osaltaan siihen, miksi puhdas JUnit-testi näytti niin hankalalta. Androidin 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 [JF12] mainitsevat Robolectricin vahvuudeksi sen, että se pyörii Javan virtuaalikoneessa ja näin ohittaa testeistä hitaan vaiheen, jossa 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.

Allevato & Edwards [AE12] käyttivät Robolectriciä opetuskäyttöön tarkoitetun RoboLIFT-työkalun kehitykseen. Robolectric auttoi heitä ohittamaan emulaattorin käytön ja nopeuttamaan opiskelijoiden testisykliä ja automaattista arviointial-

goritmia. Tässä käytössä Robolectricin ongelma oli, että se ohittaa käyttöliittymän piirtämisen kokonaan ja muunmuassa näkymien `onDraw()`-metodia ei kutsuta ollenkaan. Tämän seurauksena esimerkiksi näkymän leveys on aina 0 pikseliä, jolloin sellaiset testit, joilla haluttiin klikata näytöllä johonkin suhteelliseen kohtaan (vaikkapa keskelle) eivät toimineet oikein.

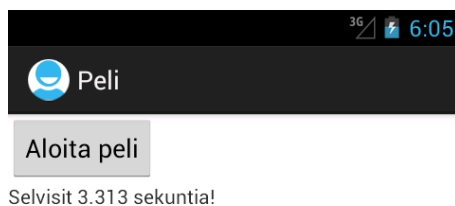
5.3 Testiprojekti



Kuva 8: Ruutukaappaus testiohjelman pelinäkömästä

Yksikkötestaustyökaluja testasin itse tehdyllä demoprojektilla. Kyseessä on yksinkertainen peli, jonka pelinäkömästä on ruutukaappaus kuvassa 8. Pelissä ohjataan kosketusnäytöllä painamalla punaista palloa ja pyritään väistämään ympäriinsä pomppivia sinisiä palloja. Kun peli päättyy, palataan takaisin päänäkömään, josta on ruutukaappaus kuvassa 9. Tästä näkömästä voi aloittaa uuden pelin ja lisäksi näkee, montako sekuntia edellinen peli kesti.

Peli koostuu kahdesta aktiviteetista, yksinkertaisemmasta `MainActivity`sta sekä hieman monimutkaisemmasta `GameActivity`sta, 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



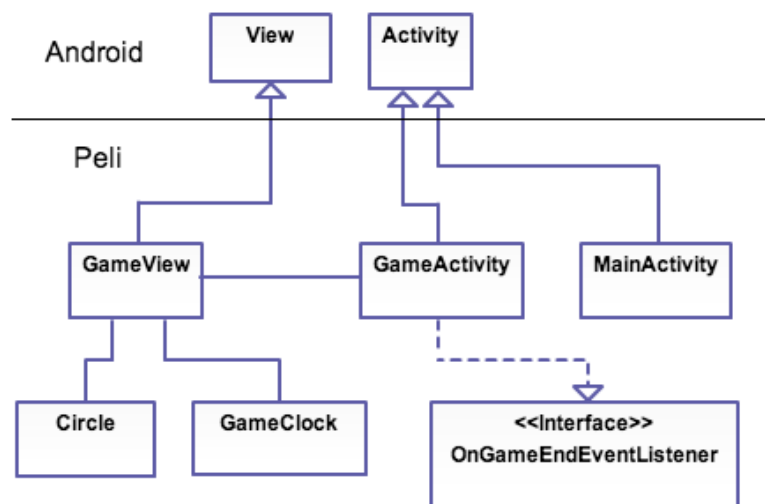
Kuva 9: Testiohjelman päänäkö

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 10.

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 lai-



Kuva 10: Testiohjelman luokkakaavio

Listaus 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"
                )))
    }
}

```

tetaan viittaamaan Android-projektin koodiin. Testiprojektin käännöspolkuun 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.

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

Listaus 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);
        assertEquals("gameView.getState()", GameState.RUNNING);
    }

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

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 `AndroidUnitTest`issä, 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 tunnisteiden perusteella. Tämän jälkeen varmistetaan, että `GameView`in tila on todella vaihtunut `Running`-tilaan.

Toinen testi on rakenteeltaan hyvin samanlainen: siinä testataan, että `onPause()`-elinkaarimetodin kutsuminen siirtää `GameView`in `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 Androidin omalla yksikkötestikirjastolla vastaavat testit kuin listauksessa 2. Itse testit ovat täysin identtisiä Robolectric-testien välillä, erot ovat alustuksessa. Androidin aktiviteettiyksikkötestit perivät ylliluokan `ActivityUnitTestCase`, jolle annetaan konstruktoriparametrina testattavan aktiviteetin luokka. Tämän jälkeen ylliluokka instrumentoi luokan testausta varten.

`setUp()`-metodissa kutsutaan ylliluokan `setUp()`-metodia ja käynnistetään testattava aktiviteetti ylliluokan tarjoamalla `startActivity()`-metodilla. Tämän jälkeen viite testattavaan aktiviteettiin saadaan ylliluokan `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 ylliluokan `setUp()`-metodia, tai ylliluokan suorittamat alustukset jäävät tekemättä.

Listauksessa 4 on hieman monimutkaisempi testitapaus Robolectricilla samasta tes-

Listaus 3: Yksinkertainen aktiviteettiyksikkötesti ActivityUnitTestCase:n 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);
        assertEquals("gameView.getState()", GameState.RUNNING);
    }

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

Listaus 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();
    assertEquals(startedIntent.getComponent().getClassName(),
        equalTo(MainActivity.class.getName()));
    assertEquals(startedIntent.getStringExtra(GameActivity.SCORE)
        , equalTo("0.0"));
}

```

tiluokasta kuin 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 yksinkertaisempaa kuin Robolectricillä, koska AndroidUnitTestCase tarjoaa getStartedActivityIntent()-metodin, jolla saadaan aktiviteetin viimeisin lähettämä aie palautettua samalla tavalla kuin Robolectricin varjo-oliolta kysyttiin edellisessä

Listaus 5: Aikeen tilatietojen tarkastelu ActivityUnitTestCase:n avulla

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

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

listauksessa. Tämän jälkeen testin läpikäynnin 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 jäljittelijäkehysiä. Näissä testeissä jäljittelijäkehystenä käytetään Mockitoa, koska se toimii myös Androidissa sellaisenaan [moc]. Emulaattorissa ajamiseen tarvitsee vain Dalvik-käännöspaketin

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 MockGameActivity, 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 jäljittelijäkehys

Listaus 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
    }
}
```

riippuvuuksien eristämiseksi.

Tämän voisi Robolectricillä tehdä vaihtoehtoisesti siten, että toteuttaa oman varjoluokan 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 käskää käyttämään varjoluokkana 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 parametrilla.

Listaus 7: Jäljittelyä käyttävä testi Robolectricillä

```

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

```

Listaus 8: Jäljittelyä käyttävä testi AndroidUnitTestillä

```

public void
    testActivityInitializesGameWithRunningStateWithMock()
    throws Exception {
    GameView gameView = mock(GameView.class);
    MockGameActivity.setView(gameView);
    startActivity(new Intent(getInstrumentation().
        getTargetContext(), MockGameActivity.class), null,
        null);
    verify(gameView).setState(GameState.RUNNING);
}

```

AndroidUnitTestin avulla tehty vastaava testi listauksessa 8 on jäljittelijänäköymän 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 AndroidUnitTestille annetaan testattavan luokan nimi jo luokan määrittelyssä ja aktiviteetti käynnistetään ylliluokan 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: Testisarjan kestot Robolectricilla ja AndroidUnitTestillä

Testasin ensin isomman testisetin ajonopeutta. Kopioin aiemmin luvussa esitellyn testiluokan jäljittelijätestin 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 käynnistymiseen, vain itse testien suoritus aika. 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 jäljittelijätesti. Sen ajo kesti jokaisella ajokerralla yli 12 sekuntia, eli yli 25% koko testisarjan kestopa. 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 Yksikkötestauksen haasteita

Koodin laadun suhteen yksikkötestaus on herkkää: alunperin tarkoitukseni oli tehdä yksikkötestaus luvussa 6.4 esiteltä sovellusta vasten kuten toiminnallinen testauskin. Tämä ei kuitenkaan onnistunut, koska sovellusta ei ollut mitenkään tehty yksikkötestaus mielessä, osoittautui tämä mahdottomaksi.

Listaus 9: Tomdroid-koodiesimerkki

```

public class EditNote extends ActionBarActivity {
    ...
    protected void onCreate(Bundle savedInstanceState) {
        ...
        content = (EditText) findViewById(R.id.content);
        ...
        Intent intent = getIntent();
        uri = intent.getData();
    }

    public void onResume(){
        ...
        if (uri == null) {
            ...
        } else handleNoteUri(uri);
    }

    private void handleNoteUri(final Uri uri) {
        ...
        note = NoteManager.getNote(this, uri);
        if(note != null) {
            noteContent = note.getNoteContent(noteContentHandler);
        }
    }

    private Handler noteContentHandler = new Handler() {
        ...
        public void handleMessage(Message msg) {
            ...
            if(msg.what == NoteContentBuilder.PARSE_OK) {
                showNote(false);
            }
            ...
        }
    }

    private void showNote(boolean xml) {
        ...
        content.setText(noteContent, TextView.BufferType.
            SPANNABLE);
        ...
    }
}

```

Aloitin yrittämällä testata muistikirjan muokkausnäkyvässä, että aktiviteetin käynnistys lataa muistikirjan sisällön. Listauksessa 9 näkyy EditNote-aktiviteetin koodi, joka liittyy muistikirjan sisällön asettamiseen. onCreate()-metodin sisältö on suoraviivainen: content-muuttuja alustetaan ja aikeelta pyydetään uri, joka on muistikirjan tunniste. onResume()-metodissa kutsutaan yksityistä handleNoteUri()-metodia. Tässä metodissa pyydetään muistikirja NoteManagerin staattisella getNote()-metodilla ja jos muistikirja löytyy, pyydetään muistikirjalta sen sisältö getContent()-metodilla, joka ottaa parametrikseen käsittelijän, joka on EditNoten yksityinen sisäluokka.

Tässä kohdassa aktiviteetti muuttuu mahdottomaksi testata eristettynä riippuvuukistaan. Ensimmäinen ongelma on staattinen NoteManager, jota ei voi suoraan jäljitellä Androidissa toimivilla jäljittelijäkehyksillä. Tämä ongelma on kuitenkin kierretävissä vähäisellä ohjelmakoodimuutoksella: kääritään staattinen kutsu EditNoteen tehtävään uuteen metodiin getNote(), joka voidaan sitten jäljitellä testissä jäljittelialiluokassa kuten listauksessa 6 tehtiin.

Lopullinen este testaamiselle on kuitenkin kohta, jossa muistikirjalta pyydetään sen sisältö getContent()-metodilla. Metodi toimii siten, että sille annetaan parametrikseksi käsittelijä, joka on EditNoten yksityinen sisäluokka. Tämän käsittelijän sisältä handleMessage()-metodista taas kutsutaan showNote()-metodia, joka asettaa tekstin käyttöliittymään. Onnistuin jäljittelemään tämänkin toiminnan siten, että Noteolio oli jäljitelty ja sen getContent()-tyngästä kutsuttiin parametrina annetun käsittelijän handleMessage()-metodia. Tässä vaiheessa testi kaatui, koska oikeasti getContent()-metodi on asynkroninen ja getContent()-metodin palauttama noteContent ehditään asettaa ennen kuin käsittelijästä kutsutaan showNote()-metodia. Tässä vaiheessa yksinkertaista testiä varten oli kertynyt jäljittelijäkoodia useita kymmeniä rivejä ja totesin testaamisen epäkäytännölliseksi.

Tämä osoittaa, että ohjelmakoodin täytyy olla rakenteeltaan kelvollista ja luokkien ja metodien noudattaa mahdollisimman pitkälti yhden vastuun periaatetta (*single responsibility principle*, ks. [Mar03, 95-98]), jotta niiden yksikkötestaus on mahdollista. Lisäksi suurin hyöty yksikkötesteistä on ennen ohjelmakoodia kirjoitettuna, jolloin myös koodin rakenne pysyy helpommin testattavana.

5.9 Analyysi

Ohjelmakoodin yksikkötestaus onnistui hyvin sekä Androidin mukana tulevilla 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 ylikuokan tarjoamien alustusmetodien sijaan. Toisaalta joskus oli vaikea tietää, mitä metodeita eri luokkien valmiit varjototeutukset tarjoavat. Näissä tapauksissa `AndroidUnitTestCase`n ylikuokkametodit olivat käytössä selkeämpiä. Toisaalta Robolectric tarjoaa mahdollisuuden kirjoittaa itse omia varjoluokkia, joissa voi toteuttaa tynkiä Androidin kirjastoluokkien toiminnalle.

Jäljittelijäkehityksen käyttö onnistui ongelmitta sekä Robolectricillä että `AndroidUnitTestCase`lla. Mockito toimi suoraan yhdessä Robolectricin kanssa ja Android-emulaattorissa ajaminenkin onnistui helposti.

Suurin ero testityökalujen välillä oli testien suoritusnopeudella. Robolectric lupaa nopeita testejä ja toteuttaa lupauksensa; Robolectric-testit ajautuivat yli 25 kertaa nopeammin kuin emulaattorissa ajettut `AndroidUnitTest`it. Lisäksi `AndroidUnitTest`illä aika ensimmäiseen testitulokseen pienelläkin testiohjelmalla oli yli puoli minuuttia, joten testilähtöinen ohjelmointi `AndroidUnitTestCase`a 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.

Toinen kysymys on ylipäänsä aktiviteettien yksikkötestauksen mielekkyys. Hyvin rakennetussa sovelluksessa sovelluslogiikka on eriytetty käyttöliittymälogiikasta omiin luokkiinsa, joten sovelluslogiikan yksikkötestaus onnistuu myös tavallisella `JUnit`illa. Sovelluslogiikassa useimmiten on myös testauksen kannalta tärkeimmät osat. Käyttöliittymälogiikan yksikkötestausta tehokkaampaa voisi sen sijaan olla suora toiminnallinen testaus käyttöliittymän kautta.

6 Toiminnallisen testauksen työkalujen vertailua

Toiminnallisia testejä voi Androidilla tehdä monella eri työkalulla ja eri abstraktiotasolla. 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, eivätkä ne vaadi tietoa ohjelman sisäisestä rakenteesta. Koska Monkeyrunnerin testien arviointi perustuu kuvakaappausten avulla verifioimiseen, en käsittele sitä tässä. Käsittelen sen sijaan `UiAutomator`illa kirjoitettuja toiminnallisia testejä. Tämän lisäksi käsittelen Robotiumia, joka on Javalla käytettävä testityökalu sekä Troydia, joka käyttää Rubya testien tuottamiseen.

6.1 Robotium

Robotium [robb] 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.

Robotium-testejä 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 testaamista samassa testissä

- Testien kirjoittamisen nopeus
- Testikoodin selkeys
- Sitkeys, joka johtuu ajoaikaisesta sidonnasta käyttöliittymäkomponentteihin
- Nopea suoritusnopeus
- Helppo liittäminen jatkuvan integroinnin työkaluihin Antin tai Mavenin avulla.

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 ”paina nappia nimeltä x” tai ”tarkista, että ruudulla näkyy teksti y”, joten testien kirjoituksen pitäisi olla nopeaa. Lisäksi Troyd tarjoaa nauhoitus-toiminnon, jolla testiä voidaan kirjoittaa siten, että testiä kirjoitettaessa 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 `mix-in`-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 [JF12] mainitsevat Robotiumin vahvuudeksi Androidin omaa Instrumentatiota rikkaamman API:n. Esimerkiksi nappien painamiseen voidaan käyttää nappien nimeä, josta Robotium laskee napin sijainnin. He myös vertaavat Robotiumia omaan Troyd-työkaluunsa ja sanovat Robotiumin 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.

Benli et al. [BHH⁺12] 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.

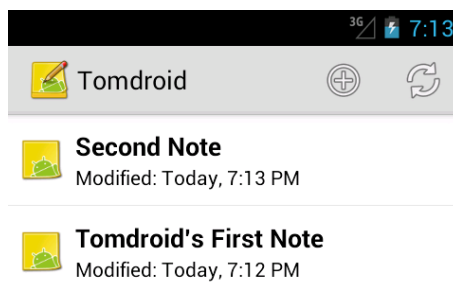
Zhongqian et al. [ZSJZ13] arvioivat Androidin mukana tulevien JUnit-pohjaisten testitapojen olevan liian hitaita ja vaikeita toiminnalliseen testaukseen. Robotiumilla taas on helpompi kirjoittaa tehokkaita ja sitkeitä testejä. Erityisen hyödylliseksi Zhongqian et al. arvioivat Robotiumin käyttötapaus-pohjaisten testien kirjoittamiseen. Lisäksi Robotiumia on heidän mielestään hyvä käyttää korkeamman tason testikehysten pohjana, kuten Troydissakin on tehty.

6.4 Testiprojekti

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 ja se on riittävän monimutkainen, jotta sille tehdyt testit kuvaisivat oikeassa Android-kehityksessä kohdattavia testaushaasteita. Tomdroidin kehitys on vasta beta-vaiheessa, joten sovelluksesta pitäisi löytyä myös bugeja, ja lisäksi sovelluksen oma testaus on lähes olematonta. Tätä tutkielmaa varten tein kopion Tomdroidin lähdekoodista versiosta 0.7.2 ja kopioin sen githubiin [toma].

Tomdroidissa olennaisimmat näkymät ovat muistikirjalista, josta on ruutukaappaus kuvassa 11, yksittäisen muistikirjan selaaminen (kuvassa 12) ja sen editointi (kuvassa 13).

Listanäkymässä näkyvät kaikki käyttäjän muistikirjat päivitysajan mukaan järjes-

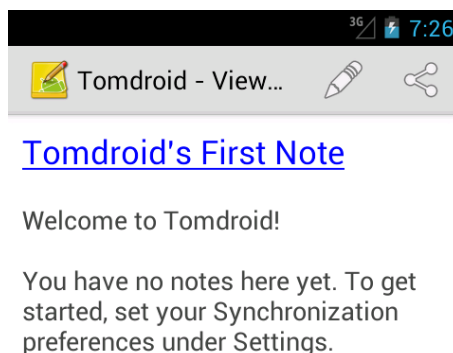


Kuva 11: Muistikirjalista Tomdroidissa

tettynä. 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.



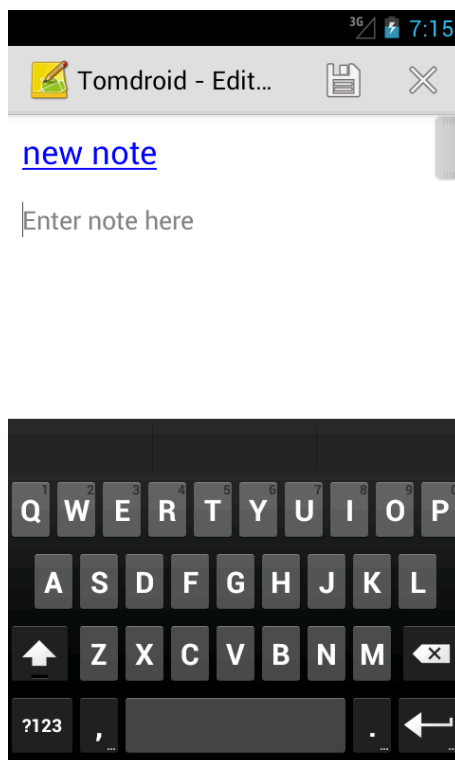
Kuva 12: Muistikirjan luku Tomdroidissa

6.5 Testitapaus

Testaan samaa testitapausta jokaisella työkalulla. Testitapauksessa luodaan uusi muistikirja, jonka otsikoksi kirjoitetaan "new note" ja tallennetaan se.

Ensimmäisenä sovellus käynnistetään ja varmistetaan, ettei muistikirjalistassa ole vielä "new note"-nimistä muistikirjaa. Tämän jälkeen painetaan yläpalkin +-symbolia, josta päästään uuden muistikirjan muokkausnäky-mään. Muokkausnäky-mässä valitaan otsikon muokkaus -kenttä ja kirjoitetaan siihen "new note". Sitten muistikirja tallennetaan painamalla yläpalkin tallennus-symbolia. Tallentamisen jäl-keen palataan listanäkymään painamalla yläpalkin vasemman reunan ikonista. Lis-tanäkymässä varmistetaan, että muistikirja on tallentunut katsomalla, että listasta löytyy nyt "new note"-niminen muistikirja.

Testitapaus testaa Tomdroidin perustoiminnallisuudet: muistikirjan luonnin, muok-kauksen, tallennuksen ja listanäkymän. Testitapauksen toteuttaminen vaatii, että testityökalussa on perustoiminnallisuudet: elementtejä voi etsiä tekstin perusteella, erilaisia ikoneita eri puolilla näyttöä voi painaa ja tekstikenttiin voi syöttää tekstiä.



Kuva 13: Muistikirjan editointi Tomdroidissa

6.6 Asennukset

Robotiumin asennus on yksinkertaista. Projektin käännöspolkuun tarvitsee vain lisätä Robotiumin jar-paketti, jossa tulee kaikki tarvittava mukana. Itse Robotium-testit perivät Androidin omasta `ActivityInstrumentationTestCase2`-yliluokasta. Listauksessa 10 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 resetoidaan takaisin alkuperäiseen tilaan, jotta testejä voi toistaa useita kertoja luotettavasti. Tomdroidin tapauksessa tämä tehtäisiin kutsumalla `tearDown()`:ssa apumetodia `removeAllNotes()`, joka on esitetty listauksessa 11. Metodi kutsuu Tomdroidin `NoteManager`in `deleteAllNotes()`-metodia, joka poistaa kaikki sovelluksen tallentamat muistikirjat.

Listaus 10: Robotium-testirunko

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

    private Solo solo;

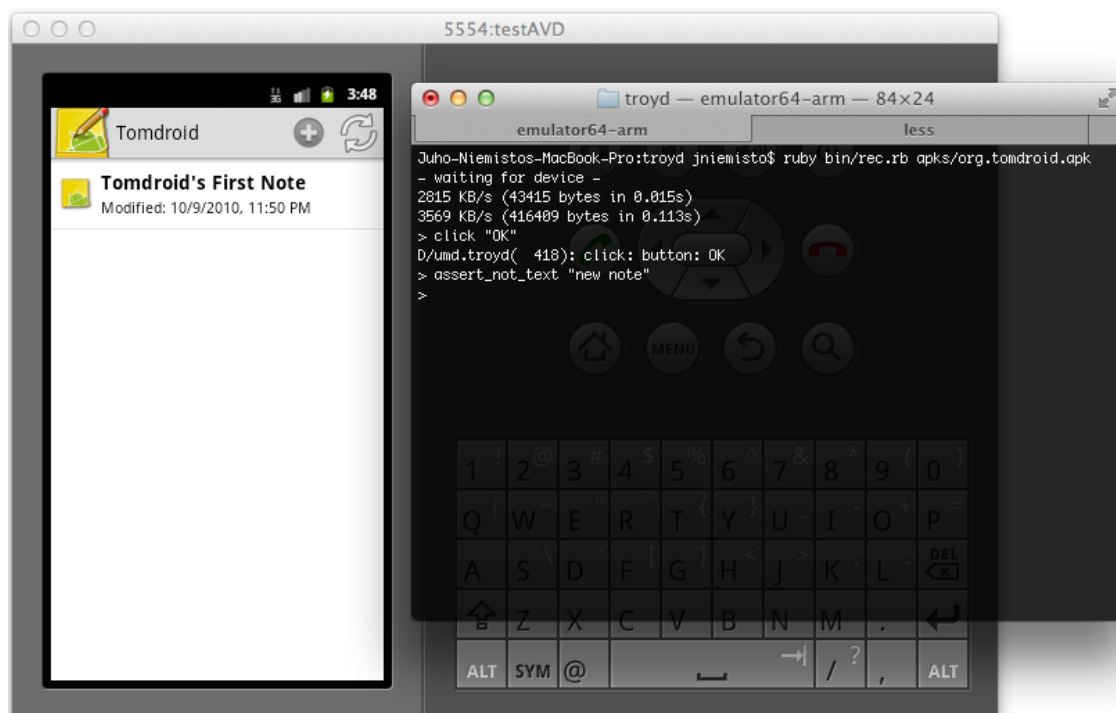
    public RobotiumTest() {
        super(Tomdroid.class);
    }

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

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

Listaus 11: Muistikirjojen poisto

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



Kuva 14: Troydin nauhoituskripti ja emulaattori

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 ja sen päivitykset loppuivat kesäkuussa 2013 [rub], joten joissain järjestelmissä näin vanhan version asennus voi olla haasteellista. Asennusten jälkeen Troydin testin kirjoittaminen onnistui kuitenkin nopeasti nauhoituskriptin avulla, joka hoiti kaiken muun kuin itse testikoodin kirjoittamisen. Testit voivat olla myös eri sovelluksille, jolloin Troyd asentaa kunkin testattavan sovelluksen erikseen. Troydin nauhoituskripti ja emulaattori vastaavassa testin kohdassa on esitetty kuvassa 14.

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 on asennettuna. Testien kääntäminen ja ajaminen on esitetty listauksessa 12. Testit käännetään Apache Antilla [ant], joka on avoimen lähdekoodin käännöstyökalu Java-sovelluksille. Toisella rivillä käytetään

Listaus 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
```

Listaus 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 editointinäkömää", EditNote.class);
    solo.enterText(0, "new note");
    solo.clickOnActionBarItem(R.id.edit_note_save);
    solo.clickOnActionBarHomeButton();
    solo.assertCurrentActivity("Koti-näppäimen painaminen
        ei vienyt takaisin muistikirjalistaan", Tomdroid.
        class);
    assertTrue(solo.searchText("new note"));
}
```

Androidin mukana tulevaa adb-työkalua (*Android debug bridge*) käännetyn testisovelluksen siirtämiseen puhelimeen ja kolmannella rivillä ajetaan testisovelluksesta luokka `org.tomdroid.test.UiAutomatorTest`.

6.7 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 varmistamaan, että testi alkaa muistikirjalistasta. Toisella rivillä varmistetaan, että testissä luotavaa muistikirjaa ei vielä löydy listasta. Seuraavalla rivillä painetaan yläpalkin uuden muistikirjan luovaa nappia `clickOnActionBarItem()`-

Lista 14: Muistikirjan luontitesti Troydilla

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

metodilla. Se ottaa parametrina id:n siitä komponentista, johon ollaan painamassa. Tämän jälkeen pitäisi avautua uusi muistikirja editointinäkömään, 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.

6.8 Troyd-testit

Troyd-testi, jossa luodaan muistikirja, on esitetty listauksessa 14. Troyd-testit asen-
tavat aina ensimmäisenä uuteen emulaattoriin koko sovelluksen, joten Robotium-
testissä tarvittuja vanhojen muistikirjojen poistoa ei tarvita. Tein testin Troydin
nauhoituskriptin avulla, joten itse testi on automaattisesti kirjoitettu tämän poh-
jalta. Testitiedosto sisältää testimetodin lisäksi metodit sovelluksen alustamiselle
ja lopettamiselle sekä testeissä käytetyt Ruby-metodit, mutta nämä kaikki Troyd
tuotti automaattisesti nauhoituskriptin avulla. Itse testimetodin sisältö on sama
kuin testiskriptissä kirjoittamani komennot. Lisäksi näin sovelluksen testiä vastaa-
vassa tilassa emulaattorissa testiä kirjoittaessa, mikä helpotti testin kirjoittamista.
Listauksesta on lisäksi huomattava, että Rubyssa metodien sulut ovat vapaaehtoi-
set, mikäli sekaantumisen vaaraa ei ole. Siksi testissä metodien parametrit eivät ole
sulkujen sisällä. Kun nauhoituskriptissä on saanut haluamansa testitapauksen val-
miiksi, se tallennetaan sofar-metodilla, jonka parametrina on testin nimi, kuten on
tehty listauksessa 15. Finish-komento lopettaa nauhoituskriptin.

Listaus 15: Testin tallennus nauhoitusskriptistä Troydilla

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

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 vierailuista 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.

6.9 Uiautomator-testit

Listauksessa 16 on esitetty Uiautomator -testi, jossa luodaan uusi muistikirja. Testi perii ylliluokan 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 resurssitiedoston kautta annettua. className()-metodi etsii kaikki elementit, jotka ovat annetun luokan ilmentymiä, ja palauttaa niistä ensimmäisen.

UiObjectin metodeista testeissä käytetään seuraavia: clickAndWaitForNewWindow() painaa käyttöliittymäelementtiä ja odottaa seuraavan aktiviteetin latautumista,

Listaus 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);
    }
}
```

Listaus 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();
}
```

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.

6.10 Testien suoritusnopeudet

	Keskiarvo (s)	Max (s)	Min (s)
Robotium	14,207	14,303	14,147
Troyd	143,778	145,961	142,304
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. Testiajoon sisältyy vain muistikirjan luontitesti kerran ajettuna.

Robotium-testit ajoin Eclipsestä Android JUnitilla emulaattorissa. Ajat ovat Eclipse-
sen ottamia. Emulaattori oli valmiiksi käynnissä testejä ajettaessa. Testiajoon sisäl-
tyy siis sovelluksen asennus emulaattoriin ja testin ajaminen.

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 Uiautomatorilla ajaessa.

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

Troyd-testit osoittautuivat selvästi hitaammaksi kuin Robotium ja Uiautomator. Robotium-testit olivat hieman Uiautomator -testejä nopeammiksi, mutta ero ei ole kovin merkittävä.

6.11 Analyysi

Perusominaisuuksien testaus onnistui niin Troydilla, Robotiumilla kuin UI Automatorillakin. Kaikki työkalut täyttivät siis toiminnallisen testityökalun perusvaatimukset.

Troydilla testaus on korkeimmalla abstraktiotasolla testatuista työkaluista. Ohjelman tilan seuraaminen testiä nauhoitettaessa nopeuttaa ja helpottaa testien kir-

joitusta. Rubyllle tehty API on ilmaisuvoimainen. Troyd on kuitenkin selvästi vielä kehittymätön. Se tukee vain vanhaa Android- ja Ruby-versiota ja testien suoritus on kymmenen kertaa hitaampaa kuin muilla testatuilla kehyksillä - joskin testien suoritusnopeus ei toiminnallisessa testauksessa ole yhtä kriittistä kuin yksikkötestauksessa, koska testejä ei useimmiten ajeta itse tdd-kehityssyklissä, vaan jatkuvan integraation työkaluilla, jolloin nopean palautteen saaminen ei ole yhtä tärkeää kehitysnopeudelle. Isommissa sovelluksissa tosin testien ajaminen jatkuvan integraation työkaluissakin voi alkaa hidastaa prosessia. Lisäksi Troyd vaatii erillisen ohjelmointiympäristön normaaliin Android-kehitykseen verrattuna, eivätkä kaikki Android-kehittäjät osaa Rubya.

Robotiumilla kirjoitetut testit ovat myös melko ilmaisuvoimaisia ja tiiviitä. Robotiumin API:n oppii helposti ja testejä tehdään samoilla työkaluilla kuin muutakin Android-kehitystä. Robotium myös tukee kaikkia Androidin versioita. Robotiumin käyttö vaatii kuitenkin hieman enemmän tietoa testattavasta sovelluksesta. Suoritusnopeudeltaan Robotium-testit olivat testatuista testikehyksistä nopeimmat.

Uiautomatorin erikoisuus on siihen liittyvä Uiautomatorviewer, jonka avulla testattavan sovelluksen käyttöliittymäelementit on helppo analysoida ilman sovelluksen lähdekoodia. Työkalu nopeuttaa testien kirjoittamista huomattavasti. Tietoja voisi myös käyttää Robotium-testien kirjoittamisessa. Itse Uiautomator-testit ovat hieman verboosimpia kuin Robotium-testit, mutta niiden kirjoittaminen on hyvin suoriinvaista. Uiautomatorin suurin puute on, että se toimii vain uusimmissa Android-versioissa, joten regressiotestaus vanhempien laitteiden toiminnan varmistamiseksi ei ole mahdollista.

Työkaluista ainoastaan Robotium toimii ilman varauksia erilaisissa Android-ympäristöissä ja on siten vahvin valinta toiminnallisen testauksen työkaluksi. Uiautomator on mielenkiintoinen, mutta sen toiminta vain uusimmissa Android-versioissa rajoittaa toistaiseksi sen käyttökelpoisuutta. Troyd taas vaatii kehitystä tullakseen vartenotettavaksi vaihtoehdoksi.

7 Yhteenveto

Android-sovellukset koostuvat Android-kohtaisista komponenteista: aktiviteeteista, palveluista ja sisällöntarjoajista, sekä tavallisista Java-luokista. Aktiviteetti kuvaa käyttäjälle näkyvää näkymää. Palvelut taas on tarkoitettu pitkäkestoisten taustaoperaatioiden suoritukseen. Sisällöntarjoajat tarjoavat rajapinnan sovelluksen tarvitsemaan tietoon ja mahdollistavat sovellustenvälisen tietojenvaihdon.

Androidin arkkitehtuuri on vahvasti tapahtumapohjaista. Aktiviteetit ja palvelut toteuttavat takaisinkutsumetodit komponenttien elinkaaren hallintaan. Komponentit eivät ole suoraan yhteydessä toisiinsa vaan niiden välillä kommunikoidaan tapahtumapohjaisilla aikeilla, jotka järjestelmä välittää vastaanottavalle komponentille.

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 kehittämää testaustyökaluja.

Tässä tutkielmassa vertailtiin Androidin omia ja kolmansien osapuolien yksikkö- ja toiminnallisen testauksen työkaluja. Nämä työkalut valitsin huomion kohteeksi, koska toiminnallinen ja yksikkötestaus ovat yleisimmät testauksen muodot mobiili-sovelluksia kehitettäessä ja työkaluvalikoima on myös monipuolisin. Toiminnallinen ja yksikkötestaus ovat myös useimmiten automatisoituja testaamisen vaiheita.

Yksikkötestauksessa haasteita tuo se, että vaikka Android-sovelluksia ohjelmoidaan Javalla, Androidin kirjastoluokat eivät toimi suoraan Javan omassa virtuaalikoneessa, vaan testit on ajettava Dalvik-ajoympäristössä emulaattorilla tai Android-laitteessa. Tämä hidastaa testien ajamista. Robolectric-yksikkötestaustyökalu mahdollistaa yksikkötestien ajamisen suoraan Javan virtuaalikoneella.

Robolectric osoittautui toimivaksi vaihtoehdoksi Androidin omalle yksikkötestityökalulle. Sen käyttö oli käytännössä yhtä helppoa ja testien ajoaika oli moninkertain nopeampi kuin Androidin omien yksikkötestien emulaattorissa. Itse testikoodista tuli kummallakin työkalulla lähes samanlaista, eikä testien kirjoittamiseen tarvittu kovin paljoa Android-spesifiä osaamista. Testattavien komponenttien alustus oli Robolectricillä jopa helpompaa, koska konstruktoreja saattoi kutsua suoraan testikehyksen tarjoamien alustusmetodien sijaan. Toisaalta aina ei voinut tietää, mitä metodeita eri luokkien valmiit varjototeutukset tarjoavat. Tämä oli selkeämpää Androidin omaa yksikkötestikehystä käytettäessä. Toisaalta Robolectric tarjoaa mahdolli-

suuden kirjoittaa itse omia varjoluokkia, joissa voi toteuttaa tynkiä Androidin kirjastoluokkien toiminnalle.

Molemmilla yksikkötestitavoilla oli myös helppo käyttää jäljittelijäkehystä testeissä. Mockito toimi suoraan yhdessä Robolectricin kanssa ja Android-emulaattorissa ajaminenkin onnistui ilman suurempia ongelmia.

Suurin ero työkalujen välillä oli ajonopeuksissa. Robolectric lunasti lupauksensa nopeista testeistä; Robolectric-testit ajautuivat yli 25 kertaa nopeammin kuin emulaattorissa ajatut `AndroidUnit`Testit. Lisäksi `AndroidUnit`Testillä ajettaessa ensimmäisen testinkin ajautuminen pienellä testiohjelmalla kesti yli puoli minuuttia, joten testilähtöinen ohjelmointi `AndroidUnit`Testiä käyttäen on käytännössä toivottoman hidasta.

Yksikkötestauksessa pohdin Android-sovelluksen aktiviteettien yksikkötestauksen mielekkyyttä. Hyvin rakennetussa sovelluksessa sovelluslogiikka on eriytetty käyttöliittymälogiikasta omiin luokkiinsa, joten sovelluslogiikan yksikkötestaus onnistuu myös tavallisella `JUnit`illa. Sovelluslogiikassa useimmiten on myös testauksen kannalta tärkeimmät osat. Käyttöliittymälogiikan yksikkötestausta tehokkaampaa voisi sen sijaan olla suora toiminnallinen testaus käyttöliittymän kautta. En testannut tässä tutkielmassa muita Androidin peruskomponentteja, joten on mahdollista, että palveluiden tai sisällöntarjoajien yksikkötestaus olisi aktiviteettien yksikkötestausta mielekkäämpää.

Toiminnallisen testauksen työkaluista vertailtiin `UiAutomator`ia, `Robotium`ia ja `Tro`ydia. Kukin näistä on toimintatavaltaan hieman erilainen.

`UiAutomator` on Androidin mukana tuleva työkalu. `UiAutomator`-testien kirjoittaminen on melko verboosia, mutta siihen liittyvä käyttöliittymän analysoiva `UiAutomator`viewer tekee testien kirjoittamisesta helppoa ilman, että sovelluksen lähdekoodia tarvitsee tuntea. `UiAutomator`in suurin puute on, että se toimii vain uusimmissa Android-versioissa, joten regressiotestaus vanhempien laitteiden toiminnan varmistamiseksi ei sillä onnistu.

`Tro`yd taas toimii vain Androidin versiolla 2.3.6. Sen erikoisuus on nauhoitusskripti, jonka avulla testiä kirjoitettaessa näkee jatkuvasti sovelluksen tilan testin kussakin vaiheessa. `Tro`ydilla testaus on korkeimmalla abstraktiotasolla testatuista työkaluista ja sillä kirjoitetut testit ovat hyvin tiiviitä. `Tro`yd kuitenkin tarvitsisi runsaasti viimeistelyä, jotta sen käyttöä voisi suositella. Myös testien suoritus aika oli kymmenen kertaa hitaampaa kuin muilla testatuilla työkaluilla.

Robotium osoittautui hyväksi toiminnallisen testauksen työkaluksi, joka toimii varkaasti Androidin eri versioilla, joskin testien kirjoittaminen oli hieman Uiautomatoria tai Troydia haastavampaa. Testit olivat kuitenkin suhteellisen tiiviitä. Testien ajoaika oli Robotiumilla ja Uiautomatorilla suunnilleen yhtä nopeaa.

Androidin testaustyökalujen tilanne kokonaisuudessaan on varsin hyvä. Google on ottanut testauksen huomioon Androidia kehittäessä ja tarjoaa Androidin kehitystyökalujen mukana melko kattavan paketin erilaisia testityökaluja ohjelmistotuotantoprosessin eri vaiheissa. Tämän lisäksi kolmansien osapuolien kehittämät testaus työkalut täydentävät Androidin omien testaustyökalujen aukkoja.

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, SIGCSE '12*, New York, 2012, ACM, sivut 547–552, URL <http://doi.acm.org/10.1145/2157136.2157293>.
- 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. *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA '04*, New York, 2004, ACM, sivut 174–175, URL <http://doi.acm.org/10.1145/1028664.1028736>.
- 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. Haettu 23.10.2013.
- andb Android developer documentation. URL <http://developer.android.com/>. Haettu 23.10.2013.
- andc Testaus androidin kehittäjädokumentaatioissa. URL http://developer.android.com/tools/testing/testing_android.html. Haettu 23.10.2013.
- ant Apache ant. URL <http://ant.apache.org/>. Haettu 23.10.2013.
- Bec03 Beck, K., *Test-driven development: by example*. Addison-Wesley, Boston, 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. *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*. IEEE, 2012, sivut 263–268.
- ecl Eclipsen kotisivut. URL <http://www.eclipse.org>. Haettu 23.10.2013.

- GLM⁺08 Godefroid, P., Levin, M. Y., Molnar, D. A. et al., Automated whitebox fuzz testing. *Proceedings of 16th Annual Network Distributed System Security Symposium*, osa 8, 2008, sivut 151–166.
- goo Google play:n kotisivut. URL <http://play.google.com>. Haettu 23.10.2013.
- 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>.
- kin Amazon kindle wikipediassa. URL http://en.wikipedia.org/wiki/Amazon_Kindle. Haettu 23.10.2013.
- Mar03 Martin, R. C., *Agile Software Development - Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, N.J., 2003.
- mav Apache mavenin kotisivut. URL <http://maven.apache.org/>. Haettu 23.10.2013.
- MBS02 Michael, J. B., Bossuyt, B. J. ja Snyder, B. B., Metrics for measuring the effectiveness of software-testing tools. *Proceedings of 13th International Symposium on Software Reliability Engineering*. IEEE, 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, Boston, 2001, sivut 287–301.
- moc Mockitoon kotisivut. URL <https://code.google.com/p/mockito/>. Haettu 23.10.2013.
- mona Monkey androidin kehittäjädokumentaatioissa. URL <http://developer.android.com/tools/help/monkey.html>. Haettu 23.10.2013.
- monb Monkeyrunner androidin kehittäjädokumentaatioissa. URL http://developer.android.com/tools/help/monkeyrunner_concepts.html. Haettu 23.10.2013.
- PS92 Poston, R. M. ja Sexton, M. P., Evaluating and selecting testing tools. *Software, IEEE*, 9,3(1992), sivut 33–42.

- pub Sovelluksen julkaisun dokumentaatio androidin kehittäjä sivuilla. URL <http://developer.android.com/distribute/index.html>. Haettu 23.10.2013.
- PY08 Pezzè, M. ja Young, M., *Software Testing And Analysis*. John Wiley Sons, Hoboken, N.J., 2008.
- roba Robolectricin kotisivut. URL <http://roboelectric.org/>. Haettu 23.10.2013.
- robb Robotiumin kotisivut. URL <http://code.google.com/p/robotium/>. Haettu 23.10.2013.
- rub Ilmoitus ruby 1.8.7 -tuen päättymisestä. URL <http://www.ruby-lang.org/en/news/2013/06/30/we-retire-1-8-7/>. Haettu 23.10.2013.
- rvm Rvm:n kotisivut. URL <https://rvm.io>. Haettu 23.10.2013.
- sam Robolectric-testiprojekti. URL <https://github.com/jmschultz/Eclipse-Robolectric-Example>. Haettu 23.10.2013.
- sel Seleniumin kotisivut. URL <http://seleniumhq.org/>. Haettu 23.10.2013.
- SLS09 Spillner, A., Linz, T. ja Schaefer, H., *Software Testing Foundations*. Rocky Nook, Santa Barbara, CA, 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, New York, 2011, sivut 163–175.
- tes Test::unitin kotisivut. URL <http://test-unit.rubyforge.org/>. Haettu 23.10.2013.
- toma Tomdroid 0.7.2 lähdekoodi. URL <https://github.com/jniemisto/tomdroid>. Haettu 23.10.2013.
- tomb Tomdroidin kotisivut. URL <https://code.launchpad.net/tomdroid>. Haettu 23.10.2013.

- tro Troydin lähdekoodi. URL <https://github.com/plum-umd/troyd>.
Haettu 23.10.2013.
- uia Uiautomator androidin kehittäjädokumentaatiassa. URL <http://developer.android.com/tools/help/uiautomator/index.html>.
Haettu 23.10.2013.
- wik Android (operating system). URL [http://en.wikipedia.org/wiki/Android_\(operating_system\)](http://en.wikipedia.org/wiki/Android_(operating_system)). Haettu 23.10.2013.
- ZG11 Zechner, M. ja Green, R., *Beginning Android 4 Games Development*. Springer, New York, 2011.
- ZSJZ13 Zhongqian, W., Shu, L., Jinzhe, L. ja Zengzeng, L., Keyword-driven testing framework for android applications. *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, 2013, sivut 1096–1102.

Liite 1. Yksikkötesteissä testatun sovelluksen lähdekoodi

Lista 18: GameActivity.java

```
package com.example.demo;

import com.example.demo.GameView.GameState;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class GameActivity extends Activity implements
    OnGameEndEventListener {
    public final static String SCORE = "com.example.demo.SCORE"
        ;
    private GameView mGameView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.game_layout);
        mGameView = (GameView) findViewById(R.id.gameview);
        mGameView.setGameEndEventListener(this);
        mGameView.setState(GameState.RUNNING);
    }

    @Override
    protected void onPause() {
        mGameView.setState(GameState.PAUSED);
        super.onPause();
    }

    @Override
    protected void onStop() {
        mGameView.setState(GameState.PAUSED);
        super.onStop();
    }

    @Override
    protected void onRestart() {
```

```

        mGameView.setState(GameState.RUNNING);
        super.onRestart();
    }

    @Override
    protected void onResume() {
        mGameView.setState(GameState.RUNNING);
        super.onResume();
    }

    public void onGameEnd(double score) {
        Intent intent = new Intent(this, MainActivity.class);
        intent.putExtra(SCORE, String.valueOf(score));
        startActivity(intent);
    }
}

```

Listaus 19: GameView.java

```

package com.example.demo;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.os.Handler;
import android.os.Message;
import android.util.AttributeSet;
import android.view.MotionEvent;
import android.view.View;

public class GameView extends View {

    public enum GameState {
        READY, RUNNING, PAUSED, LOST;
    }

    private RefreshHandler mRedrawHandler = new RefreshHandler
        ();
    private long mMoveDelay = 50;
    private List<Circle> mCircles;

```

```

private Circle mPlayer;
private GameState mState;
private GameClock mClock;
private OnGameEndEventListener mOnGameEndEventListener;

public GameView(Context context, AttributeSet attributes) {
    super(context, attributes);
    mClock = new GameClock();
    initPlayer();
    initCircles();
    mState = GameState.READY;
}

private void initPlayer() {
    Paint paint = new Paint();
    paint.setARGB(255, 255, 0, 0);
    mPlayer = new Circle(20, 100, 100, paint);
}

private void initCircles() {
    mCircles = new ArrayList<Circle>();
    Paint circlePaint = new Paint();
    circlePaint.setARGB(255, 0, 0, 255);
    Random random = new Random();
    int numCircles = 3+random.nextInt(3);
    for (int i=0;i<numCircles;i++) {
        int radius = random.nextInt(35)+5;
        int speedX = random.nextInt(20)-10;
        int speedY = random.nextInt(20)-10;
        Circle circle = new Circle(radius, 300, 550, speedX,
            speedY, 450, 700, circlePaint);
        mCircles.add(circle);
    }
}

public void setState(GameState newState) {
    GameState oldState = mState;
    mState = newState;
    if (oldState != GameState.RUNNING && newState ==
        GameState.RUNNING) {
        mClock.start();
        update();
    }
    if (newState != GameState.RUNNING) {

```



```

        mClock.pause();
    }
    if (mState == GameState.LOST && mOnGameEndEventListener
        != null) {
        mOnGameEndEventListener.onGameEnd(mClock.getTime()
            /1000.0);
    }
}

public GameState getState() {
    return mState;
}

public void setGameEndEventListener(OnGameEndEventListener
    onGameEndEventListener) {
    mOnGameEndEventListener = onGameEndEventListener;
}

@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawCircle(mPlayer.getX(), mPlayer.getY(), mPlayer
        .getRadius(), mPlayer.getPaint());
    for (Circle circle : mCircles) {
        canvas.drawCircle(circle.getX(), circle.getY(), circle.
            getRadius(), circle.getPaint());
    }
}

protected void update() {
    if (mState == GameState.RUNNING) {
        for (Circle circle : mCircles) {
            circle.update();
            if (circle.collisionsWith(mPlayer)) {
                setState(GameState.LOST);
            }
        }
        mRedrawHandler.sleep(mMoveDelay);
    }
}

@Override
public boolean onTouchEvent(MotionEvent event) {
    if (mState == GameState.RUNNING) {

```

```

        mPlayer.setX((int)event.getX());
        mPlayer.setY((int)event.getY());
        invalidate();
    }
    return super.onTouchEvent(event);
}

class RefreshHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        GameView.this.update();
        GameView.this.invalidate();
    }

    public void sleep(long delayMillis) {
        this.removeMessages(0);
        sendMessageDelayed(obtainMessage(0), delayMillis);
    }
}
}

```

Listaus 20: MainActivity.java

```

package com.example.demo;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);
        showScore();
    }

    private void showScore() {
        Intent intent = getIntent();
        String message = intent.getStringExtra(GameActivity.SCORE
        );
    }
}

```

```

        TextView scoreText = (TextView) findViewById(R.id.
            scoretext);
        if (message != null && !message.equals("")) {
            scoreText.setText("Selvisit "+message+" sekuntia!");
        }
    }

    public void startGame(View view) {
        Intent intent = new Intent(this, GameActivity.class);
        startActivity(intent);
    }
}

```

Listaus 21: Circle.java

```

package com.example.demo;

import android.graphics.Paint;

public class Circle {

    private final int mRadius;
    private int mPositionX;
    private int mPositionY;
    private int mSpeedX;
    private int mSpeedY;
    private final Paint mPaint;
    private final int mWorldWidth;
    private final int mWorldHeight;

    public Circle(final int radius, final int positionX, final
        int positionY, final int speedX, final int speedY, final
        int worldWidth, final int worldHeight, final Paint
        paint) {
        this.mRadius = radius;
        this.mPositionX = positionX;
        this.mPositionY = positionY;
        this.mSpeedX = speedX;
        this.mSpeedY = speedY;
        this.mWorldWidth = worldWidth;
        this.mWorldHeight = worldHeight;
        this.mPaint = paint;
    }
}

```

```

public Circle(final int radius, final int positionX, final
    int positionY, Paint paint) {
    this(radius,positionX,positionY,0,0,0,0,paint);
}

public int getX() {
    return mPositionX;
}

public int getY() {
    return mPositionY;
}

public void setX(int x) {
    mPositionX = x;
}

public void setY(int y) {
    mPositionY = y;
}

public Paint getPaint() {
    return mPaint;
}

public int getRadius() {
    return mRadius;
}

public void update() {
    mPositionX += mSpeedX;
    if (mPositionX > mWorldWidth-mRadius || mPositionX <
        mRadius) {
        mSpeedX = -1 * mSpeedX;
    }
    mPositionY += mSpeedY;
    if (mPositionY > mWorldHeight - mRadius || mPositionY <
        mRadius) {
        mSpeedY = -1 * mSpeedY;
    }
}

public boolean collisionWith(Circle circle) {
    Double distance = Math.sqrt(Math.pow(circle.getX() -

```

```

        mPositionX, 2) + Math.pow(circle.getY()-mPositionY, 2)
    );
    return distance < mRadius + circle.getRadius();
}
}

```

Lista 22: GameClock.java

```

package com.example.demo;

public class GameClock {

    private long mStartTime;
    private long mPausedAt;
    private boolean mRunning;

    public GameClock() {
        mRunning = false;
        mStartTime = 0;
        mPausedAt = 0;
    }

    public void start() {
        if (!mRunning) {
            mStartTime = System.currentTimeMillis() - mPausedAt;
        }
        mRunning = true;
    }

    public void pause() {
        if (mRunning) {
            mPausedAt = System.currentTimeMillis() - mStartTime;
        }
        mRunning = false;
    }

    public long getTime() {
        if (mRunning) {
            return System.currentTimeMillis() - mStartTime;
        }
        return mPausedAt;
    }
}

```

Listaus 23: OnGameEndEventListener.java

```
package com.example.demo;

public interface OnGameEndEventListener {
    public void onGameEnd(double score);
}
```

Listaus 24: game_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/
    res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.example.demo.GameView
        android:id="@+id/gameview"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

Listaus 25: main_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/
    res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/start_game"
        android:onClick="startGame" />

    <TextView
        android:id="@+id/scoretext"
        android:text=""
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

</LinearLayout>

Listaus 26: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
    package="com.example.demo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="15" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        <activity
            android:name=".MainActivity"
            android:label="@string/title_game" >

                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />
                    <category android:name="android.intent.category.
                        LAUNCHER" />
                </intent-filter>
            </activity>

            <activity
                android:name=".GameActivity"
                android:label="@string/title_game" >
            </activity>
        </application>
    </manifest>
```