hyväksymispäivä arvosana

arvostelija

Android-sovellusten testaaminen

Juho Niemistö

Helsinki 11.3.2013

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

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

Tekijä — Författare — Author
Juho Niemistö

Työn nimi — Arbetets titel — Title

Android-sovellusten testaaminen

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	11.3.2013	51 sivua

Tiivistelmä — Referat — Abstract

Ääkkönen

Avainsanat — Nyckelord — Keywords
Android

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Tällä hetkellä copypaste suoraan aihe-esittelystä.. Refaktoroin sitten kun muu sisältö alkaa olla paremmin hahmotettuna

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 noin 500 000.

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

Sovellusten laatu on erityisen tärkeää Android-alustalla, jossa kilpailua on runsaasti ja sovellusten hinta niin alhainen, ettei se muodosta estettä sovelluksen vaihtamista toiseen. Sovelluskauppa on myös aina saatavilla suoraan laitteesta.

Gradun tavoitteena on syventyä Android-sovellusten testaamiseen ja testaustyökaluihin. 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 gradussa vain Androidille javalla kehitettyjä natiivi-sovelluksia. Androidille voi kehittää sovelluksia myös muunmuassa html5:llä ja erilaisilla työkaluilla, jotka generoivat automaattisesti natiivikoodia useille mobiilialustoille. Osa testaustyökaluista mahdollistaa toki myös tällaisten sovellusten testaamisen. Keskityn vain automaattisiin testaustyökaluihin, joten esimerkiksi manuaaliseen käytettävyystestaukseen liittyvät prosessit ja työkalut on rajattu tämän työn ulkopuolelle.

Aihe-esittelyssä listatut alustavat lähteet: [TKH11] [HSJ11] [MYC+11] [Spa10] [KM10] [MMP+12] [HN11] [Was10] [HP11]

2 Android

Perusjutut androidista.

2.1 Historia

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

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

Tähän vähän kattavammin jotain..

2.2 Androidin kehitystyökalut

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

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

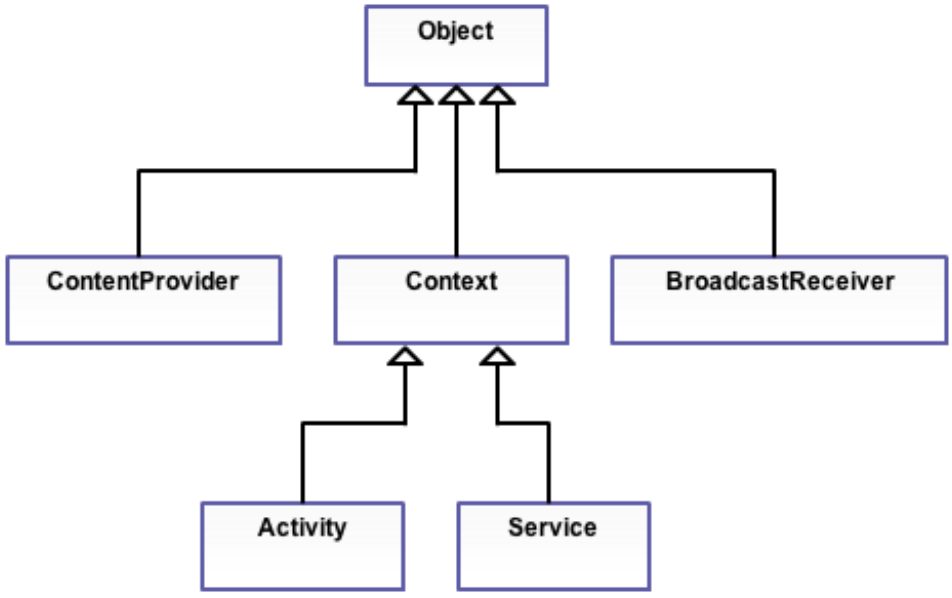
Sovelluksen ja testien ajamista varten SDK:n mukana tulee Android-emulaattori. Emulaattoreita voi ajaa eri Android API:n versioilla ja laitteistoprofiileilla, jotta on mahdollista testata sovelluksen toimivuutta erilaisissa Android-ympäristöissä. Emu-

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

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

2.3 Android-sovellusten rakenne

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



Kuva 1: Androidin tärkeimpien komponenttien luokkahierarkia

Android-sovellukset koostuvat neljästä komponenttityypistä: aktiviteeteista (acti-

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

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

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

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

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

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

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

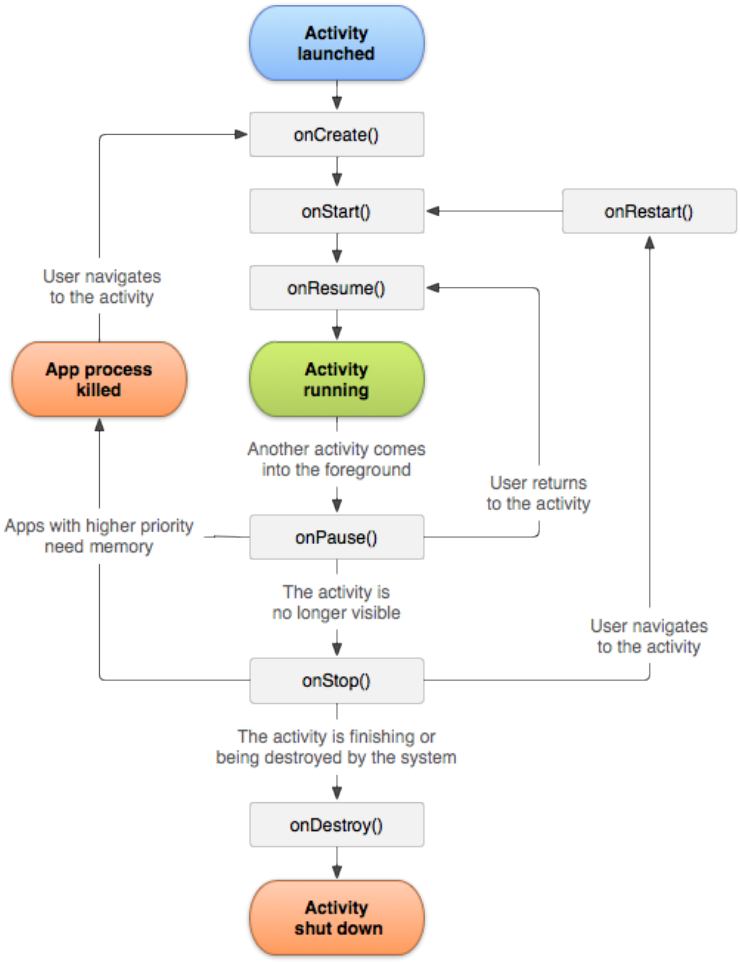
Android-sovelluksilla ei ole yksittäistä main-metodia, joka käynnistäisi ohjelman, kuten usein muissa sovelluksissa on tapana. Sovellus voi sen sijaan käynnistyä vastaanottamansa aikeen johdosta monen eri komponentin kautta. Lisäksi sovellus saatetaan joutua käynnistämään ja sulkemaan useita kertoja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

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

2.4 Aktiviteetit

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

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



Kuva 2: Aktiviteetin elinkaari

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

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

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

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

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

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

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

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

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

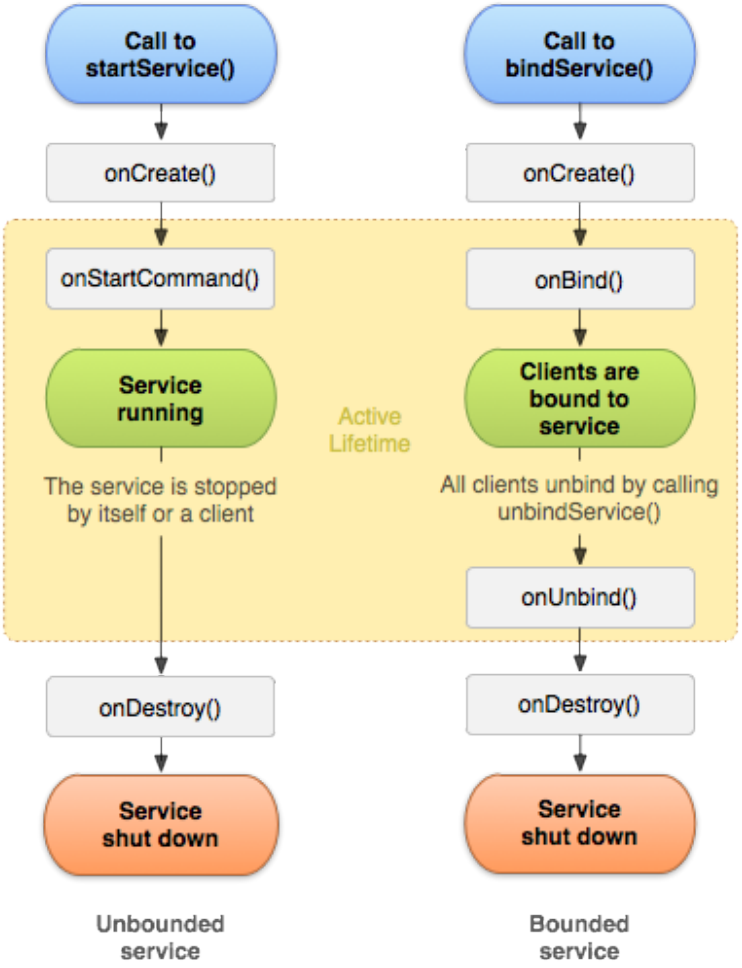
Palat

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

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

2.5 Palvelut

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



Kuva 3: Palvelun elinkaari

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

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

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

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

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

2.6 Sisällöntarjoajat

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

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

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

2.7 Aikeet

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

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

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

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

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

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

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

2.8 Android manifest

Jokaisella Android-sovelluksella on AndroidManifest.xml-tiedosto, joka sisältää järjestelmälle välttämätöntä tietoa sovelluksen ajamiseksi. Manifestissa määritellään muunmuassa sovelluksen javapaketti, joka toimii samalla sovelluksen uniikkina tunnisteen, määritellään sovelluksen komponentit, eli aktiviteetit, palvelut, sisällöntarjoajat ja lähetysten vastaanottajat, joista sovellus koostuu, sekä niiden toiminnallisuus ulkopuolelta tulevien aikeiden kannalta, mitä oikeuksia sovellus tarvitsee toimiakseen, mitä oikeuksia toisilla sovelluksilla pitää olla, jotta ne voivat käyttää kyseisen sovelluksen palveluita, mikä on sovelluksen vaatima Android API:n minimiversio sekä mitä kirjastoja sovellus tarvitsee toimiakseen. [andb]

3 Android-sovellusten testaamisen perusteet

Pitäiskö tässä luvussa olla jotain ylipäänsä testaamisen perusteita, vai riittääkö että johdannossa sivutaan aihetta?

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]

Yksikkötestauksella (unit testing) tarkoitetaan testejä, joiden kohteena on pienimmät mahdolliset ohjelmakomponentit. Olio-ohjelmoinnissa tämä tarkoittaa useimmiten yksittäistä luokkaa, koska yksittäiset metodit vaikuttavat olion tilaan ja siten metodien toimintaan. [PY08, 282-286]. JUnit [jun] on Javan standardi yksikkötestaustyökalu.

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

Toiminnallisessa testauksessa (functional testing) testattavan ohjelman sisäistä rakennetta ei tunneta vaan ollaan kiinnostuttu vain syötteistä ja niitä vastaavista tulosteista. Toiminnallista testausta voidaan kutsua myös musta laatikko -testaukseksi (black box testing) (varmista jostain toisesta lähteestä että nämä ovat oikeasti synonyymi, wikipedian mukaan ei ole vaan functional on black boxin alalaji!) Toiminnallisessa testauksessa ollaan kiinnostuttu ohjelman käyttäjälle näkyvästä toiminnasta ja niitä voidaan tehdä esimerkiksi ohjelman määrittelyn pohjalta. [PY08, 161-162]

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

Järjestelmätestauksessa (system testing) testataan koko järjestelmää, se perustuu ohjelman havaittavaan toimintaan ja se on itsenäinen suhteessa ohjelman toteutukseen. Jos ohjelma läpäisee järjestelmätestit, sen voi olettaa olevan vapaa tunnetuista bugeista. [PY08, 418-421]

Hyväksyntätestien (acceptance tests) tarkoitus on kertoa, onko ohjelma valmist julkaistavaksi. Hyväksyntätesteissä ei etsitä vikoja ohjelmasta, vaan yritetään varmistaa sen riittävä laatutaso. Hyväksyntätestit ovat usein tilastollisia: niissä mitataan ohjelman luotettavuutta, saavutettavuutta tai häiriötiheyttä. Ongelmana tällaisessa testauksessa on, että vaatii hyvin paljon testiainestoa ennen kuin voidaan olla varmoja ohjelman riittävästä laadusta. Usein hyväksyntätestauksessa käytetään alfa- ja beta-testausta, jolloin ohjelman käyttäjät pääsevät testaamaan ohjelmaa ja raportoimaan sen laadusta. [PY08, 421-423]

Mockaus (suomennos?) on tekniikka, joka helpottaa yksikkötestien kirjoittamista. Yksikkötestien ulkoiset riippuvuudet voidaan korvata testeissä kontrolloitavilla mock-olioilla. Tällöin testit ovat vakaampia, koska ulkopuolisten komponenttien muokkaus ei vaikuta testeihin, testin haluttuun lopputulokseen vaikuttava ympäristö on helppo saada haluttuun muotoon. Tällöin testien on helppo testata myös sellaisia olosuhteita, jotka ovat harvinaisia, tai vaikea saada muokattua. Lisäksi mockaamalla voidaan korvata vielä tekemättömät ulkoiset riippuvuudet mock-toteutuksilla. [MFC01]

3.2 Mikä on hyvä testityökalu?

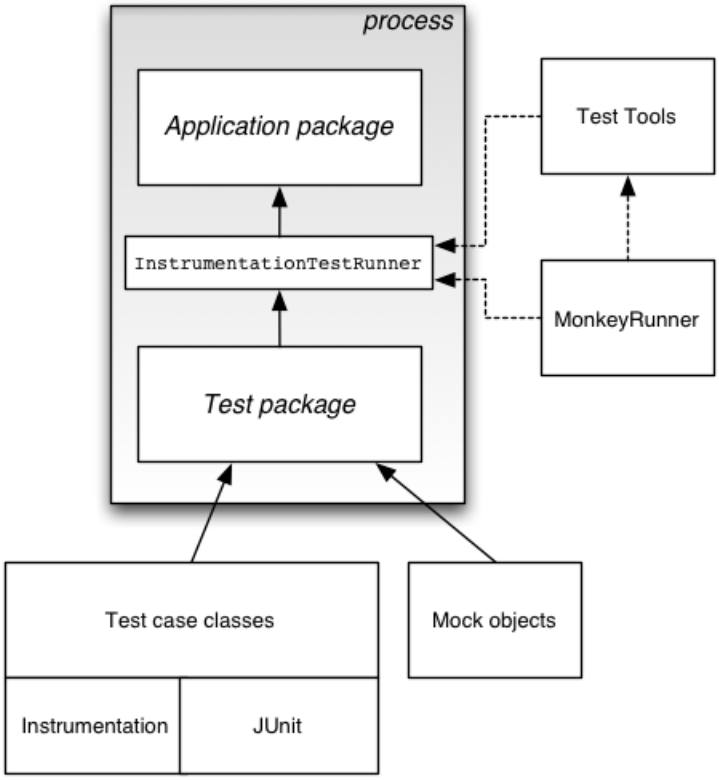
Androidille on tehty Androidin mukana tulevien testaustyökalujen lisäksi monia muita testaustyökaluja. Nämä työkalut erottautuvat Androidin työkaluista jollain tavalla.

Testaustyökalujen arviointia käsittelevät muunmuassa [PS92]

3.3 Android SDK:n mukana tulevia testityökaluja

Androidin SDK:n (suomennos!) mukana tulee monia testaustyökaluja. Testejä voi ajaa joko emulaattorilla tai suoraan puhelimessa.

Androidin testisarjat(suite?) perustuvat JUnit-työkaluun. 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 test case (suomennos?) luokat. Nämä luokat tar-



Kuva 4: placeholder omalle kuvalle

joavat apumetodeita mock-olioiden(suomennos?) luomiseen ja komponentin elinkaaren hallintaan. Androidin JUnit-toteutus mahdollistaa JUnitin versio 3:n mukaisen testityylin, ei uudempaa versio 4:n mukaista.

Kuvassa 4 on esitetty Androidin testien ajoympäristö. Testattavaa sovellusta testataan ajamalla testipaketissa olevat testitapaukset MonkeyRunnerilla (ks. luku 3.5). Testipaketti sisältää testitapausten lisäksi Androidin instrumentaatiota, eli apuvälineitä sovelluksen elinkaaren hallintaan ja koukkuja, joilla järjestelmän lähettämiä callback-metodikutsuja pääsee muokkaamaan, sekä mahdollisesti mock-olioita korvaamaan järjestelmän oikeita luokkia testin ajaksi mock-toteutuksella. [andb]

3.4 Komponenttikohtaiset testiluokat

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

Aktiviteettien testauksessa Androidin JUnit-laajennus on tärkeä, koska aktiviteeteil-

la on monimutkainen elinkaari, joka perustuu paljolti callback-metodeihin (suomennos), joiden suora kutsuminen ei ole mahdollista. Aktiviteettien testauksen pääyläluokka on InstrumentationTestCase. Sen avulla on mahdollista käynnistää, pysäyttää ja tuhota testattavana oleva aktiviteetti halutuissa kohdissa. Lisäksi sen avulla voi mockata järjestelmäolioita, kuten Contexteja ja Applicationseja. Tämä mahdollistaa testin eristämisen muusta järjestelmästä ja Intentien luomisen testiä varten. Lisäksi yliluokassa on metodit käyttäjäinteraktion, kuten kosketus- ja näppäimistötapahtumien lähettämiseen suoraan testattavalle luokalle.

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

Palveluiden testaaminen on paljon yksinkertaisempaa kuin aktiviteettien. Ne toimivat eristyksessä muusta järjestelmästä, joten testattaessakaan ei tarvita Androidin instrumentaatiota. Android tarjoaa ServiceTestCase-yliluokan palveluiden testaamiseen. Se tarjoaa mock-oliot Application- ja Context-luokille, joten palvelun saa testattua eristettynä muusta järjestelmästä. Testiluokka käynnistää testattavan palvelun vasta kutsuttaessa sen startService() tai bindService()-metodia, jolloin mock-oliot voi alustaa ennen palvelun käynnistymistä. Mock-olioiden käyttö palveluiden testaamisessa paljastaa myös mahdolliset huomaamatta jääneet riippuvuudet muuhun järjestelmään, koska mock-oliot heittävät poikkeuksen, mikäli niihin tulee metodikutsu, johon ei ole varauduttu.

Sisällöntarjoajien testaaminen on erityisen tärkeää, jos sovellus tarjoaa sisällöntarjoajiaan muiden sovellusten käyttöön. Tällöin on myös olennaista testata niitä käyttäen samaa julkista rajapintaa, jota muut sovellukset joutuvat käyttämään kommunikoidessaan sisällöntarjoajien kanssa. Sisällöntarjoajien testauksen yliluokka on ProviderTestCase2, joka tarjoaa käyttöön mock-oliot ContentResolveriosta ja Contextista, jolloin sisällöntarjoajia voi testaja eristyksissä muusta sovelluksesta. Yliluokka tarjoaa myös metodit sovelluksen oikeuksien testaamisen. Contextin mock-olio mahdollistaa tiedosto- ja tietokantaoperaatiot, mutta muut Androidin kirjas-

tokutsut on toteutettu stubeina. Lisäksi tiedon kirjoitusosoite on uniikki testissä, joten testien ajaminen ei yliaja varsinaista sovelluksen tallentamaa tietoa. Sisällöntarjoajatestit ajetaan emulaattorissa tai Android-laitteella. [andb]

3.5 MonkeyRunner

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

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

3.6 Monkey

Monkey on Androidin mukana tuleva työkalu, jota voi ajaa emulaattorissa tai Android-laitteessa ja joka tuottaa pseudosatunnaisia syötteitä ohjelmalle, kuten painalluksia, eleitä sekä järjestelmätason viestejä. Monkeytä voi käyttää esimerkiksi sovelluksen stressitestaukseen tai fuzz-testaukseen.

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

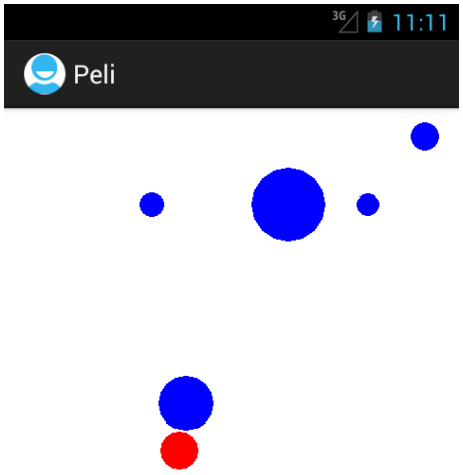
Monkey pystäyttää testin, jos ohjelmasta lentää käsittelemätön poikkeus tai jos järjestelmä lähettää sovellus ei vastaa -virheviestin. Näissä tapauksissa Monkey antaa raportin virheestä ja miten se syntyi. Monkey voi myös haluttaessa tehdä profilointiraportin testistä.[andb]

3.7 Androidin testaustyökalujen arviointia

Mikä on hyvällä mallilla, mihin tarvitaan lisätyökaluja

4 Testiprojektista

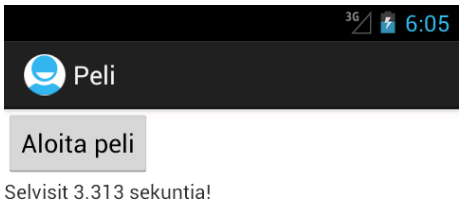
4.1 Yksikkötestidemo



Kuva 5: Ruutukaappaus pelist

Yksikkötestaustyökaluja testasin itse tehdyllä demoprojektilla. Kyseessä on yksinkertainen peli, jonka pelinäkymästä on ruutukaappaus kuvassa 5. Pelissä ohjataan kosketusnäytöllä painamalla punaista palloa ja pyritään väistämään ympäriinsä pomppivia sinisiä palloja. Kun peli päättyy, palataan takaisin päänäkymään, josta on ruutukaappaus kuvassa 6. Tästä näkymästä 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 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 7.



Kuva 6: Ruutukaappaus pelist

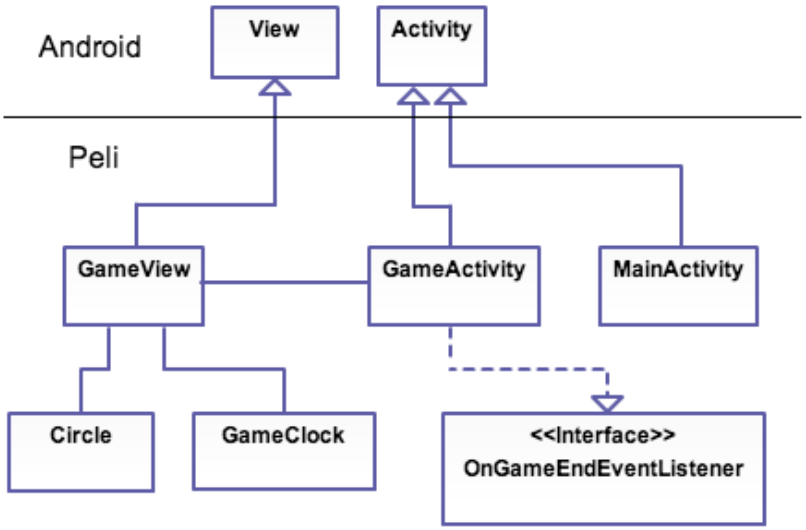
4.2 Tomdroidin toiminnallinen testaus

Testityökalujen vertailussa käytän testattavana ohjelmana Tomdroidia. Tomdroid on GPL-lisenssillä julkaistu avoimen lähdekoodin sovellus, joka toimii muistikirjasovelluksena, joka synkronisoi sisältönsä automaattisesti. [tomb] Tomdroid on valittu testattavaksi sovellukseksi, koska sen lähdekoodi on saatavilla, se on riittävän monimutkainen, jotta sille tehdyt testit kuvaisivat oikeassa android-kehityksessä kohdattavia testaushaasteita, se on vasta beta-vaiheessa, joten sovelluksesta pitäisi löytyä myös bugeja ja lisäksi sovelluksen oma testaus on lähes olematonta.

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

Tomdroidissa olennaisimmat näkymät ovat muistikirjalista, josta on ruutukaappaus kuvassa 8, yksittäisen muistikirjan selaaminen (kuvassa 9) ja sen editointi (kuvassa 10).

Listanäkymässä näkyvät kaikki käyttäjän muistikirjat päivitysajan mukaan järjestettynä. Muistikirjan koskeminen avaa kyseisen muistikirjan selausnäkymän. Ylä-

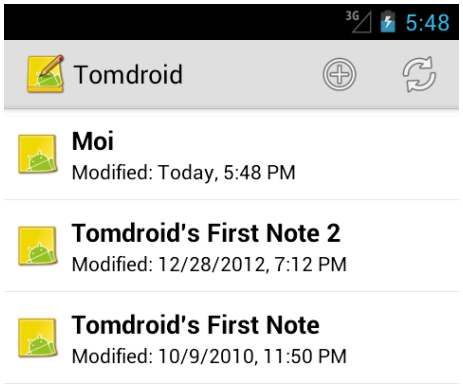


Kuva 7: Pelin luokkakaavio

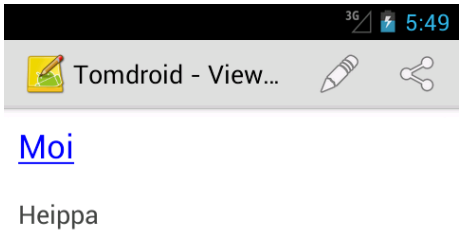
palkin +-symboli luo uuden muistikirjan ja avaa sen editointinäkymään. Yläpalkin oikeassa reunassa on synkronointi-symboli, josta muistikirjojen tila päivitetään palvelimen kanssa.

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.

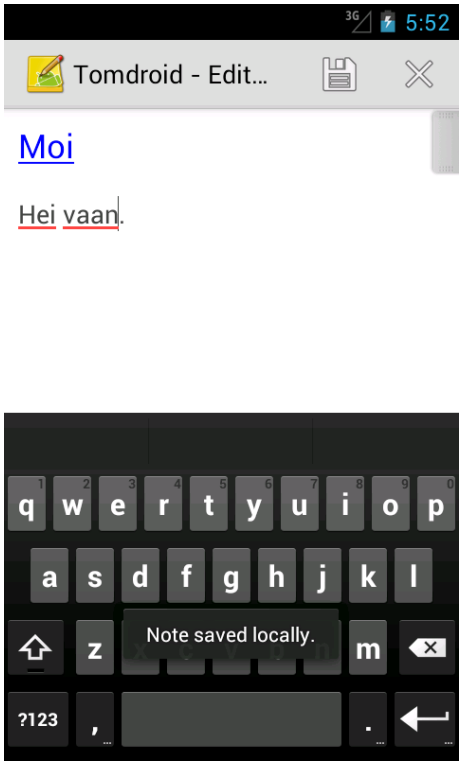
Editointinäkymässä yläpalkissa on kuvakkeet muutosten tallentamista ja perumista varten. Tallennettaessa ruudulla näkyy hetken aikaa leijuke, jossa kerrotaan muutosten tallennuksesta. Painettaessa peru-nappia ruudulle tulee dialogi, jossa pitää vahvistaa peruminen edelliseen tallennettuun versioon. Lisäksi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



Kuva 8: Muistikirjalista Tomdroidissa



Kuva 9: Muistikirjan luku Tomdroidissa



Kuva 10: Muistikirjan editointi Tomdroidissa

5 Yksikkötestityökaluja

Androidin oma yksikkötestitapa on tehdä JUnit-testejä Androidin oman AndroidUnitTestCase-luokan aliluokkana. Nämä testit ajetaan emulaattorissa Dalvik-ympäristössä. Tässä tavassa on kaksi heikkoutta, jonka takia on olemassa myös kolmansien osapuolien yksikkötestityökaluja Android-ympäristöön. Ensinnäkin testien ajaminen emulaattorissa on hitaampaa kuin jos niitä voisi ajaa suoraan standardissa Javan virtuaalikoneessa JUnit-testeinä. Toiseksi muutkaan testauksen apuna käytettävät työkalut, kuten mock-kehykset, eivät välttämättä toimi ongelmitta Dalvik-ympäristössä. Tässä luvussa vertailen Android-sovelluksen yksikkötestausta AndroidUnitTestCasella ja suosituimmalla JVM-vaihtoehdolla: Robolectricillä.

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

Tavoitteista: helppokäyttöinen,nopea, toimii mock-kehysten kanssa, lue lisää lähteistä

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 IDE:stä ajoaikaisen poikkeuksen, mutta Robolectric korvaa nämä luokat varjototeutuksilla, jotka palauttavat poikkeuksen sijaan tyhjän oletusvastauksen (kuten null, 0 tai false) tai jos Robolectricissa on kyseistä metodia varten olemassa varjototeutus, se palauttaa toteutuksen määrittelemän oikeamman paluuarvon.

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

5.2 Aiempaa tutkimusta

Sadeh et al. vertasivat Androidin aktiviteettien yksikkötestausta JUnitilla, AIT:lla (lyhenne/suomennos?) ja Robolectricilla. JUnitilla testattaessa ongelmaksi muodostui se, että ohjelmakoodia jouduttiin melko rajusti refaktoroimaan(suomennos?), 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 AIT:lla. AIT:hen verrattuna Robolectricin vahvuudet oli virhepaikkojen paikantamisen helppous ja nopea ajonopeus. Robolectric-testit ajautuivat viisi kertaa nopeammin kuin AIT:lla ajetut testit, koska AIT ajaa testit emulaattorilla Dalvikilla, Robolectric taas suoraan Javalla. AIT:n suurin vahvuus oli testien kirjoittamisen helppous.[S+11]

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

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

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

Listing 1: CustomRobolectricTestRunner

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

5.3 Robolectricin asentaminen

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

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

5.4 Perusominaisuudet

Aktiviteetin elinkaarimetodien testaus on olennaisin osa android-sovellusten yksikkötestauksesta. Testeissä testataan koodia, joka löytyy liitteestä x.

Listauksessa 2 on yksinkertainen aktiviteettiyksikkötesti Robolectricillä toteutettu-

Listing 2: Robolectric Activity test

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

  private GameActivity activity;

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

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

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

}
```

na. Robolectricin testit ovat yhteensopivia JUnitin version 4 kanssa, mikä mahdollistaa annotaatioiden käytön testeissä. @RunWith-annotaatiolla määritellään testien ajossa käytettävä testiajaja (runner), tässä tapauksessa käytetään luvussa 5.3 esiteltyä ajajaa. @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 automaattisesti havaitsee Androidin metodien kutsun ja korvaa ne robolectricin toteutuksella, jotta ne toimivat testissä. Tämän takia aktiviteetti voidaan alustaa suoraan konstruktorilla ja kutsua sen onCreate()-metodia toisin kuin AndroidUnitTestissä, jossa aktiviteetti käynnistetään AndroidUnitTestCase-luokan apumetodien avulla.

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

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

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

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

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

Robolectricistä poiketen Androidin yksikkötestit ovat JUnit3-pohjaisia, joten annotaatioita ei käytetä, vaan metodien nimien perusteella päätellään setUp()-metodi sekä testimetodit siitä, että niiden nimi alkaa sanalla test. Koska setUp()-metodeita on vain yksi, on setUp()-metodin alussa kutsuttava yliluokan setUp()-metodia tai

Listing 3: ActivityUnitTestCase

```
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 testCreatedActivityInitializesGameViewWithRunningState()
    GameView gameView = (GameView) activity.findViewById(R.id.gameview)
    assertThat(gameView.getState(), equalTo(GameState.RUNNING));
  }

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

}
```

Listing 4: Robolectric Shadow objects

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

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

yliluokan suorittamat alustukset jäävät tekemättä.

Listauksessa 4 on hieman monimutkaisempi testitapaus samasta testiluokasta kuin listaus 2. Tässä testissä testataan että pelin loppua. Jos aktiviteetti toimii oikein, se rekisteröi GameView-oliolle observer-patternin (suomennos?) mukaisen tarkkailijan, jota kutsutaan, kun peli päättyy. Aktiviteetin pitäisi tämän jälkeen pyytää GameView-oliolta pistemäärä ja lähettää uusi aie, jolla käynnistetään MainActivity-aktiviteetti ja annetaan ylimääräisenä tietona saatu pistemäärä.

Androidin Activity-luokka ei tarjoa suoraa julkista metodia, jolla voitaisiin kysyä, minkä aikeen aktiviteetti on lähettänyt. Tällaisia tapauksia varten Robolectric tarjoaa varjo-olioita, jotka kopioivat oikean android-luokan tilan ja tarjoaa laajemman apin näiden olioiden tilaan. Tätä varten testissä tehdään varjo-olio testattavasta aktiviteetista ja 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 pisteet, tässä tapauksessa 0.0, koska kello ei ole ollut testissä käytössä.

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 ta-

Listing 5: ActivityUnitTestCase intents

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

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

valla kuin Robolectricin varjo-oliolta kysyttiin edellisessä listauksessa. Tämän jälkeen testin läpipääsyn varmistus tapahtuu täsmälleen samalla tavalla kuin Robolectric-testissä.

5.5 Toiminta mock-kehysten kanssa

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

Luvussa 5.4 esitetyt yksikkötestit ovat riippuvaisia GameView-luokan toteutuksesta. Assertit voivat mennä väärin, 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 mock-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

Listing 6: Mock Subclass

```
private class MockGameActivity extends GameActivity {

        private GameView gameView;

  public MockGameActivity(GameView gameView) {
          this.gameView = gameView;
  }

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

olla esimerkiksi mock-näkymä.

Tämän voisi robolectricillä tehdä vaihtoehtoisesti siten, että toteuttaa oman Shadow-luokan Activity-luokasta, joka on kaikkien aktiviteettien yliluokka. Toteutuksen voi tehdä siten, että käyttää Robolectricin oletustoteutusta kaikkeen muuhun paitsi findViewById()-metodin toteutukseen [roba]. Mock-näkymän injektointia varten tälle voisi tehdä myös oman set-metodin, jolla mock-näkymä sijoitettaisiin palautettavaksi. Yllä esitetty testattavan luokan aliluokka on kuitenkin helpompi tapa toteuttaa sama asia, koska robolectricille ei tarvitse erikseen sanoa, että käyttää shadow-luokkana omaa toteutusta. Lisäksi on helpompi vaihdella, käytetäänkö testeissä omaa aliluokkaa, vai varsinaista testattavaa luokkaa. (tarkista vielä, että kuvattu tapa toimii käytännössäkin..)

Listauksessa 7 on esitetty edellisen listauksen aliluokkaa käyttävä robolectric-testi. Ensimmäisellä rivillä luodaan Mockiton mock-olio GameView-luokasta. Toisella rivillä luodaan testattavan aktiviteetin aliluokka, jolle syötetään mock-näkymä konstruktoriparametrina. Sitten kutsutaan onCreate-metodia, kuten listauksen 2 vastaavassa testissä. Assertin sijaan testin läpäisy testataan Mockiton verify-metodilla, joka varmistaa, että parametrina annettua mock-olion annettua metodia kutsuttiin annetulla parametrillä.

AndroidUnitTestin avulla tehty vastaava testi listauksessa 8 on mock-näkymän luo-

Listing 7: Mock-testi robolectrcicillä

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

Listing 8: Mock-testi androidunittestillä

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

misen ja testin läpäisyn varmistamisen kannalta täsmälleen samanlainen kuin robolectric-testi. Testissä käytetään apuna vastaavaa aliluokkaa kuin Robolectric-testissäkin ja aktiviteetin käynnistys tapahtuu samoin kuin listauksen 3 testissä.

5.6 TDD-syklin 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 esimerksi ci-palvelimella (onko relevanttia mobiilissa, oisko parempia esimerkkejä?) voi kestää hyvin pitkään, mikä saattaa hidastaa koodin jakamista tai koodausprosessia, kun muutosten varmistus kestää pitkään.

Toinen, ehkä olennaisempi, tekijä on se, että yksikkötestausta tehdään nykyään usein Test Driven Development -prosessin mukaisesti siten, että ensin tehdään testi, joka testaa vielä olematonta koodia ja tämän jälkeen koodia muutetaan niin, että testi saadaan menemään läpi. Tällöin testejä ajetaan jatkuvasti ja ohjelmointiprosessi pysähtyy aina, kun odotetaan testien ajautumista ennen kuin päästään jatkamaan TDD-sykliä seuraavaan vaiheeseen. (Lähde TDD:lle, jne) Jos testien, edes yksittäisen testiluokan, ajaminen on kovin hidasta, tämä tekee TDD-tyylisestä ohjelmistokehityksestä liian hidasta, jotta se olisi käytännöllistä.

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

Taulukko 1: Testikestot

Testasin ensin isomman testisetin ajonopeutta. Kopioin aiemmin luvussa esitellyn testiluokan mock-testin kanssa 32 erilliseksi testiluokaksi niin, että testimetodeita kertyi yhteensä 128. Ajoin kaikki testit yhtenä sarjana ja annoin Eclipsen ottaa aikaa testien suorituksesta. Tähän aikaan ei sisälly sitä aikaa, joka kuluu JUnitin käynnistymiseen. (Tämä aika on pitempi AUT:lle, mittaa). 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 reaalimaailman tapauksessa. Robolectricin lupaus nopeammista yksikkötesteistä vaikuttaa siis toteutuvan.

Testisarjan lisäksi tutkin yksittäisten testien kestoja. Erityisen kauan emulaattorissa ajetuista testeistä kesti ensimmäisen testiluokan mock-testi. Sen ajo kesti jokaisella ajokerralla yli 12 sekuntia, eli yli 25% koko testisarjan kestosta. Tämä johtunee 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 hyvin hidas, noin 0.5 sekuntia. Robolectric toimii Mockiton tavoin tavukooditasolla ja instrumentoi jotain roskaa.. katso lähteistä.

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.

Aika ensimmäiseen testitulokseen on merkittävä ohjelmoidessa Test Driven Development (tdd) -filosofian mukaan. Tdd:ssä on kaksi merkittävää periaatetta: ohjelmakoodia saa kirjoittaa vain automaattisen testin korjaamiseksi ja duplikaattikoodin poistamiseksi. Näistä periaatteista seuraa tunnettu tdd-sykli: punainen, vihreä, refaktoroi. Ensin kirjoitetaan testi, joka ei mene läpi, koska testin toteuttavaa ohjelmakoodia ei ole vielä olemassa. Vaiheen nimi on punainen, koska useimmilla yksikkötestityökaluilla lopputuloksena näkyy punainen palkki, jos jokin testi ei mene

läpi. Toinen vaihe on kirjoittaa juuri sen verran koodia, mitä tarvitaan testin läpäisemiseksi. Tässä vaiheessa ei välitetä miten rumaa koodi on. Vaiheen nimi on vihreä, koska useimmillaa yksikkötestityökaluilla lopputuloksena näkyy vihreä palkki, kun testit menevät läpi. Viimeisessä vaiheessa refaktoroidaan toisessa vaiheessa mahdollisesti syntynyt duplikaattikoodi. Refaktorointivaiheessa kirjoitetut testit auttavat varmistamaan, ettei refaktoroidessa hajoiteta vanhaa toiminnallisuutta. Jotta tällainen ohjelmointisykli on mahdollinen, yksi vaatimus on, että ohjelmointiympäristö tarjoaa nopean palautteen testeistä [Bec03].

5.7 Mitä jäi kokeilematta, puuttuvia featureja?

Onko tällaisia? Entä muiden kuin aktiviteettien testaus robolectricilla. Mitä muuta on vielä käsittelemättä?

5.8 Analyysi

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

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

Suurin ero testityökalujen välillä oli testien suoritusnopeudella. Robolectric lupaa nopeita testejä ja toteuttaa lupauksensa; Robolectric-testit ajautuivat yli 20 kertaa nopeammin kuin emulaattorissa ajetut AndroidUnitTestit. Lisäksi AndroidUnitTestillä aika ensimmäiseen testitulokseen pienelläkin testiohjelmalla oli yli puoli minuuttia, joten tdd-filosofian mukainen ohjelmointi AndroidUnitTestCasea käyttäen

on toivottoman hidasta.

6 Toiminnallinen testaus

Robotium vs. monkeyrunner

6.1 Robotium

Robotium ei tule Androidin sdk:n mukana, mutta se on paljon käytetty testityökalu Android-sovellusten testauksessa. Robotiumin slogan on, että se on kuin Selenium, mutta Androidille. Selenium taas on laajasti integraatio- ja funktionaalisessa testauksessa käytetty työkalu, joka mahdollistaa selaimen toimintojen automatisoimisen, kuten linkkien klikkauksen, lomakekenttien täyttämisen jne. [sel]

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

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

Robotiumin sivuilla sille esitellään useita vahvuuksia Android SDK:n mukana tuleviin työkaluihin verrattuna. Testit vaativat vain vähäistä tuntemusta testattavasta sovelluksesta, Robotium tukee usean aktiviteetin testaamiseta samassa testissä, testien kirjoittamisen nopeus, testikoodin selkeys ja sitkeys, joka johtuu ajoaikaisesta sidonnasta käyttöliittymäkomponentteihin, nopea suoritusnopeus ja helppo integrointi jatkuvan integroinnin työkaluihin Antin tai Mavenin avulla. (pitäisikö ant/maven esitellä jossain?) [robb]

6.2 Troyd

Troyd on Robotiumia käyttäen tehty integraatiotestaustyökalu, jonka tavoite on yhdistää Monkeyn skriptausominaisuudet ja Robotiumin tarjoama korkean tason API. Troyd-testit käyttävät korkean tason komentoja, kuten paina nappia nimeltä x, tarkista, että ruudulla näkyy teksti y, jne, joten testien kirjoituksen pitäisi olla nopeaa. Lisäksi Troyd tarjoaa nauhoitus-toiminnon, jolla testiä voidaan kirjoittaa siten, että testiä kirjoittaessa ohjelma etenee aina seuraavaan tilaan testin mukaisesti. Lopuksi

testi tallentuu testitapauksiksi. [JF12]

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

Troydin heikkouksia on Jeonin ja Fosterin 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 oikeassa tilassa ennen testin jatkamista. [JF12]

6.3 TEMA

Pleh.

6.4 Aiempaa tutkimusta

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

6.5 Asennus

Robotiumin asennus on helppoa, katso listaus 9

6.6 Robotium-testit

Robotium-testi, jossa testataan uuden muistikirjan luonti, on esitetty listauksessa 10. Robotiumilla testiä ohjataan Solo-luokan instanssin kautta, jossa on sovelluksen

Listing 9: Robotium testirunko

```
public class RobotiumTest extends ActivityInstrumentationTestCase2<Tom

        private Solo solo;

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

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

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

Listing 10: Muistikirjan luontitesti robotiumilla

```
public void testCreateNoteAddsNote() {
        solo.assertCurrentActivity("Testi␣alkoi␣v   r st ␣aktiviteet
        assertFalse(solo.searchText("new␣note"));
        solo.clickOnActionBarItem(R.id.menuNew);
        solo.assertCurrentActivity("Uuden␣muistikirjan␣luonti␣ei␣avannu
        solo.enterText(0, "new␣note");
        solo.clickOnActionBarItem(R.id.edit_note_save);
        solo.clickOnActionBarHomeButton();
        solo.assertCurrentActivity("Koti-n pp imen␣painaminen␣ei␣vien
        assertTrue(solo.searchText("new␣note"));
}
```

kanssa kommunikointiin tarkoitettuja metodeja, sovelluksen tilasta kertovia metodeja, sekä assertteja. Testin ensimmäisellä rivillä käytetään assertCurrentActivity()-metodia asserttia varmistamaan, että testi alkaa muistikirjalistasta. Toisella rivillä varmistetaan, että testissä luotavaa muistikirjaa ei vielä löydy listasta. Ilman tätä testissä ei voisi olla varma, että muistikirja on luotu onnistuneesti juuri testin aikana. Seuraavalla rivillä painetaan yläpalkin uuden muistikirjan luovaa nappia clickOnActionBarItem()-metodilla. Se ottaa parametrina komponentin id:n, johon ollaan painamassa. Tämän jälkeen pitäisi avautua uusi muistikirja editointinäkymään, mikä varmistetaan seuraavalla rivillä. Sitten syötetään enterText()-metodilla uuden muistikirjan otsikoksi ïew note:. Ensimmäinen parametri kertoo, monenteenko ruudulla näkyvään tekstinmuokkauskomponenttiin teksti syötetään. Tämän jälkeen klikataan yläpalkin tallennus-nappia ja sitten muistikirjalistaukseen vievää nappia. Lopuksi vielä varmistetaan, että palattiin takaisin muistikirjalistaan ja listasta löytyy nyt juuri luotu aktiviteetti.

6.7 Analyysi

Se.

7 yksityisyys/turvallisuus/tms

monkey vs ??

7.1 Androidin tietoturvaratkaisuista

Android-sovelluksia ajetaan Javan virtuaalikoneessa, joka pyörii väliohjelmistona (middleware) linuxin ytimen päällä. Jokainen Android-sovellus on oma käyttäjänsä Android-järjestelmässä. Sovellusten välinen kommunikointi on rajattu Androidin oman API:n kautta toimivaksi aikeilla, joten sovellukset eivät pääse suoraan käsiksi toistensa koodiin. Tämä tarkoittaa myös sitä, että jos jostain sovelluksesta löytyy haavoittuvuus, sen kautta ei pääse käsiksi muuhun järjestelmään.

Android-sovellukset määrittelevät xml-manifestissaan (suomennos?) permission labels (suomennos?), joiden perusteella sovellustenvälinen kommunikaatio sallitaan tai estetään. Kutsuvasta sovelluksesta pitää löytyä vastaanottavan pään permisison label. (...) Sovelluksen komponentteja voi myös määritellä yksityisiksi (private), jolloin muista sovelluksista ei pääse niihin mitenkään käsiksi. Julkiset komponentit täytyy erikseen määritellä xml-manifestissa (joten vähemmän hyökättävää).

[OM09]

7.2 Fuzzing

Sovellusten turvallisuusominaisuuksien testaaminen on vaikeaa, koska testitapa poikkeaa tavanomaisesta. Yleensä testeillä halutaan varmistaa, että sovelluksessa on jokin ominaisuus kun taas turvallisuustestaus on negatiivista testaamista: halutaan varmistaa, että sovelluksessa ei ole turvallisuusaukkoja tai muita ei-haluttuja ominaisuuksia. Tällöin on mahdotonta kirjoittaa testitapauksia, koska ei voida mitenkään testata kaikkia mahdollisia sovelluksen suorituspolkuja.[MEK+12]

Fuzzing

8 Fragmentaatio

Android-alustan suurimpia haasteita ohjelmistokehityksen ja testaamisen kannalta on fragmentaatio.

Appthwack[app]

8.1 Androidin fragmentaatio

Google tarjoaa android-laitteiden fragmentaation seurantaan palvelua, jossa kerrotaan kahden viikon jaksolla Androidin sovelluskaupassa käyneiden laitteiden jakauma androidin version, näytön koon ja resoluution sekä OpenGL:n version mukaan.[andc] Jakauma ei välttämättä vastaa käytössä olevien Android-laitteiden jakaumaa, mutta toisaalta sovelluskehittäjän kannalta ne käyttäjät, jotka käyttävät sovelluskauppaa, lienevät olennaisimpia.

Versio	Koodinimi	API	Osuus
1.5	Cupcake	3	0.1%
1.6	Donut	4	0.4%
2.1	Eclair	7	3.4%
2.2	Froyo	8	12.9%
2.3 - 2.3.2	Gingerbread	9	0.3%
2.3.3 - 2.3.7	Gingerbread	10	55.5%
3.1	Honeycomb	12	0.4%
3.2	Honeycomb	13	1.5%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	23.7%
4.1	Jelly Bean	16	1.8%

Taulukko 3: Androidin versioiden osuus 1.10.2012 päättyneellä 2-viikkoisjaksolla

Taulukko 3 kuvaa androidin käyttöjärjestelmäversioiden jakaumaa. Miksi olennainen? Androidin 4-versio julkaistiin lokakuussa 2011, mutta vuotta myöhemmin vain noin neljäsosa laitteista käyttää 4. versiota. Jos sovelluskehittäjä haluaa tukea esimerkiksi 90% markkinoilla olevista laitteista, on tuki ulotettava 2.2-versioon, joka julkaistiin toukokuussa 2010. [andd]

Android-laitteet poikkeavat toisistaan sekä näytön fyysisen koon, että pikselitiheyden puolesta. Taulukossa 4 on kuvattuna erilaisten näyttötyyppien jakaumaa. Näytön koon arvioinnissa android käyttää tiheysnormalisoituja pikseleitä (Density-

	ldpi (120dpi)	mdpi (160dpi)	hdpi (240dpi)	xhdpi (320dpi)
small	1.7%		1.0%	
normal	0.4%	11%	50.1%	25.1%
large	0.1%	2.4%		3.6%
xlarge		4.6%		

Taulukko 4: Android-laitteiden näyttökokojen ja pikselitiheyden osuudet 1.10.2012 päättyneellä 7 päivän jaksolla.

independent pixel, dp), jotka lasketaan kaavalla px = dp * (dpi / 160), missä px on pikseli ja dpi on pikseleiden määrä tuumalla. Siten esimerkiksi 240 dpi:n näytöllä, yksi dp vastaa puoltatoista fyysistä pikseliä. Sovellusten ulkoasu tulisi aina suunnitella käyttäen dp:tä yksikkönä, jolloin skaalaus eri kokoisille ja pikselitiheyksisille näytöille onnistuu parhaiten. Taulukossa 4 näyttökoot on lajiteltu niin, että xlarge näyttöjen resoluutio on vähintään 960dp x 720dp, large: 640dp x 480dp, normal: 470dp x 320dp ja small vähintään 426dp x 320dp.

OpenGL ES versio	jakauma
1.1	9.2%
2.0 & 1.1	90.8%

Taulukko 5: OpenGL versiot

Taulukossa 5 on kuvattu OpenGL ES -versioiden jakauma Android-laitteissa.[andc]

Oheisten muuttujien lisäksi Android-laitteet poikkeavat toisistaan myös monilla muilla tavoin. Suoritintehoa laitteissa on hyvin eri määrissä käytössä ja näytönohjainten tehotkin vaihtelevat. Lisäksi erilaisia lisälaitteita, kuten gps, kiihtyvyysantureita, kompasseja yms. saattaa olla laitteissa, tai olla olematta.

8.2 Mitä keinoja android sdk tarjoaa fragmentaation hallintaan

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 suomennos?) määritellään Androidin APIn mini-

miversio. 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. Jos sovellus on käyttökelpoinen myös ilman ominaisuutta, voi tämän hallintaan käyttää muita keinoja (ja ne oli..?) [anda]

9 Profiling

Nii..

10 Pohdintaa

Hmm..?

11 Yhteenveto

Yhteen veto.

Lähteet

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

anda *Android 4.1 Compatibility Definition.* http://static.googleusercontent.com/external_content/untrusted_dlcp/source.android 4.1-cdd.pdf.

andb Android developer documentation, http://developer.android.com/.

andc Android platform versions, http://developer.android.com/about/dashboards/index.ht

andd Android version history, http://en.wikipedia.org/wiki/Android_version_history.

app Appthwackin kotisivut, http://www.appthwack.com/.

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

HN11 Hu, C. ja Neamtiu, I., Automating gui testing for android applications. *Proceedings of the 6th International Workshop on Automation of Software Test*, 2011, sivut 77–83.

HP11 Ham, H. K. ja Park, Y. B., Mobile application compatibility test system design for android fragmentation. Teoksessa *Software Engineering, Business Continuity, and Education*, Kim, T.-h., Adeli, H., Kim, H.-k., Kang, H.-j., Kim, K. J., Kiumi, A. ja Kang, B.-H., toimittajat, Springer, 2011, sivut 314–320.

HSJ11 Hyungkeun, S., Seokmoon, R. ja Jin, H. K., An integrated test automation framework for testing on heterogeneous mobile platforms. *First ACIS International Symposium on Software and Network Engineering*, 2011, sivut 141–145.

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`.

jun Junitin kotisivut, http://www.junit.org/.

KM10 Kropp, M. ja Morales, P., Automated gui testing on the android platform. *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers*, October 2010, sivut 67–72.

mav Apache mavenin kotisivut, http://maven.apache.org/.

MEK+12 Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S. ja Stavrou, A., A whitebox approach for automated security testing of android applications on the cloud. *7th International Workshop on Automation of Software Test (AST)*, June 2012, sivut 22–28.

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

MMP+12 Mirzaei, N., Malek, S., Păsăreanu, C. S., Esfahani, N. ja Mahmood, R., Testing android apps through symbolic execution. *JPF Workshop*, 2012.

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

MYC+11 Ma, X., Yan, B., Chen, G., Zhang, C., Huang, K. ja Drury, J., A toolkit for usability testing of mobile applications. Teoksessa *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Zhang, J. Y., Wilkiewicz, J. ja Nahapetian, A., toimittajat, Springer, 2011, sivut 226–245.

OM09 Ongtang, M. ja McDaniel, P., Understanding android security. *IEEE Security Privacy*.

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

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

roba Robolectricin kotisivut, http://pivotal.github.com/robolectric/.

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

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

sel Seleniumin kotisivut, http://seleniumhq.org/.

Spa10 Spataru, A. C., Agile development methods for mobile applications. Pro gradu, University of Edinburgh, 2010.

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

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

TKH11 Takala, T., Katara, M. ja Harty, J., Experiences of system-level model-based gui testing of an android application. *IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, 2011, sivut 377–386.

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

tomb Tomdroidin kotisivut, https://code.launchpad.net/tomdroid.

Was10 Wasserman, A. I., Software engineering issues for mobile application development. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, sivut 397–400.

wik Android (operating system), http://en.wikipedia.org/wiki/Android_(operating_syste

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