



UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO – CHILE



ANÁLISIS DE HERRAMIENTAS PARA MEJORAR LA CALIDAD DE APLICACIONES PARA ANDROID

Memoria presentada como requerimiento parcial

para optar al título profesional de

INGENIERO CIVIL EN INFORMÁTICA

por

Cristopher Nicolás Oyarzún Altamirano

Comisión Evaluadora:

Cecilia Reyes (Guía, UTFSM)

Chihau Chau (Correferente, PUCV)

JUNIO 2014

Agradecimientos

Agradezco a Lorem ipsum ad his scripta blandit partiendo, eum fastidii accumsan euripidis in, eum liber hendrerit an. Qui ut wisi vocibus suscipiantur, quo dicit ridens inciderint id. Quo mundi lobortis reformidans eu, legimus senserit definiebas an eos. Eu sit tincidunt incorrupte definitionem, vis mutat affert percipit cu, eirmod consectetur signiferumque eu per. In usu latine equidem dolores. Quo no falli viris intellegam, ut fugit veritus placerat per.

Ius id vidit volumus mandamus, vide veritus democritum te nec, ei eos debet libris consulatu. No mei ferri graeco dicunt, ad cum veri accommodare. Sed at malis omnesque delicata, usu et iusto zzril meliore. Dicunt maiorum eloquentiam cum cu, sit summo dolor essent te. Ne quodsi nusquam legendos has, ea dicit voluptua eloquentiam pro, ad sit quas qualisque. Eos vocibus deserunt quaestio ei.

“Dedicado a mis padres, Guillermo y Alicia, que se han sacrificado por darme el mejor de los regalos, la educación”
– Christopher Oyarzún

Resumen

El crecimiento que ha tenido el sistema operativo Android en el mercado de las tecnologías móviles es considerable. Existe más de un millón de aplicaciones disponibles en la tienda de Google y cada mes este número se ve incrementado. Es por ello que el proceso de desarrollo de aplicaciones para Android ha ganado vital importancia.

El objetivo de esta memoria es estudiar y comparar herramientas que ayuden a mejorar el desarrollo de aplicaciones Android, entregando a los desarrolladores una guía práctica que les permita tomar mejores decisiones durante el transcurso de un proyecto.

Los problemas más comunes que se enfrentan al desarrollar para Android están relacionados con la fragmentación, tanto a nivel de software como de hardware. Por otro lado está la distribución de versiones betas antes de una publicación oficial y el manejo de los crashes. Para solucionar y mitigar estos problemas se han estudiado y comparado herramientas clasificadas en las siguientes categorías: testing, distribución de versiones y reporte de crashes. En la comparación se consideran parámetros que ayudan a los desarrolladores a discernir sobre qué usar, teniendo en cuenta su costo, usabilidad, madurez, documentación, soporte para múltiples plataformas, entre otras.

Para validar las características de las herramientas estudiadas, se implementaron las que más se adaptan a las necesidades de una aplicación que se encuentra disponible en la tienda oficial de Google, Seahorse.

Palabras Claves: Android, Aplicaciones móviles, Testing, Distribución de versiones, Reporte de crashes.

Abstract

The growth that have had the operative system Android is remarkable. There are more than one million applications availables on Google Play and every month these numbers are increasing. Consequently, the development process of Android applications have gained a lot of importance.

The goal of this work is do research and compare tools that help to improve the development process of Android applications, and provide future developers with some guidance that let them take better decisions during a project.

The most common problems for Android developers are related with the fragmentation in two levels, software and hardware. Also there is the distribution of pre-release versions before launching and handling crashes. Have been studied and compared tools to solve and mitigate these problems, classified in the following categories: testing, beta distribution and crash reports. The comparison parameters used to help the developers to decide what tool is best suited for their application are: price, usability, maturity, support for multiplatforms, between others.

The tools that are more suited to the needs of an application at the official store of Google, Seahorse, were implemented to validate its features.

Keywords: Android, Mobile Applications, Testing, Distribution, Crash Reports.

Índice de Contenidos

Resumen	v
Abstract	vi
Índice de Contenidos	vii
Índice de Tablas	xii
Índice de Figuras	xiii
Glosario	xv
1. Introducción	1
1.1. Definición del problema	1
1.2. Objetivos	3
1.2.1. Objetivo principal	3
1.2.2. Objetivos específicos	3
1.3. Estructura del documento	3
2. Estado del Arte	5
2.1. Introducción a Android	5
2.1.1. Inicios de Android	5
2.1.2. Arquitectura	6
2.1.3. ¿Cómo las aplicaciones son compiladas?	9
2.2. Tipos de dispositivos	10
2.3. Versiones	11
2.4. Testing	13
2.5. Problemas al desarrollar en Android	18
2.5.1. Fragmentación	18
Fragmentación a nivel de software	18
Fragmentación a nivel de hardware	20
Fragmentación en otras características	23
Reporte sobre Fragmentación	23

2.5.2. Distribución de versiones Alpha y Beta	25
2.5.3. Crashes	27
3. Herramientas Actuales	29
3.1. Herramientas de Testing	29
3.1.1. Herramientas de Testing Unitario provistas por Android	29
JUnit [40]	30
Simulando objetos (Mock objects) [12]	31
3.1.2. Herramientas de Testing de UI provistas por Android	31
Instrumentación [12]	31
uiautomator [55]	32
uiautomatorviewer [55]	33
Monkey [13]	33
Monkeyrunner [10]	33
3.1.3. Herramientas de Testing Unitario de Terceros	34
EasyMock [57]	34
Mockito [58]	34
Robolectric [59]	34
3.1.4. Herramientas de Testing de UI de Terceros	35
Robotium [56]	35
Espresso [31]	35
Spoon [49]	36
3.2. Herramientas de Distribución de Versiones	37
3.2.1. Herramienta de Distribución de Versiones provista por Android	37
3.2.2. Herramienta de Distribución de Versiones de Terceros	38
HockeyApp [38]	38
AppBlade [15]	39
The Beta Family [26]	39
UserTesting [53]	40
3.3. Herramientas de Reporte de Crashes	41
3.3.1. Herramienta de Reporte de Crashes provista por Android	41
3.3.2. Herramientas de Reporte de Crashes de Terceros	42
Crittercism [22]	42
BugSense [19]	43
Google Analytics [32]	44
ACRA [2]	44
4. Análisis Comparativo	46
4.1. Análisis Comparativo entre las Herramientas de Testing Unitario	46
4.1.1. Framework de Testing de Android	46
Ventajas	46

Desventajas	47
Resumen	47
4.1.2. Robolectric	47
Ventajas	47
Desventajas	47
Resumen	48
4.1.3. EasyMock	48
Ventajas	48
Desventajas	48
Resumen	48
4.1.4. Mockito	49
Ventajas	49
Desventajas	49
Resumen	49
4.1.5. Resumen General	49
4.2. Análisis Comparativo entre las Herramientas de Testing de UI	51
4.2.1. uiautomator	51
Ventajas	51
Desventajas	51
Resumen	52
4.2.2. Robotium	52
Ventajas	52
Desventajas	52
Resumen	52
4.2.3. Espresso	54
Ventajas	54
Desventajas	54
Resumen	54
4.2.4. Resumen General	55
4.3. Otras Herramientas de Testing	56
4.3.1. Monkey	56
Ventajas	56
Desventajas	56
Resumen	56
4.3.2. Monkeyrunner	56
Ventajas	56
Desventajas	57
Resumen	57
4.3.3. Spoon	57
Ventajas	57
Desventajas	58

Resumen	58
4.4. Análisis Comparativo entre las Herramientas de Distribución de Versiones	59
4.4.1. Google Play Console	59
Ventajas	59
Desventajas	59
Resumen	59
4.4.2. HockeyApp	60
Ventajas	60
Desventajas	60
Resumen	60
4.4.3. AppBlade	60
Ventajas	60
Desventajas	61
Resumen	61
4.4.4. The Beta Family	61
Ventajas	61
Desventajas	62
Resumen	62
4.4.5. UserTesting	62
Ventajas	62
Desventajas	63
Resumen	63
4.4.6. Resumen General	64
4.5. Análisis Comparativo entre las Herramientas de Reporte de Crashes . .	66
4.5.1. Google Play Console	66
Ventajas	66
Desventajas	66
Resumen	66
4.5.2. Crittercism	67
Ventajas	67
Desventajas	67
Resumen	67
4.5.3. Bugsense	68
Ventajas	68
Desventajas	68
Resumen	68
4.5.4. Google Analytics	69
Ventajas	69
Desventajas	69
Resumen	69
4.5.5. ACRA	70

Ventajas	70
Desventajas	70
Resumen	70
4.5.6. Resumen General	71
5. Implementación	73
5.1. Implementación de Herramientas de Testing	74
5.1.1. Robolectric	74
5.1.2. Robotium	77
5.1.3. Spoon	78
5.1.4. uiautomatorviewer	82
5.1.5. Monkey	82
5.2. Implementación de Herramientas de Distribución	83
5.2.1. HockeyApp	83
5.2.2. UserTesting	84
5.3. Implementación de Herramientas de Reportes de Crashes	88
5.3.1. Crittercism	88
6. Conclusiones	91
6.1. Conclusiones Generales	91
6.2. Conclusiones Específicas	93
6.3. Trabajo Futuro	95
Bibliografía	96

Índice de Tablas

4.1. a) Tabla comparativa entre herramientas que se enfocan en el testing unitario en Android.	50
4.2. b) Tabla comparativa entre herramientas que se enfocan en el testing unitario en Android.	51
4.3. a) Tabla comparativa entre herramientas que se enfocan en el testing funcional en Android.	56
4.4. b) Tabla comparativa entre herramientas que se enfocan en el testing funcional en Android.	56
4.5. a) Tabla comparativa entre herramientas que se enfocan en la Distribución de Versiones en Android.	65
4.6. b) Tabla comparativa entre herramientas que se enfocan en la Distribución de Versiones en Android.	65
4.7. a) Tabla comparativa entre herramientas que se enfocan tanto en Distribución, como Testing de Versiones en Android.	65
4.8. b) Tabla comparativa entre herramientas que se enfocan tanto en Distribución, como Testing de Versiones en Android.	65
4.9. a) Tabla comparativa entre herramientas que se enfocan en los Reportes de Crashes.	72
4.10. b) Tabla comparativa entre herramientas que se enfocan en los Reportes de Crashes.	72

Índice de Figuras

2.1.	Arquitectura de Android, compuesta por cuatro capas. [54]	8
2.2.	Últimos smartphones y tablets destacadas en el sitio de Android. [9] . .	11
2.3.	Versiones de Android. [50]	12
2.4.	Estadísticas relativas al número de dispositivos que tiene cada versión de Android en Abril del 2014. [6]	13
2.5.	Distribución histórica de versiones de Android.[25]	20
2.6.	Gráfico con la distribución de tamaños de pantalla y sus densidades durante el mes de Junio del 2014. [6],	21
2.7.	Consecuencias de asignar alto y ancho en pixeles en vez de en densidad de pixeles.[11]	22
2.8.	Fragmentación de dispositivos entregado por OpenSignal el 2013. [44] .	24
2.9.	Fragmentación a nivel de fabricantes de dispositivos Android entregado por OpenSignal el 2013. [44]	25
2.10.	Fragmentación de pantallas en dispositivos Android entregado por Open- Signal en Julio del 2013. [44]	26
2.11.	Fragmentación de pantallas en dispositivos iOS entregado por OpenSig- nal en Julio del 2013. [44]	26
2.12.	Concepto de Google sobre como el proceso de reportar crashes hacen al usuario feliz. [5]	28

3.1. Vista del sistema de versiones alphas y betas provisto por Android. Fuente: Elaboración Propia	38
3.2. Vista del sistema de reporte de crashes provisto por Android. Fuente: Elaboración Propia	42
3.3. Vista del sistema de reporte de crashes de Crittercism. Fuente: Elabo- ración Propia	43
3.4. Vista del sistema de reporte de crashes de Bugsense. Fuente: Elaboración Propia	44
3.5. Vista del sistema de reporte de crashes de ACRA. [23]	45
 5.1. Vistas de la aplicación Seahorse. Fuente: Elaboración Propia	73
5.2. Flujo para acceder a la pantalla de Login en la aplicación Seahorse. En la primera imagen se presiona el botón <i>Log In</i> , lo cual abre el menu de la segunda imagen, y se presiona el botón <i>Sign in with email</i> , lo que abre la vista de Login de la última imagen. Fuente: Elaboración Propia . . .	75
5.3. Resultados del test ejecutado a través de Spoon. Fuente: Elaboración Propia	81
5.4. Vista de la herramienta uiautomatorviewer en una aplicación. Fuente: Elaboración Propia	83
5.5. Vista de una aplicación beta de Seahorse en HockeyApp. Fuente: Ela- boración Propia	85
5.6. Gráficos que entrega Crittercism sobre cantidad de crashes y usuarios afectados en un periodo de un mes. Fuente: Elaboración Propia	90

Glosario

ADB	Android Debug Bridge
ANR	Application Not Responding
API	Application Programming Interface
APK	Android Package
GPS	Global Positioning System
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JVM	Java Virtual Machine
RAM	Random Access Memory
SD	Secure Digital
SDK	Software Development Kit
SMS	Short Message Service
SSL	Secure Sockets Layer
UI	User Interface
XML	eXtensible Markup Language

Capítulo 1

Introducción

1.1. Definición del problema

Android es una sistema operativo emergente de código abierto, diseñado especialmente para dispositivos móviles, el cual fue presentado el año 2007. El crecimiento que ha tenido los últimos años ha sido considerable, dominando el mercado ampliamente, existiendo más de mil millones de dispositivos activados en todo el mundo.

El gran problema que ha tenido que enfrentar la gente que desarrolla aplicaciones para Android es la fragmentación. Por un lado está la fragmentación a nivel de hardware generada por los más de 11.000 diferentes tipos de dispositivos [44], sólo considerando smartphones y tablets, ya que también existen notebooks, netbooks y televisores que tienen Android. Esto conlleva dificultades a la hora de diseñar y desarrollar aplicaciones ya que es prácticamente imposible poder testear una aplicación en cada uno de los dispositivos para los cuales estará disponible. Debido a esto, lo más probable es que existan problemas en diferentes áreas, por ejemplo, si la aplicación no está lista para soportar variadas resoluciones de pantalla, la interfaz gráfica no se verá como fue diseñada. Esto es solamente uno de los problemas que puede ocurrir debido a la diversidad de dispositivos, ya que también se debe tener en cuenta que cada uno de los teléfonos y tablets tienen especificaciones distintas de memoria, RAM, procesador,

fabricante, etc. Por otro lado existe la fragmentación a nivel de software, provocada por las ocho versiones de Android que se encuentran vigentes hoy en día [6]. Esto conlleva que, por ejemplo, sea necesario tener un buen sistema de reporte de crashes, ya que muchas veces por más que el código funcione de forma correcta en un dispositivo con Android 4.3, en el mismo dispositivo con Android 4.0.4 se puede comportar de forma distinta. Si bien, estos son sólo algunos de los problemas que se deben enfrentar a causa de la fragmentación, existen muchos más.

Desde su lanzamiento hasta la fecha, Android ha estado acompañado por una activa comunidad de desarrolladores. Ellos son los responsables de que exista una gran cantidad de proyectos de código abierto que buscan dar solución a los problemas mencionados anteriormente. Estas herramientas generalmente se dan a conocer a través de comunidades como Github o Google+, por lo que se encuentran dispersas y normalmente sólo se conoce una parte de las posibles soluciones disponibles. Esto provoca que muchas veces, por desconocimiento o falta de tiempo, el desarrollador tome una decisión apresurada y no use la biblioteca que más beneficie a su proyecto.

Si se revisan estadísticas del sitio AppBrain [16] correspondientes al 3 de Mayo del 2014, se puede ver que de un total de 1.203.555 aplicaciones disponibles para descargar, un 41.4 % tienen una calificación promedio menor a 3 estrellas, de un total de 5. Esta calificación es entregada por los mismos usuarios y oscila entre 1 y 5 estrellas. Normalmente, si se desea tener una buena nota por parte del usuario, es necesario que la aplicación sea de utilidad y resuelva un problema real, aunque también es muy importante que la aplicación sea robusta y estable. Además, cada mes se crean entre 10.000 y 80.000 nuevas aplicaciones [16], por lo que es fundamental diferenciarse del resto, entregando un producto de calidad.

1.2. Objetivos

1.2.1. Objetivo principal

Estudiar y comparar herramientas que ayuden a mejorar la calidad de las aplicaciones desarrolladas para Android, de tal manera de proveer a los desarrolladores una guía práctica que les permita tomar mejores decisiones en el transcurso de un proyecto.

1.2.2. Objetivos específicos

- Identificar los distintos problemas existentes durante el desarrollo de aplicaciones Android.
- Estudiar las herramientas que actualmente permiten mejorar la calidad de las aplicaciones, y clasificarlas en base a los distintos problemas que buscan solucionar.
- En base a la clasificación realizada, llevar a cabo un análisis comparativo entre las herramientas estudiadas.
- En base al análisis realizado, implementar las herramientas que puedan ser más útiles en un entorno real de desarrollo.

1.3. Estructura del documento

Esta memoria está organizada de la siguiente manera: El capítulo 2 corresponde al Estado del Arte. En él se realizará una introducción a Android, explicando a grandes rasgos sus inicios, arquitectura, tipos de dispositivos, entre otras cosas. Además se estudiarán los problemas más comunes, inherentes a un sistema operativo tan fragmentado como Android. En el capítulo 3 se presenta un listado clasificado con las distintas herramientas para mejorar el desarrollo de aplicaciones. Se examinará cada una de éstas, lo que permitirá tener un panorama general de las fortalezas y debilidades que poseen.

En el capítulo 4 se realizará un análisis y se compararán algunas herramientas para poder tener claras las diferencias entre cada una y concluir qué se debe usar y para qué casos. En el capítulo 5 se llevará a cabo la implementación de las herramientas más destacadas en un entorno real de desarrollo. Finalmente, en el capítulo 6 se presentan las conclusiones obtenidas a partir de los análisis e implementaciones previas.

Capítulo 2

Estado del Arte

En este capítulo se dará a conocer una breve descripción del sistema operativo Android. Se comenzará con una introducción, hablando de sus inicios, su arquitectura y la evolución que ha tenido con el tiempo. Además se hablará sobre los problemas más comunes al momento de comenzar a desarrollar una aplicación para Android.

2.1. Introducción a Android

Android es un sistema operativo basado en Linux, diseñado principalmente para dispositivos móviles táctiles, tales como smartphones y tablets.

2.1.1. Inicios de Android

Android, Inc. fue fundada en Palo Alto, California en Octubre del 2003 por Andy Rubin, Rich Miner, Nick Sears and Chris White. Su objetivo era desarrollar dispositivos móviles más inteligentes, con un mayor foco en la localización del dueño y en su personalización .

Google compró a Android Inc. el 17 de Agosto del 2005 [24]. Poco se sabía sobre esta compañía para ese entonces ya que estuvo funcionando de forma secreta, sin dar a conocer detalles sobre lo que desarrollaban. Muchos asumían que Google estaba planeando entrar al mercado de dispositivos móviles. De ahí en adelante los esfuerzos de

Google se enfocaron en conversaciones con fabricantes y carriers, con la promesa de proveer un sistema flexible y actualizable.

Sin embargo, la aparición del iPhone el 9 de Enero del 2007 [17] tuvo un efecto disruptivo en el desarrollo de Android. Hasta el momento se contaba con un prototipo, el cual se acercaba más a lo que podría ser un teléfono BlackBerry, sin pantalla táctil y con un teclado físico. Por lo que se comenzó inmediatamente un trabajo de reingeniería del sistema operativo y del prototipo para que fuese capaz de competir con el iPhone.

El 6 de Noviembre del 2007 [3] fue fundada la Open Handset Alliance, una alianza comercial liderada por Google con compañías tecnológicas como HTC, Sony y Samsung, operadores de carriers como Nextel y T-Mobile y fabricantes de chips, con el objetivo de desarrollar estándares abiertos para dispositivos móviles. El primer smartphone disponible que funcionaba sobre Android fue el HTC Dream, lanzado el 22 de Octubre del 2008.

2.1.2. Arquitectura

La arquitectura del sistema Android [14], también llamado stack, se puede apreciar en la figura 2.1 y está compuesta por cuatro capas:

- **Kernel de Linux:** La capa más profunda es su núcleo en Linux, un sistema operativo abierto, portable y seguro. Para cada pieza de hardware, como la cámara o el bluetooth, existe un driver dentro del kernel, que permite a la capa superior hacer uso de ella, por lo que funciona como una capa de abstracción. El kernel además se encarga de la gestión de los diversos recursos del dispositivo, como la energía o la memoria, elementos de comunicación, procesos, etc.
- **Bibliotecas:** La segunda capa en el stack contiene bibliotecas nativas, las cuales están escritas en C o C++, y son compiladas para la arquitectura específica

del dispositivo. En la mayoría de los casos el fabricante es quien se encarga de instalarla en su dispositivo. Las bibliotecas incluidas en esta capa son: el motor gráfico OpenGL, el sistema de gestión de base de datos SQLite, cifrado de comunicaciones SSL, motor de manejo de tipos de letra FreeType, entre otras.

El entorno de ejecución de Android también está compuesto por bibliotecas, por lo que no se considera una capa. Debido a las limitaciones de los dispositivos en los que debe funcionar, Google decidió crear la máquina virtual Dalvik, que funciona de forma similar a la máquina virtual de Java. Esta permite crear aplicaciones con un mejor rendimiento y menor consumo de energía, lo que es muy importante en dispositivos móviles. Además en el entorno de ejecución se incluyen la mayoría de las bibliotecas básicas de Java.

- **Marco o Framework de Aplicaciones:** La tercera capa está compuesta por todas las clases y servicios que se utilizan al momento de programar aplicaciones.

Los componentes que posee son:

- **Administrador de Activities (Activity Manager):** Gestiona la pila de *Activities* de la aplicación, como también su ciclo de vida. En el desarrollo en Android, se llama *Activity* a cada una de las pantallas con las que el usuario interactúa. Por lo que se puede asumir que una aplicación en la mayoría de los casos va a tener varias *Activities*. Estas son uno de los componentes más importantes dentro de una aplicación.
- **Administrador de ventanas (Windows Manager):** Organiza lo que se mostrará en pantalla. Crea las superficies en la pantalla, que posteriormente estarán ocupadas por las *Activities*.
- **Proveedor de contenidos (Content Provider):** Encapsula los datos que pueden ser compartidos por las aplicaciones, facilitando la comunicación entre éstas.

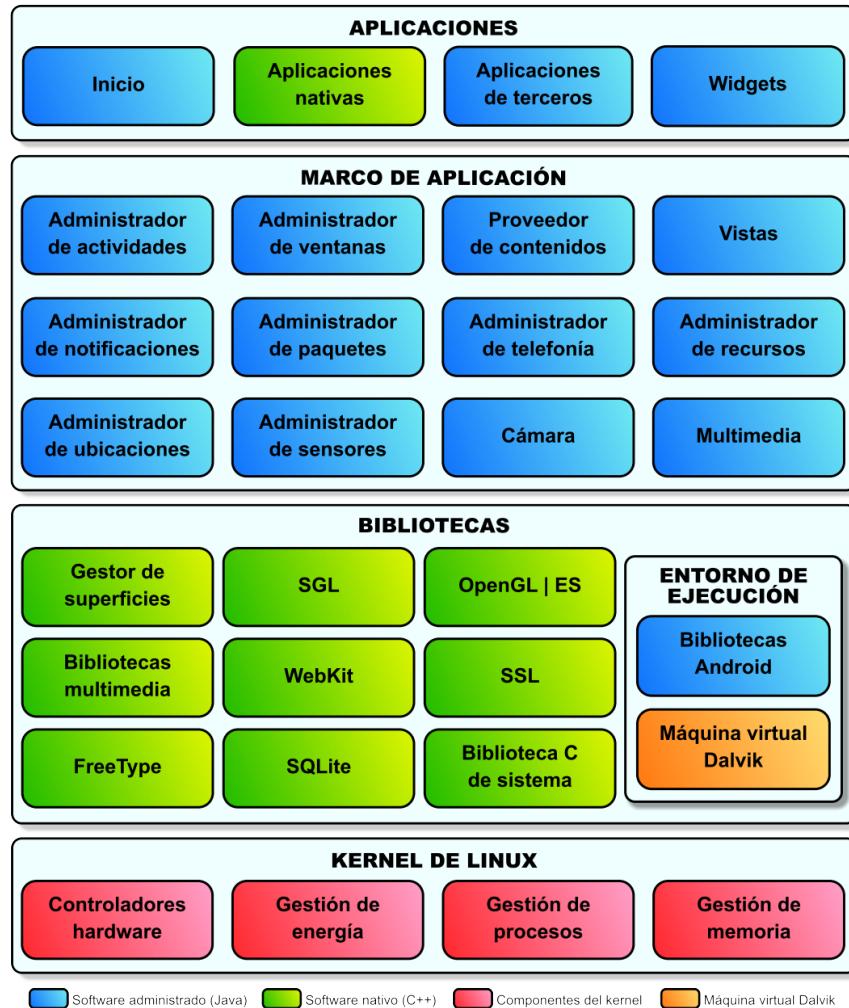


Figura 2.1 – Arquitectura de Android, compuesta por cuatro capas. [54]

- **Vistas (Views):** Son los elementos que permiten construir las interfaces de usuario, como listas, botones, textos, hasta otros elementos más avanzados como visores de mapas.
- **Administrador de notificaciones (Notification Manager):** Provee los servicios que notifican al usuario, mostrando alertas en la barra de estado. También permite activar el vibrado, reproducir alertas de sonido y utilizar las luces del dispositivo.

- **Administrador de paquetes (Package Manager):** Gestiona la instalación de nuevos paquetes y además permite obtener información sobre los que ya están instalados.
 - **Administrador de telefonía (Telephony Manager):** Permite realizar llamadas, como también el envío y recepción de SMS.
 - **Administrado de recursos (Resource Manager):** A través de este administrador se podrá acceder a los elementos que no forman parte del código, como imágenes, sonidos, layouts, etc.
 - **Administrado de ubicaciones (Location Manager):** Permite obtener la posición geográfica actual del dispositivo a través de GPS o redes.
 - **Administrado de sensores (Sensor Manager):** Permite la manipulación de distintos sensores del dispositivo, como el acelerómetro, giroscopio, brújula, sensor de proximidad, etc.
 - **Cámara:** Permite el uso de la cámara del dispositivo para la obtención de fotografías o vídeos.
 - **Multimedia:** Permite la visualización y reproducción de imágenes, vídeos y audio.
- **Aplicaciones:** En esta capa se encuentran todas las aplicaciones del dispositivo, tanto las preinstaladas, como aquellas instaladas por el usuario. También está la aplicación principal del sistema, el Inicio o launcher, desde donde se inician todas las aplicaciones.

2.1.3. ¿Cómo las aplicaciones son compiladas?

Al comenzar a desarrollar una aplicación de Android, generalmente se crea un proyecto usando un IDE (Integrated Development Environment) como Eclipse o Android Studio. El proyecto contendrá código fuente en Java y recursos. Cuando se compila el

proyecto [43, p. 14] lo que ocurre es que se generan los Bytecode Java (archivos .class) en base al código fuente Java (archivos .java). Luego se compilán estos archivos .class generándose archivos ejecutables Dalvik (archivos .dex), los cuales pueden ser ejecutados por la máquina virtual Dalvik que está disponible en todos los dispositivos Android.

Al compilar un proyecto se colocan los archivos .dex y el resto de los archivos del proyecto en uno solo llamado APK (Android Package). Este contiene todos los archivos necesarios para ejecutar la aplicación, incluyendo los .dex, recursos compilados, recursos sin compilar, y una versión binaria del Android Manifest.

El *Android Manifest* es un archivo que especifica información esencial que el sistema debe tener antes de ejecutar la aplicación. Toda aplicación debe tener este archivo de forma no binaria en su proyecto.

Por razones de seguridad todas las aplicaciones de Android deben ser firmadas digitalmente con un certificado.

Finalmente el ADB (Android Debug Bridge) permiten que el IDE se comunique con un dispositivo físico de Android o un emulador.

2.2. Tipos de dispositivos

En el sitio web de Android [9], se pueden apreciar los dos tipos de dispositivos más populares de la plataforma, los smartphones y las tablets (Figura 2.2). Sin embargo, debido a que el código de Android es de código abierto, éste puede ser personalizado para que funcione con otros tipos de dispositivos electrónicos.

A continuación se listan los otros dispositivos que cuentan con Android:[43, p. 5]

- Lectores de libros.



Figura 2.2 – Últimos smartphones y tablets destacadas en el sitio de Android. [9]

- Cámaras.
- Sistemas en vehículos.
- Casas inteligentes.
- Consolas de videojuegos.
- Televisores inteligentes.
- Relojes inteligentes.

2.3. Versiones

En la figura 2.3 se detallan las distintas versiones que ha tenido Android. La primera versión comercial fue lanzada en Septiembre del 2008. Android está bajo constante desarrollo por parte de Google y de la Open Handset Alliance, contando con un gran número de actualizaciones desde su lanzamiento.

Desde Abril del 2009, los nombres de las versiones de Android han estado relacionados con postres y dulces, y además han seguido un orden alfabético. El orden es Cupcake (1.5), Donut (1.6), Eclair (2.0-2.1), Froyo (2.2-2.2.3), Gingerbread (2.3-2.3.7), Honeycomb (3.0-3.2.5), Ice Cream Sandwich (4.0-4.0.4), Jelly Bean (4.1-4.3), y

KitKat(4.4). El 3 de Septiembre del 2013, Google anunció que existían un billón de dispositivos activos usando el sistema operativo Android en todo el mundo. La actualización más reciente de android fue KitKat 4.4, lanzado para dispositivos comerciales el 22 de Noviembre del 2013.



Figura 2.3 – Versiones de Android. [50]

Al comenzar el desarrollo de una aplicación Android, se debe decidir cuál va a ser la API mínima a la que se dará soporte. Esto tendrá repercusiones al momento de que un usuario desee instalar la aplicación, ya que si su dispositivo cuenta con una versión como Froyo o Eclair, lo más probable es que no pueda instalar prácticamente ninguna de las aplicaciones disponibles en Google Play, la tienda en que se encuentran todas las aplicaciones que suben los desarrolladores.

Android actualiza mes a mes las estadísticas relativas al número de dispositivos que tiene cada versión del sistema operativo [6]. Esto ayuda a tener una guía sobre cuál va a ser la API mínima soportada. En la figura 2.4 se muestran las estadísticas correspondientes al mes de Abril del 2014. Esta información es recolectada durante los últimos 7 días de cada mes. Además se ignoran las versiones que tienen menos

de un 0.1 %. Se puede apreciar que el sistema operativo que hoy en día es dominante corresponde a Jelly Bean con más de un 60 %. El nuevo sistema operativo KitKat tiene sólo un 5.3 % debido principalmente a que los operadores y fabricantes aún no tienen listas sus versiones personalizadas de KitKat, en las que pueden incluir nuevas funcionalidades o quitar lo que estimen conveniente.

Version	Codename	API	Distribution
2.2	Froyo	8	1.1%
2.3.3 - 2.3.7	Gingerbread	10	17.8%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	14.3%
4.1.x	Jelly Bean	16	34.4%
4.2.x		17	18.1%
4.3		18	8.9%
4.4	KitKat	19	5.3%

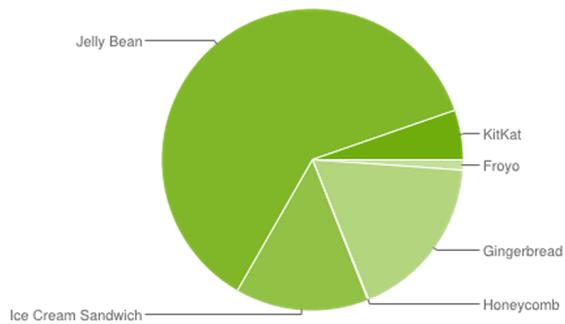


Figura 2.4 – Estadísticas relativas al número de dispositivos que tiene cada versión de Android en Abril del 2014. [6]

2.4. Testing

Con tantos dispositivos y sistemas operativos vigentes, asegurar la calidad de la aplicación a través de testing es un proceso vital y necesario, aunque también puede ser uno de los mayores dolores de cabeza para los desarrolladores. Lo que funciona perfectamente en un dispositivo, en otro puede no resultar como se espera. Es por ello que es absolutamente necesario el uso de herramientas que ayuden a reducir los riesgos inherentes de que una aplicación sea compatible con más de 11.000 dispositivos distintos [44]. A continuación se darán a conocer los tipos de testing más conocidos.

Testing unitario

Un test unitario (unit test) es una pieza de código escrito por un desarrollador que ejecuta una funcionalidad específica en el código que va a ser testeado. Este tipo de test se enfoca en aislar un componente, por ejemplo un método o una clase, para ser capaces de testearlo de forma replicable. Es por esto que los test unitarios y los objetos simulados (mock objects) normalmente se usan en forma conjunta. Estos objetos simulados se usan para poder repetir el test innumerables veces. Por ejemplo, si se quisiera testear el momento en que se borra información desde una base de datos, probablemente no se quiere que los datos realmente se borren y que la próxima vez que se desee testear, éstos ya no se encuentren.

Los test unitarios aseguran que el código funcione como se espera. También son muy útiles para asegurar que el código sigue funcionando correctamente después de hacer cambios en otras partes del proyecto, al momento de arreglar bugs o añadir nuevas funcionalidades.

Testing de User Interface (UI)

Además de testear los componentes individuales que permiten el funcionamiento de la aplicación, como *Activities* y servicios, es muy importante testear el comportamiento de la interfaz de la aplicación cuando está en funcionamiento en un dispositivo. El testing de UI asegura que la aplicación se comportará de forma correcta en respuesta de acciones que realice el usuario en un dispositivo, tales como escribir en el teclado, presionar botones o imágenes, entre otros controles. Se debe tener consideración especial con los test que involucran elementos de UI, ya que únicamente la hebra principal tiene permisos para alterar la UI en Android.

Un estrategia común es testear de forma manual la UI, verificando que la aplicación

se comporta como se espera al realizar una serie de acciones. Sin embargo, este enfoque puede consumir mucho tiempo y ser bastante tedioso, como también, se pueden pasar por alto algunos errores. Un método más eficiente y confiable sería automatizar el testing de la interfaz con algún framework que facilite esta tarea.

Testing de integración

Los test de integración están diseñados para testear el comportamiento que los componentes individuales tienen cuando funcionan de forma conjunta. Esto normalmente se realiza una vez que ya se han aprobado los test unitarios. Además tienden a ser más complejos y lentos que los test unitarios.

Testing de aceptación

Normalmente estos tipos de test son creados por profesionales del área de negocios y de control de calidad. Además son expresados en un lenguaje de negocios. Estos son test de alto nivel para testear el correcto funcionamiento de los requerimientos o características que debería tener la aplicación. Los testers y desarrolladores también pueden colaborar en la creación de éstos.

Testing de sistema

El sistema es testeado como un todo, y la interacción entre los componentes, software y hardware es testeada. Normalmente, los test de sistema incluyen:

- Smoke test: Es un testing rápido que se lleva a cabo sobre toda la aplicación. Su objetivo no consiste en encontrar bugs, sino que asegurar que las funcionalidades básicas se comportan de manera correcta.
- Test de desempeño: Los test de desempeño miden alguna característica de un componente en una forma replicable. Si se necesitan mejoras en el desempeño de algún componente de la aplicación, el mejor enfoque es medir el desempeño antes

y después de la inclusión de un cambio. De esta forma se entiende claramente el impacto que ha tenido el cambio en el desempeño.

La lista anterior describe los posibles tests que hoy en día existen para asegurar la calidad de las aplicaciones móviles, pero tan importante como saber hacer testing, es saber qué cosas son necesarias testear. A continuación se listan algunas situaciones comunes relacionadas con Android que se deberían tener en cuenta al momento de testear.

Cambios de orientación

Para dispositivos que soportan múltiples orientaciones, Android detecta los cambios de orientación cuando el usuario rota el dispositivo, dejándolo en *landscape* (posición horizontal) en vez de *portrait* (posición vertical).

Cuando ocurre esto, el comportamiento por defecto es destruir y recomenzar la *Activity*. Se deberían tener en cuenta las siguientes preguntas:

- ¿Se dibuja de forma correcta la pantalla?
- ¿La aplicación mantiene el estado? La *Activity* no debería perder nada que el usuario haya ingresado en la UI.

Cambios de configuración

También pueden ocurrir otros cambios más generales en el sistema, como un cambio de idioma. Este tipo de cambios desencadena el comportamiento por defecto de destruir y recomenzar la *Activity*.

Dependencias de fuentes externas

Si la aplicación depende de acceso a internet, o usa GPS, entonces se debería testear qué pasa cuando estos recursos no están disponibles.

Cambios en segundo plano

Si la aplicación está inactiva y ya ha pasado a segundo plano, lo más probable es que el sistema destruya la o las *Activities* de la aplicación para dar memoria a otras aplicaciones que estén ejecutándose actualmente. Es por ello que es necesario testear el ciclo de vida de la *Activity* para corroborar si se destruye y recomienza de forma exitosa, sin pérdida del estado actual.

Para el caso de testing de clases en aplicaciones Android, es posible que ocurran dos casos: que las clases realicen llamadas a la API de Android o que sólamente usen código Java.

Testing de clases en Java [55]

Si las clases que se tienen no hacen llamadas a la API de Android, se puede usar el framework de JUnit sin ninguna restricción.

La ventaja de este método es que se puede usar cualquier framework de testing que sea para Java y la velocidad con que se ejecutan los tests debería ser mucha más rápida comparada con los tests que requieren de un sistema con Android.

Testing de clases en Java que usan la API de Android [55]

Si se quieren hacer tests que usen la API de Android, éstos necesitan llevarse a cabo en un dispositivo con Android. Esto hace que la ejecución de los tests tome más tiempo, principalmente porque el archivo *android.jar* no contiene el código del framework de Android. Este archivo es únicamente usado al momento de compilar una aplicación. Una vez que la aplicación está instalada, se utilizará el *android.jar* que está en el dispositivo.

2.5. Problemas al desarrollar en Android

Ahora que ya se han dado a conocer aspectos básicos sobre Android, se puede profundizar en los problemas más comunes que se enfrentan al desarrollar aplicaciones.

2.5.1. Fragmentación

La Fragmentación es el elemento que más afecta a Android. Debido a los más de 11.000 diferentes tipos de dispositivos [44], como también a las ocho versiones vigentes del sistema operativo [6], es mucho más difícil desarrollar una aplicación robusta y estable, ya que es prácticamente imposible poder probar la aplicación en todas las combinaciones de dispositivos y software existentes. Es posible clasificar la fragmentación en dos categorías, a través de las cuales desembocan la mayoría de los problemas que un desarrollador debe enfrentar: software y hardware.

Fragmentación a nivel de software

Como ya se mencionó anteriormente, existen muchas versiones del sistema operativo Android vigentes hoy en día. Esto priva al desarrollador de métodos útiles al momento de programar su aplicación, ya que se debe establecer una API mínima. En base a las estadísticas que provee Android, la mayoría de los desarrolladores decide dar soporte desde Gingerbread en adelante. Si el desarrollador desea utilizar métodos de una API superior a la de Gingerbread, debe especificar en el código fuente que esa parte sólo tiene que ser ejecutada si el dispositivo del usuario es mayor o igual a la API 11. La siguiente porción de código [8] es un ejemplo de lo que los desarrolladores deben hacer:

```
if (Build.VERSION.SDK_INT > Build.VERSION_CODES.GINGERBREAD_MR1) {  
    // Aquí va código superior a la API 10 de Gingerbread  
}
```

Esto provoca muchas veces que el desarrollador deba programar una funcionalidad más de una vez. Actualmente varios desarrolladores están optando por dar soporte a sus aplicaciones desde Ice Cream Sandwich en adelante, debido principalmente a la gran recepción que ha tenido Jelly Bean y a la caída constante que está teniendo Gingerbread. Si se toma en cuenta que este último sistema operativo fue lanzado el año 2010 [4] y aún cuenta con cerca de un 20 %, se puede apreciar claramente el nivel de fragmentación que existe, principalmente por la rápida evolución con la que Android ha estado mejorando su sistema operativo, lanzando aproximadamente una nueva versión cada año.

A continuación, en el gráfico de la figura 2.5 se puede ver la distribución histórica de versiones que ha tenido Android con el pasar de los años. Si bien se visualiza que aún existe una gran fragmentación, esto ha ido disminuyendo y Android cada vez se está convirtiendo en un sistema operativo más estable y maduro. Por ejemplo, si se compara el porcentaje que tenía Gingerbread en el año 2013 (39.8 %) con el de este año (17.8 %) se ve que hay una diferencia sustancial, y gran parte de este porcentaje se ha trasladado a Jelly Bean, que el año pasado contaba con un 25 % del mercado y hoy en día cuenta con más del 60 %.

Muchas veces los crashes y errores en los que la aplicación deja de funcionar correctamente, ocurren en un sistema operativo más antiguo y ya fueron arreglados en los sistemas operativos más nuevos. Esto ocasiona que al probar la aplicación en un smartphone como un Nexus 4 o como un Samsung S3, no ocurran problemas que si podrían verse en dispositivos más antiguos. Además, debido a que a veces los operadores o fabricantes no ofrecen actualizaciones al sistema actual, el dispositivo no es actualizado, por lo que estos errores persistirán.

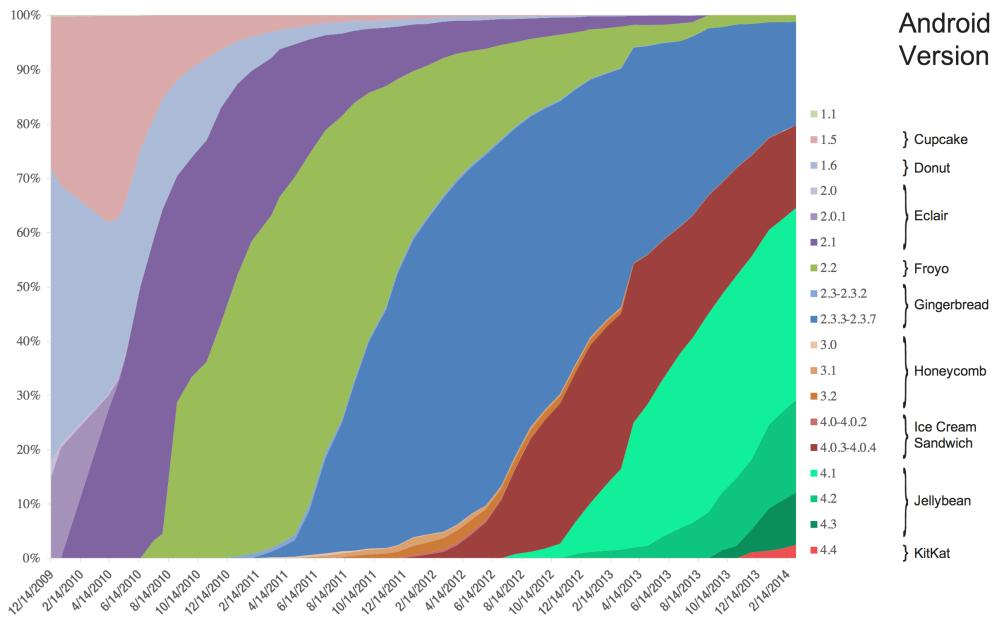


Figura 2.5 – Distribución histórica de versiones de Android.[25]

Fragmentación a nivel de hardware

Este tipo de fragmentación es la que más afecta a los desarrolladores. Por un lado, la gran cantidad de dispositivos es la que ha permitido la rápida evolución de Android, ya que cualquier fabricante puede adaptar el sistema operativo a sus necesidades e incluirlo en su hardware. Para los desarrolladores, este tipo de fragmentación es la que genera más pesadillas, pues cada dispositivo con Android cuenta con diferentes características. A continuación se detallan los problemas derivados por la fragmentación de hardware.

Fragmentación en los tamaños de pantalla

Existen cuatro tamaños generales de pantallas [6]: pequeña (small), normal, grande (large) y extra grande (xlarge). Para optimizar la experiencia del usuario, muchas veces se debe implementar una interfaz distinta para un tipo específico de pantalla. Esto se lleva a cabo a través de un archivo llamado layout, escrito en XML que define los elementos presentes en la interfaz.

Si se desea usar el mismo layout para todas las pantallas simplemente se deben ir guardando estos archivos en la carpeta de recursos del proyecto, *res/layout*. En caso que se quiera agregar un layout distinto para un tipo de pantalla, además de tener un archivo en la ruta antes mencionada, se debe crear una nueva carpeta de layouts, añadiendo el sufijo que corresponda a cada tamaño de pantalla.

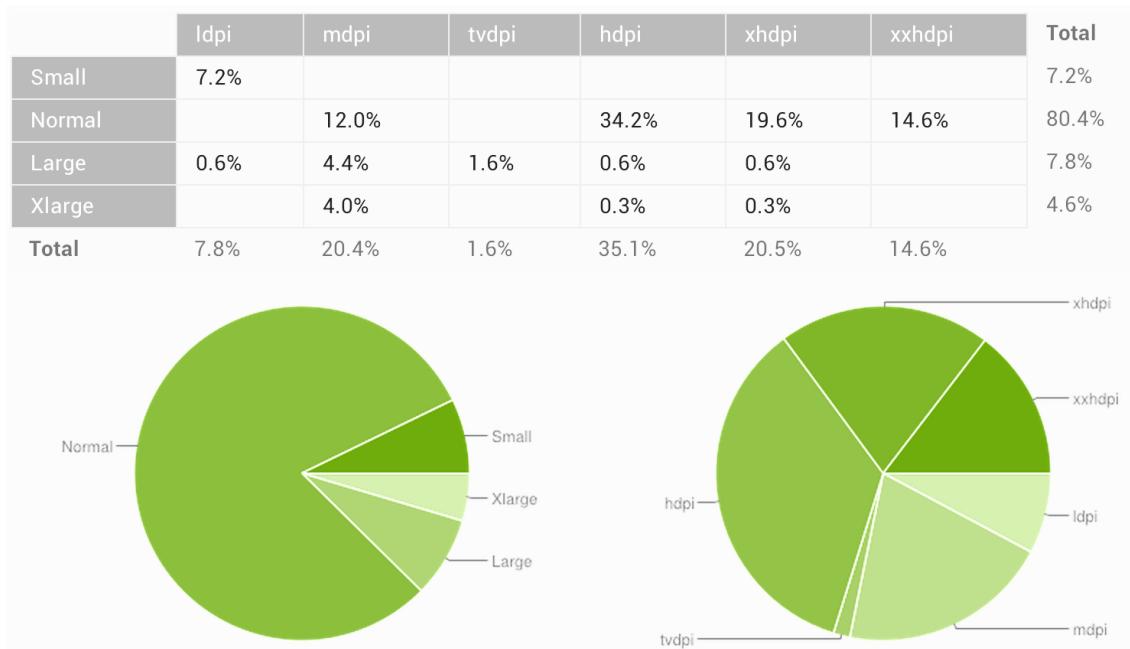


Figura 2.6 – Gráfico con la distribución de tamaños de pantalla y sus densidades durante el mes de Junio del 2014. [6],

Por ejemplo, si se quisiera agregar un layout distinto para pantallas grandes (large), se debe añadir un archivo de layout a la carpeta *res/layout-large*, por lo que existirán dos versiones de este archivo, uno en la carpeta antes señalada, mientras que el otro estará en la carpeta general de layouts, *res/layout*.

En el sitio web de Android [9], se entregan estadísticas mensuales sobre el porcentaje de dispositivos que tiene cada tamaño de pantalla. En el gráfico de la figura 2.6 se muestra la distribución de tamaños de pantalla durante el mes de Junio del 2014.

Fragmentación en las resoluciones de pantalla

Android categoriza las resoluciones de cada pantalla en base a la densidad de píxeles que poseen. Existen cinco tipos de densidades [6]: baja (ldpi), media (mdpi), alta (hdpi), extra-alta (xhdpi) y extra-extra-alta (xxhdpi). Además existe otro tipo de densidad, la cual es usada principalmente para televisores (tvdpi).

Muchas veces es necesario agregar diferentes versiones del mismo recurso gráfico. Por ejemplo, si sólo se agrega un recurso gráfico en la resolución más alta (xxhdpi), este probablemente se verá bien en resoluciones altas como xxhdpi y xhdpi, pero para las más bajas, como hdpi o mdpi, la imagen será redimensionada y se perderá mucha calidad.

Al agregar elementos de UI como botones, textos, tablas, entre otros, también se debe considerar la densidad de píxeles, ya que si el ancho y alto se asignan en píxeles, va a ocurrir lo que se muestra en la figura 2.7. Por ello, es muy importante asignar valores que estén en la escala de densidad de píxeles.

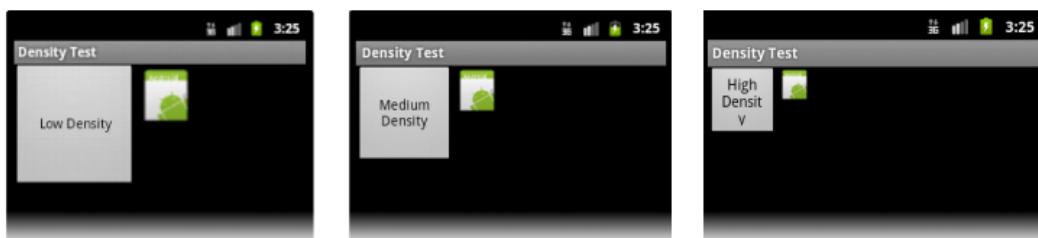


Figura 2.7 – Consecuencias de asignar alto y ancho en píxeles en vez de en densidad de píxeles.[11]

Aún con estas precauciones, es muy difícil saber cuáles son los valores de alto y ancho adecuados que deben ser asignados. Esto se debe a que existe una gran diferencia entre cada tipo de resolución, ya que si por ejemplo se asignara a un elemento un ancho de 350dp, no existirán problemas en una pantalla con una resolución alta como

un Nexus 4 (xhdpi), pues el elemento tomaría un espacio de 700px, pero para un dispositivo con resolución más baja como un Samsung Galaxy Young (ldpi) si existirían, ya que el elemento tomaría 262.5px y el ancho de la pantalla de este dispositivo es de 240px. A través del siguiente sitio [37], se puede saber cuál es la densidad de pixeles de prácticamente todos los dispositivos más populares de Android.

En el gráfico de la figura 2.6 se puede apreciar la información que entrega Google sobre la distribución de densidad existente durante el mes de Junio del 2014 [6].

Fragmentación en otras características

Además de los tamaños de pantalla y sus resoluciones, existen otras características muy importantes que los desarrolladores deben tener en cuenta. Una de las principales es la RAM con la que cuenta el dispositivo, ya que es allí donde se cargan las instrucciones que ejecuta el procesador. También es muy importante el procesador y la cantidad de núcleos con los que cuenta. Aunque la mayoría de los dispositivos tiene cámara, también se debe tener en cuenta que algunos no la tiene. Por último, muchas veces los fabricantes como Samsung, hacen cambios en el sistema operativo, cambiando cosas nativas, lo cual produce diferentes experiencias en cada dispositivo, y muchas veces genera errores que el desarrollador no se espera.

Reporte sobre Fragmentación

La compañía *OpenSignal* ha entregado dos reportes sobre la fragmentación en Android en los años 2012 [45] y 2013 [44]. En su último reporte de Julio del 2013 se entregan gráficas que ayudan a visualizar el número de dispositivos existentes. Estas gráficas se basan en 682.000 dispositivos únicos que descargaron la aplicación de OpenSignal. La razón de elegir este número de dispositivos es porque se quería hacer una comparación justa con respecto al reporte entregado el año 2012, en el que también se había tomado una muestra de 682.000 dispositivos.

En la gráfica de la figura 2.8 se apreciar los 11.868 dispositivos distintos que han

descargado la aplicación de *OpenSignal* en los últimos meses. En su reporte del año 2012 este número era de 3.997. En el sitio web [44] se menciona lo siguiente:

“Esta es la mejor forma de visualizar realmente el número de diferentes dispositivos Android que han descargado la aplicación de OpenSignal en los últimos meses. Desde el punto de vista de un desarrollador, comparando la fragmentación de este año con el anterior, hemos visto que se ha triplicado, con dispositivos incluso más raros de todas partes del mundo. Si se quiere entender el desafío de crear una aplicación que funcione con todos los dispositivos que pueden descargarla, esta imagen es un buen punto para comenzar!”

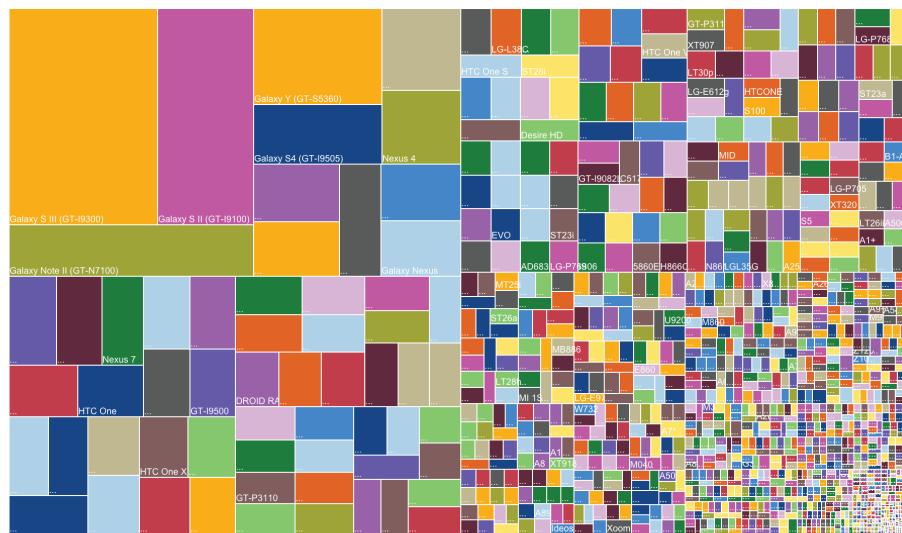


Figura 2.8 – Fragmentación de dispositivos entregado por OpenSignal el 2013. [44]

Similar a las estadísticas de dispositivos, la gráfica de la figura 2.9 muestra el porcentaje de mercado que maneja cada uno de los fabricantes. Se puede ver a Samsung claramente sobre el resto, con un 47.5 % del mercado. Sony se encuentra en la segunda posición con un 6.5 %, menos de un séptimo de lo que tiene Samsung. Algunas otras marcas que se encuentran en esta gráfica, pero que tienen porcentajes muy bajos son Motorola, HTC, Huawei, LG y Google.

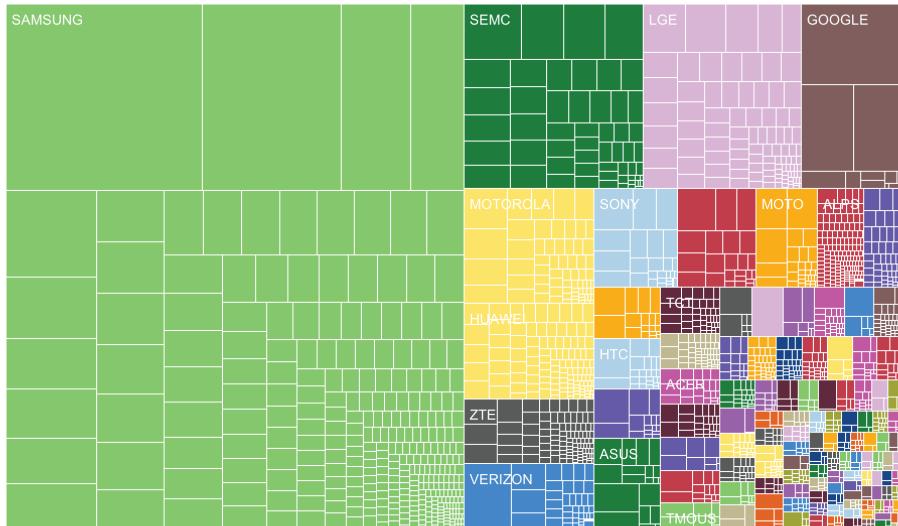


Figura 2.9 – Fragmentación a nivel de fabricantes de dispositivos Android entregado por OpenSignal el 2013. [44]

Con respecto a los tamaños de pantallas, se menciona que la clave del éxito para cualquier aplicación es tener una buena interfaz, y en este aspecto Android presenta dos desafíos para los desarrolladores. Primero, los fabricantes tienden a producir sus propias variantes en la interfaz del usuario, cambiando el aspecto de varios elementos nativos de Android, como botones, switchs, listas, entre otros, por lo que no se puede entregar la misma experiencia a todos los usuarios a menos que se hagan cambios profundos para personalizar la interfaz. El otro desafío tiene relación con los tamaños de pantalla, ningún otro sistema operativo móvil tiene tanta diversidad. En las figuras (2.10 y 2.11) se incluye una comparación entre los tamaños de pantalla de los dispositivos Android en contraste con los de iOS. Las líneas más oscuras representan la frecuencia de estas pantallas. Se hace énfasis en que en la gráfica lo que se muestra son los tamaños de pantalla física, no los tamaños en pixeles.

2.5.2. Distribución de versiones Alpha y Beta

Durante el proceso de desarrollo de una aplicación, normalmente se compilan versiones Alpha y Beta, las cuales son versiones que no son las finales, por lo que se necesita

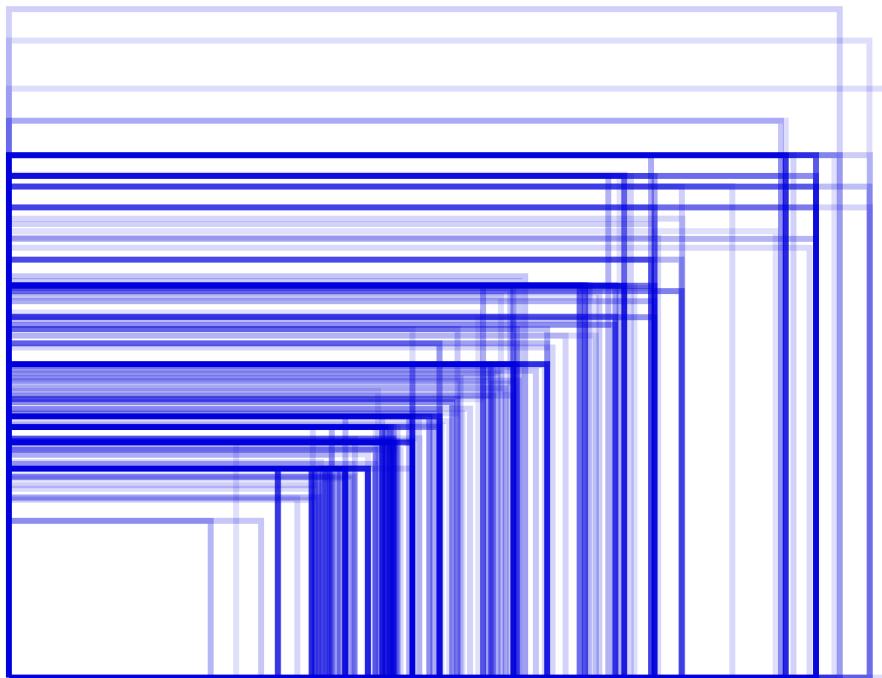


Figura 2.10 – Fragmentación de pantallas en dispositivos Android entregado por OpenSignal en Julio del 2013. [44]

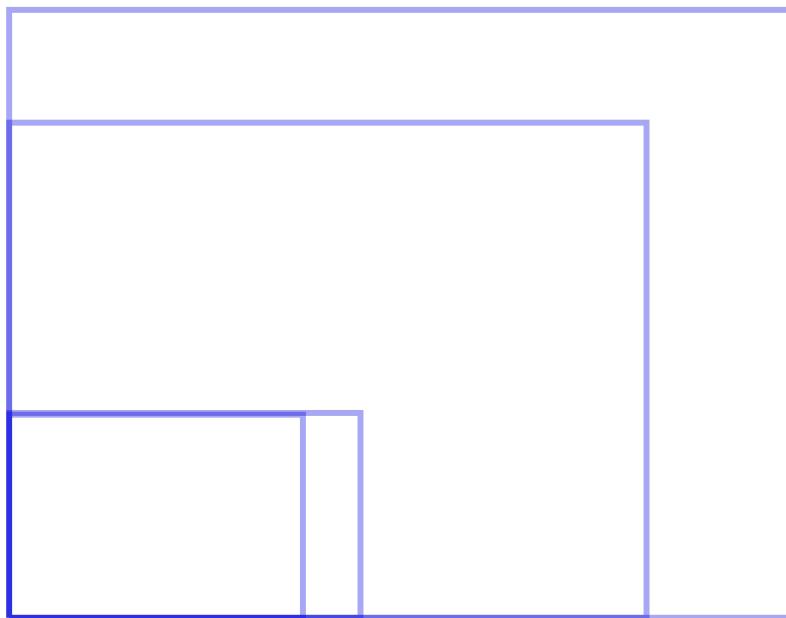


Figura 2.11 – Fragmentación de pantallas en dispositivos iOS entregado por OpenSignal en Julio del 2013. [44]

una forma de distribuirlas para que sean testeadas, antes de su publicación oficial. Este proceso no solamente se lleva a cabo antes de la primera publicación oficial, sino que es un proceso constante, que se realiza en cada iteración, y antes de cada actualización.

Generalmente, la distribución se realiza con un reducido grupo de usuarios de prueba, escogidos por los desarrolladores. Esto permite corroborar que todas las características de la aplicación están funcionando correctamente antes de subir una actualización. Además ayuda a encontrar y corregir posibles problemas a través de la retroalimentación obtenida por parte de los testers.

Al ser un proceso cíclico, normalmente los usuarios de prueba tienen que recibir versiones semanales de la aplicación, por lo que es importante mantener el contacto con ellos y contar con vías de comunicación siempre disponibles.

Por otro lado, también existen desarrolladores que quieren dejar disponibles sus versiones Alphas y Betas a todo el mundo, ya que mientras más usuarios las usen, más posible es que se encuentren errores que se puedan haber pasado por alto.

2.5.3. Crashes

Los crashes se entienden como la condición en la que una aplicación deja de funcionar de forma esperada, en el caso de Android, cuando una aplicación se congela o deja de responder. Una vez que la aplicación ya está publicada y disponible de forma oficial, es posible ver los reportes de crashes que envían los usuarios. Esto fue implementado por parte de Google el año 2010 [5], a través de la versión de Android 2.2 (Froyo). Si bien esto es bastante útil, la mayoría de las veces no es suficiente, ya que cuando la aplicación deja de funcionar, se le pregunta al usuario si desea enviar este crash al desarrollador y son pocos los usuarios que realizan esta acción.

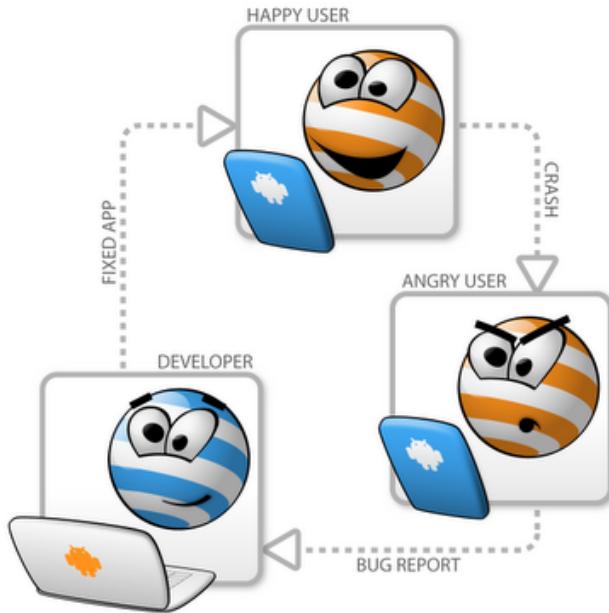


Figura 2.12 – Concepto de Google sobre como el proceso de reportar crashes hacen al usuario feliz. [5]

Este proceso es fundamental para el desarrollo y mantenimiento de una aplicación, pues gracias a los reportes de crashes es posible saber en qué casos es necesario realizar mejoras en el código para dar más estabilidad a la aplicación y que los errores pasados no se vuelvan a cometer. Como se puede ver en la figura 2.12, la idea básica es recibir feedback del usuario para reparar la aplicación y que éste tenga una mejor experiencia la próxima vez.

Capítulo 3

Herramientas Actuales

En este capítulo se detallarán las herramientas más utilizadas para combatir los diversos problemas durante el proceso de desarrollo de aplicaciones Android.

3.1. Herramientas de Testing

3.1.1. Herramientas de Testing Unitario provistas por Android

Android provee un buen framework de testing [12] para hacer pruebas en varios aspectos de la aplicación. Los elementos claves son:

- Los *test suites* (conjuntos de prueba) de Android están basados en JUnit. Se puede usar JUnit puro para testear una clase que no hace llamadas a la API de Android, o las extensiones de JUnit para Android si se desea testear componentes de Android.
- Las extensiones de JUnit proveen clases con tests específicos para componentes. Estas clases entregan métodos para crear *mocking objects* (objetos simulados) y métodos que ayudan a controlar el ciclo de vida de los componentes.
- El SDK (Software Development Kit) de Android también provee herramientas para realizar testing a la UI, como monkeyrunner.

A continuación se revisará en detalle cada una de las herramientas relacionadas a testing del SDK:

JUnit [40]

El testing en Android se basa en JUnit. Actualmente la API de testing soporta JUnit 3. Este requiere que las clases de test hereden de la clase *junit.framework.TestCase*. Además, en JUnit 3 los métodos de testing deben comenzar con el prefijo *test*. También se debe llamar al método *setUp()* para configurar el test y al método *tearDown()* para finalizar el test.

Es una buena práctica, al realizar testing en Android, tener un método llamado *testPreconditions()* que se encargue de corroborar las precondiciones para cada uno de los test. Si este método falla, se sabe inmediatamente que las suposiciones para los otros test no se han cumplido.

Se puede usar la clase *TestCase* de JUnit para hacer testing unitario en una clase que no haga llamadas a la API de Android. *TestCase* es también la clase base para *AndroidTestCase*, que puede ser usada para testear objetos que dependen de Android.

La clase *Assert* de JUnit es usada para mostrar los resultados de los test. Los métodos *assert* comparan los valores que se esperan de un test con los valores reales y se lanza una excepción si la comparación falla. Android también provee una clase para *assertions* que extiende los posibles tipos de comparaciones, y otra clase de *assertions* para testear la UI.

Simulando objetos (Mock objects) [12]

Android entrega clases para crear objetos llamados *mock objects*, que son objetos de sistema simulados como *Context*, *ContentProvider*, *ContentResolver* y *Service*. Algunos tests también proveen objetos simulados de Intent. Se pueden usar estos *mocks* para aislar los tests del resto del sistema y facilitar la inyección de dependencias. Estas clases se encuentran en los paquetes *android.test* y *android.test.mock*.

Por ejemplo se puede usar *MockContext* en vez de *Context*. La clase *RenamingDelegatingContext* entrega las llamadas a un contexto dado y ayuda a la base de datos y a las operaciones con archivos agregando un prefijo a todos los nombres de los archivos. De esta forma se pueden testear componentes sin afectar la base de datos del sistema de archivos de un dispositivo Android.

3.1.2. Herramientas de Testing de UI provistas por Android Instrumentación [12]

La API de testing de Android provee interacciones entre los componentes de Android y el ciclo de vida de la aplicación. Estas interacciones son realizadas a través de la API de Instrumentación, que permite a los tests controlar el ciclo de vida de las *Activities* y los eventos de interacción del usuario.

Normalmente, un componente de Android se ejecuta en un ciclo de vida determinado por el sistema. Por ejemplo, el ciclo de vida de un objeto *Activity* comienza cuando este es iniciado por un *Intent*. El método *onCreate()* es llamado, seguido del método *onResume()*. Cuando el usuario abre otra aplicación, el método *onPause()* es llamado. Si la *Activity* llama al método *finish()*, entonces el método *onDestroy()* también es llamado. La API de Android no provee una forma de llamar estos métodos directamente, pero se puede hacer a través de *Instrumentation*.

Únicamente una clase de test basada en *Instrumentation* permite enviar eventos de teclado o toques de pantalla a la aplicación bajo test. Por ejemplo, se puede testear una llamada al método *getActivity()*, el cual comienza una *Activity* y retorna la *Activity* que está siendo testeada. Después se puede llamar al método *finish()*, seguido por un método *getActivity()* nuevamente, y así se puede testear si la aplicación restaura su estado de forma correcta.

El sistema ejecuta todos los componentes de una aplicación en el mismo proceso. Se puede permitir a algunos componentes, tales como *Content Providers*, ejecutarse en un proceso separado, pero no se puede forzar a una aplicación a ejecutarse en el mismo proceso en el que otra aplicación está ejecutándose.

uiautomator [55]

El SDK de Android contiene la biblioteca uiautomator para testear y ejecutar tests a la interfaz gráfica. Esto fue implementado a partir de la API 16.

Los proyectos de test de uiautomator son proyectos en Java, en donde la biblioteca de JUnit 3, junto con los archivos *uiautomator.jar* y *android.jar* son agregados a la compilación.

Además esta biblioteca provee la clase *UiDevice* que permite la comunicación con el dispositivo, la clase *UiSelector* para buscar elementos en la pantalla y la clase *UiObject* que presenta los elementos de la interfaz. La clase *UiCollection* permite seleccionar un número de elementos de la interfaz gráfica al igual que la clase *UiScrollable* permite hacer scroll en una vista para encontrar un elemento.

uiautomatorviewer [55]

Android también provee la herramienta uiautomatorviewer, que permite analizar la interfaz gráfica de una aplicación. Esta herramienta puede ser usada para encontrar los id, texto o atributos de los elementos de la interfaz.

Esta herramienta permite a la gente que no programa, analizar una aplicación y desarrollar test a través de la biblioteca uiautomator.

Monkey [13]

Monkey es un herramienta de línea de comando que envíá eventos aleatorios a un dispositivo. Se puede restringir a Monkey para que se ejecute sólo en ciertos paquetes, por lo que se le pueden dar instrucciones para testear únicamente una aplicación.

Monkeyrunner [10]

La herramienta Monkeyrunner provee una API en Python para escribir programas que controlen un dispositivo Android o un emulador, fuera del código fuente escrito.

A través de Monkeyrunner se puede hacer un script para realizar un test. Este se ejecuta a través del *adb debug bridge* y permite instalar programas, iniciarlos, controlar los flujos y tomar screenshots.

Para usar Monkeyrunner se debe tener instalado Python en el computador.

Las siguientes clases son las principales:

- MonkeyRunner: permite conectarse con los dispositivos.
- MonkeyDevice: permite instalar y desintalar aplicaciones, como también enviar eventos de teclado y toques en la pantalla a una aplicación.
- MonkeyImage: permite crear, comparar y guardar screenshots.

3.1.3. Herramientas de Testing Unitario de Terceros

EasyMock [57]

EasyMock es un framework para mocking, es decir, para crear objeto simulados. Este puede ser usado en conjunto con JUnit. A continuación se muestra como es la instanciación de un objeto basado en una clase.

```
import static org.easymock.EasyMock.createNiceMock;
...
// ICalcMethod es el objeto que es simulado
ICalcMethod calcMethod = createNiceMock(ICalcMethod.class);
```

El método *createNiceMock()* crea un mock que retorna los valores por defecto para métodos que no están sobreescritos. Además tiene varios métodos que pueden ser usados para configurar el objeto mock. El método *expect()* le dice a Easymock que simule un método con ciertos argumentos. El método *andReturn()* define lo que va a retornar este método. El método *times()* define que tan seguido el objeto va a ser llamado.

Mockito [58]

Mockito es un framework bastante popular que puede ser usado en conjunto con JUnit. Este permite crear y configurar objetos simulados. Además, desde la versión 1.9.5 puede ser usado directamente con los test de Android.

Mockito soporta la creación de objetos simulados con el método estático *mock()*. Este también soporta la creación de objetos basados en la anotación *@Mock*. Si se usan anotaciones, se debe inicializar el objeto simulado con una llamada al método *MockitoAnnotations.initMocks(this)*.

Robolectric [59]

Robolectric es un framework que simula parte del framework de Android contenido en el archivo *android.jar*. Está diseñado para permitir testear aplicaciones de Android

en la JVM (Java Virtual Machine). Este permite ejecutar los test de Android en un entorno de integración continua, sin necesidad de configuraciones extras. Está basado en JUnit 4.

Robolectric reemplaza todas las clases de Android por los llamados *shadow objects*. Si un método es implementado por Robolectric, éste dirige las llamadas al *shadow object*, que se comporta de forma similar a los objetos del SDK de Android. Si un método no es implementado por el *shadow object*, éste simplemente retorna un valor por defecto, como null o 0.

Robolectric soporta el manejo de recursos, como inflar vistas. También puede usarse el *findViewById()* para buscar una vista.

3.1.4. Herramientas de Testing de UI de Terceros

Robotium [56]

Robotium es una extensión del framework de test de Android y fue creado para hacer más fácil los test de interfaz gráfica para las aplicaciones de Android. Robotium hereda de *ActivityInstrumentationTestCase2* y permite definir casos de test a través de las *Activities* de Android.

Los test con Robotium interactúan con la aplicación como test de caja negra, esto quiere decir que únicamente se interactúa con la interfaz y no a través del código interno de la aplicación. La clase principal para testear con Robotium se llama *Solo* y es inicializada en la primera *Activity* que se desea testear.

Espresso [31]

Google lanzó el framework Espresso para testing en Octubre del 2013. Esta es una API para realizar tests de interfaz gráfica, localizando elementos de la UI e interactuando con ellos. A continuación se presentan los componentes principales de Espresso:

- Espresso: Punto de entrada para interactuar con las vistas, a través de los métodos `onView()` y `onData()`. También permite la interacción con métodos que no necesariamente están atados a una vista, como por ejemplo el método `pressBack()`.
- ViewMatchers: Una colección de objetos que implementan la interfaz `Matcher<super View>`. Se puede pasar uno o más de estos objetos al método `onView()` para localizar una vista que actualmente esté dentro de la jerarquía de vistas.
- ViewActions: Una colección de `ViewActions` que pueden pasarse al método `ViewInteraction.perform`, por ejemplo un click.
- ViewAssertions: Una colección de `ViewAssertions` que pueden pasarse al método `ViewInteraction.perform`.

Spoon [49]

Spoon es una herramienta de código abierto para test automatizados que permite ejecutar los test escritos en Java en varios dispositivos al mismo tiempo. Este fue desarrollado por la compañía Square.

La aplicación se ejecuta en base a los test definido en las pruebas de instrumentación. Spoon genera un informe con los resultados a través de un HTML. Cada dispositivo testeado tiene una ficha con los resultados de cada uno de los test.

Además, Spoon permite obtener screenshot de cada estado que se haya definido en la ejecución de los test, los cuales pueden verse en las distintas resoluciones de cada dispositivo en los que se realizaron las pruebas. En el siguiente código se obtienen dos screenshots, uno al inicio y otro después de realizar los respectivos test:

```
Spoon.screenshot(activity, "initial_state");
/* aqui va un codigo de test... */
Spoon.screenshot(activity, "after_login");
```

3.2. Herramientas de Distribución de Versiones

Antes de la publicación de una aplicación, es necesario distribuir usuarios para corroborar que no existen problemas de usabilidad, errores de interfaz o que la aplicación deja de responder ante alguna acción del usuario. Para ello existen herramientas que permiten distribuir de manera segura versiones alpha y beta, para posteriormente poder recibir feedback que permita corregir posibles problemas, o simplemente recibir retroalimentación por parte de testers.

3.2.1. Herramienta de Distribución de Versiones provista por Android

Durante la Conferencia Google I/O de Mayo del año 2013 [41], se llevó a cabo el anuncio de una actualización a la *Google Play Developer Console* [34], que corresponde al lugar en donde el desarrollador sube y administra la versión oficial de su aplicación. Las mejoras consistieron en añadir un servicio de traducción para las aplicaciones, estadísticas relacionadas con ganancias, consejos de optimización, seguimiento de referidos y finalmente la opción de subir versiones alphas y betas, ya que antes de este evento, no existía una forma nativa de realizar esta distribución.

Tal como se ve en la figura 3.1, además de la versión de producción, que corresponde a la versión oficial de una aplicación, existen dos secciones más, una para versiones alpha y otra para versiones beta. El desarrollador tiene la opción de elegir qué usuarios quiere que reciban las versiones experimentales de su aplicación. Esto se realiza creando un grupo en *Google Groups* [33] o una comunidad en *Google+ Communities*[35]. Una vez que el grupo ha sido creado, es responsabilidad del desarrollador agregar o quitar usuarios que recibirán las versiones aún experimentales de la aplicación. Es posible dejar la opción de unirse a estos grupos de forma abierta, para que cualquier usuario que desee tener versiones alpha y beta, puedan obtener una.

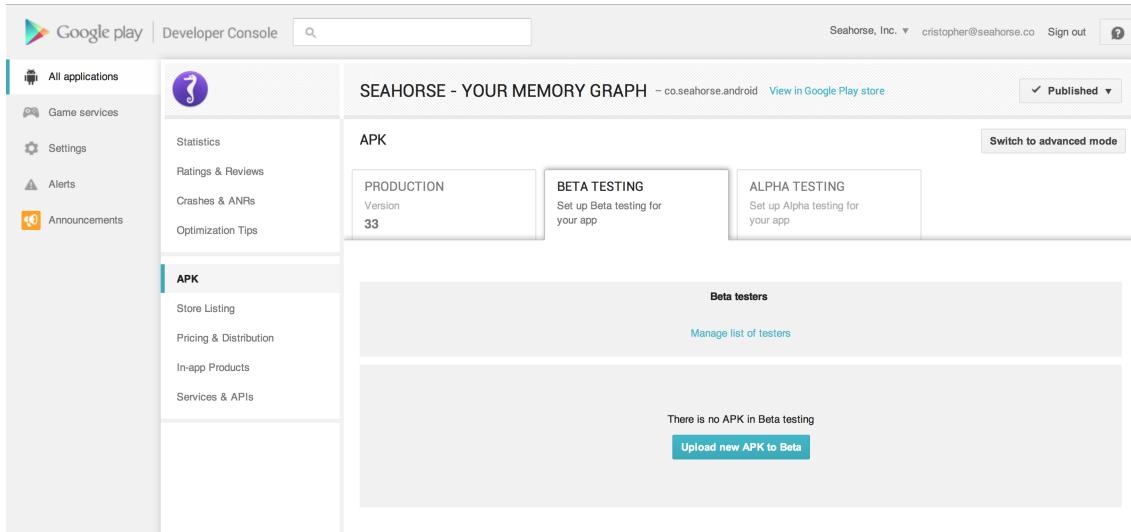


Figura 3.1 – Vista del sistema de versiones alphas y betas provisto por Android. Fuente: Elaboración Propia

3.2.2. Herramienta de Distribución de Versiones de Terceros

Debido a que antes de Mayo del 2013 no existía una forma provista por Google de distribuir aplicaciones de prueba, se contaba con varias herramientas que suplían esta necesidad y que aún siguen haciéndolo. La gran diferencia con la opción que entrega Google es que todas terminan siendo de pago después de algún tiempo o después de una determinada cantidad de usuarios. Además ofrecen otras características complementarias a la distribución de versiones, para entregar mayor valor y diferenciación a sus productos.

HockeyApp [38]

HockeyApp es una plataforma que permite la distribución de versiones beta a múltiples plataformas, entre las que se encuentran Android, iOS, Windows Phone y Mac OS. Esto se complementa con la recolección de reportes de crashes, retroalimentación por parte de los usuarios y estadísticas sobre los testers que están usando la aplicación, como el dispositivo e idioma que tienen.

Para la distribución sólamente es necesario subir el APK de la aplicación e invitar a

los usuarios a través del correo electrónico del tester. Cada vez que se sube una nueva versión es posible enviar un correo notificando a los testers que existe una actualización. Los testers pueden descargar la aplicación accediendo desde el sitio web de HockeyApp [38], como también a través de la aplicación oficial, la cual también puede ser descargada desde su sitio web en la sección de aplicaciones.

Si se desean implementar las otras características que ofrece Hockeyapp, es necesario agregar el SDK de ellos a la aplicación. Esto permitirá tener los reportes de crashes de los testers y otras estadísticas acerca de ellos.

AppBlade [15]

AppBlade es una plataforma que soporta la distribución de versiones beta a Android, iOS y BlackBerry. También cuenta con un sistema de reporte de crashes que notifica al desarrollador de estos errores.

Para la distribución también es necesario subir el APK y comenzar a invitar testers. Luego, los testers deben hacerse una cuenta y descargar la aplicación desde el sitio web de AppBlade [15].

Además existe un SDK para obtener reportes de crashes, obtener feedback por parte de los usuarios y obtener estadísticas.

El servicio permite tener 25 usuarios de forma gratuita. Al sobrepasar este límite existen distintos tipos de planes, cuyos precios varían dependiendo de la calidad del plan, la cantidad de usuarios y la cantidad de meses durante los cuales se tendrá el plan.

Por último AppBlade permite la integración con servicios externos como GitHub [29], Pivotal Tracker [42], HipChat [18], entre otros.

The Beta Family [26]

The Beta Family es una plataforma que permite la distribución de versiones de aplicaciones Android y iOS a testers. A diferencia de las plataformas anteriores, el

desarrollador puede pagar para que un grupo de usuarios de prueba usen su aplicación y reporten los posibles problemas que puedan encontrar.

El servicio también cuenta con una opción gratuita en que 5 testers, de poca reputación dentro de la plataforma, pueden testear una aplicación. Dependiendo de la reputación de los testers, el valor del servicio va variando.

Para el uso de la plataforma es necesario subir el APK y llenar un par de datos básicos sobre la aplicación, así como también preguntas básicas enfocadas en lo que el desarrollador desee.

UserTesting [53]

UserTesting es una plataforma que permite distribuir versiones de aplicaciones Android y iOS a testers. Similar a como funciona The Beta Family, el desarrollador compra una cantidad de créditos, y cada uno de éstos le permite hacer un test distinto.

La gran ventaja con la que cuenta esta plataforma es que el tester debe grabar la pantalla de la aplicación mientras va desarrollando el test, por lo que se pueden observar directamente las reacciones o los posibles problemas que pueda tener el tester al usar la aplicación.

Para el uso de esta plataforma es necesario subir el APK y añadir información que detalle lo que el desarrollador desea que el tester realice. También se pueden agregar preguntas tales como:

- ¿Qué fue lo más frustante de la aplicación?
- ¿Usarías la aplicación?
- ¿Qué nota le pondrías de 1 a 10?

3.3. Herramientas de Reporte de Crashes

Una vez publicada una aplicación, comienza el proceso de correcciones de errores y mejoras. Esto se puede realizar gracias a los reportes de crashes que se reciben cuando la aplicación deja de funcionar correctamente. Es muy importante corregir todos estos errores para dar mayor estabilidad a la aplicación y para ofrecer un mejor producto al usuario. Android cuenta con un sistema de reporte de crashes desde el año 2010[5]. A continuación se revisarán las características con las que cuenta:

3.3.1. Herramienta de Reporte de Crashes provista por Android

Cuando la aplicación ya está publicada, es posible acceder a una sección dentro de la *Google Play Developer Console* [34], titulada *CRASHES & ANRS*. En este sitio se pueden ver los reportes de los últimos 6 meses, y como se muestra en la figura 3.2, es posible aplicar distintos filtros para obtener información más detallada sobre estos reportes. Por ejemplo, se puede filtrar por versiones de sistema operativo o el número de versión de la aplicación.

Tal como se mencionó anteriormente, estos reportes son los que envía el usuario cuando la aplicación deja de funcionar correctamente. En ese momento, el sistema le pregunta al usuario si desea enviar el reporte del crash al desarrollador. Además, el usuario tiene la opción de enviar algún mensaje adicional que pueda ser útil para el desarrollador, como por ejemplo, que al momento del crash se estaba usando la cámara.

Al presionar en el reporte de crash para ver más detalles, se puede ver la hora y el día en el que ocurrió el crash, cuántas veces ha ocurrido y el dispositivo que estaba usando el usuario. También se pueden ver algunas líneas del stack trace del usuario al momento del error. El stack trace es un reporte de todas las acciones que se realizan en cierto punto, durante la ejecución de una aplicación. Esto es muy útil para los desarrolladores, ya que la mayoría de las veces se puede ver en qué línea del código

NAME	REPORTS THIS WEEK	REPORTS TOTAL	LAST REPORTED	HIDE
<code>java.lang.NullPointerException</code> in co.seahorse.android.views.mediacviewer.MetadataMediaA...	0	1	May 3 6:50 AM	<button>Hide</button>
<code>java.lang.ClassNotFoundException</code> in dalvik.system.BaseDexClassLoader.findClass	0	2	Apr 23 2:14 PM	<button>Hide</button>
<code>java.lang.NullPointerException</code> in co.seahorse.android.views.seahorse.SeahorseStartSlide...	0	1	Apr 16 9:56 AM	<button>Hide</button>
<code>java.lang.SecurityException</code> in android.os.Parcel.readException	0	1	Apr 10 7:51 AM	<button>Hide</button>

Figura 3.2 – Vista del sistema de reporte de crashes provisto por Android. Fuente: Elaboración Propria

la aplicación dejó de responder correctamente y qué tipo de error ocurrió. Con esta información es posible comenzar a revisar el código y ver qué problema existe.

3.3.2. Herramientas de Reporte de Crashes de Terceros

Existen varias herramientas que entregan reportes de crashes mucho más completos que los que entrega Google de forma nativa. La mayoría de éstas entregan una versión gratuita con restricciones y es posible pagar para acceder a la versión premium, la cual cuenta con características más especializadas. Además, todos los reportes de crashes son enviados, ya que al momento de instalar la aplicación se piden los permisos necesarios para ello, por lo que el usuario afectado no debe hacer nada. A continuación se listan las más populares:

Crittercism [22]

Crittercism es un sistema muy completo que cuenta con monitoreo, manejo de excepciones, como también reportes de crashes y performance. Actualmente soporta múltiples plataformas, entre las que se encuentran: Android, Android NDK, iOS,

Windows Phone 8 y HTML5.

Como se puede ver en la figura 3.3, se tienen opciones parecidas al reporte de crashes de Google. La gran diferencia está en el detalle de la información, ya que al revisar un crash, se puede ver información muy específica como: nivel de batería, espacio en el disco, espacio en la tarjeta SD, uso de RAM, estabilidad de la red, orientación del dispositivo, idioma del dispositivo, *Activities* que estaban ejecutándose, entre muchos otros puntos.

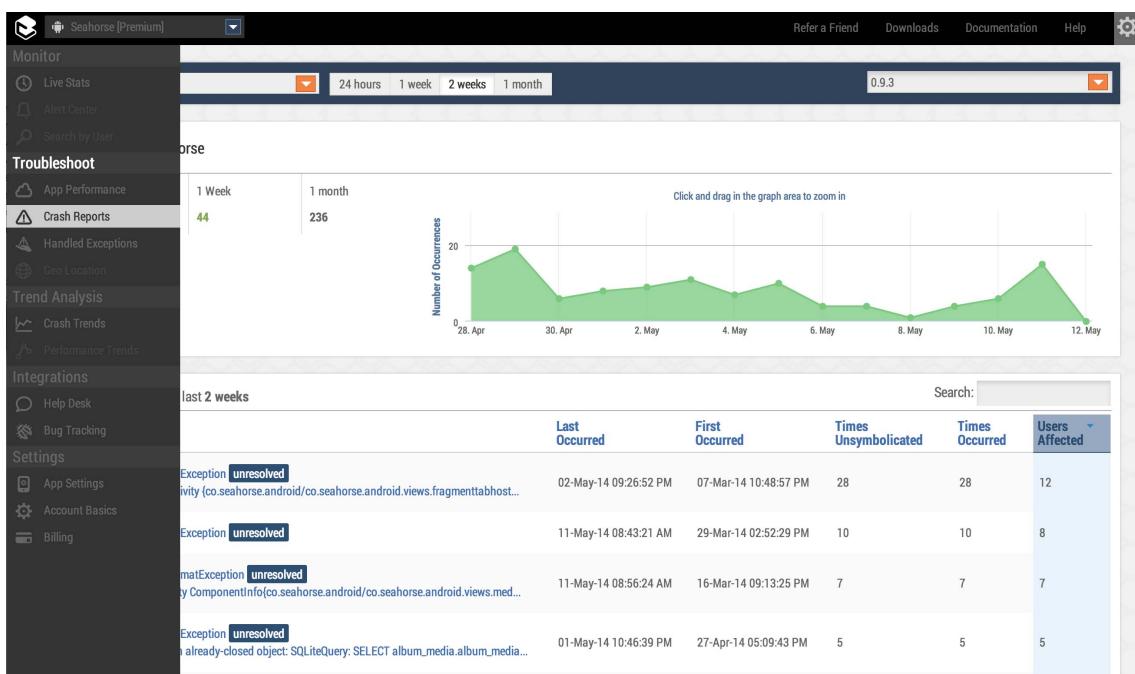


Figura 3.3 – Vista del sistema de reporte de crashes de Crittercism. Fuente: Elaboración Propia

Bugsense [19]

Bugsense también es un sistema bastante completo. Cuenta con monitoreo, reportes de crashes, manejo de excepciones, tendencias de crashes e integración con ACRA. Soporta múltiples plataformas, entre las que se encuentran: Android, iOS, Windows Phone 8 y HTML5.

En la figura 3.4 se puede ver como es el panel con estadísticas que ofrece Bugsense.

Similar a lo ofrecido por Crittercism, es posible filtrar los crashes por versión de la aplicación, como también por versión del sistema operativo.

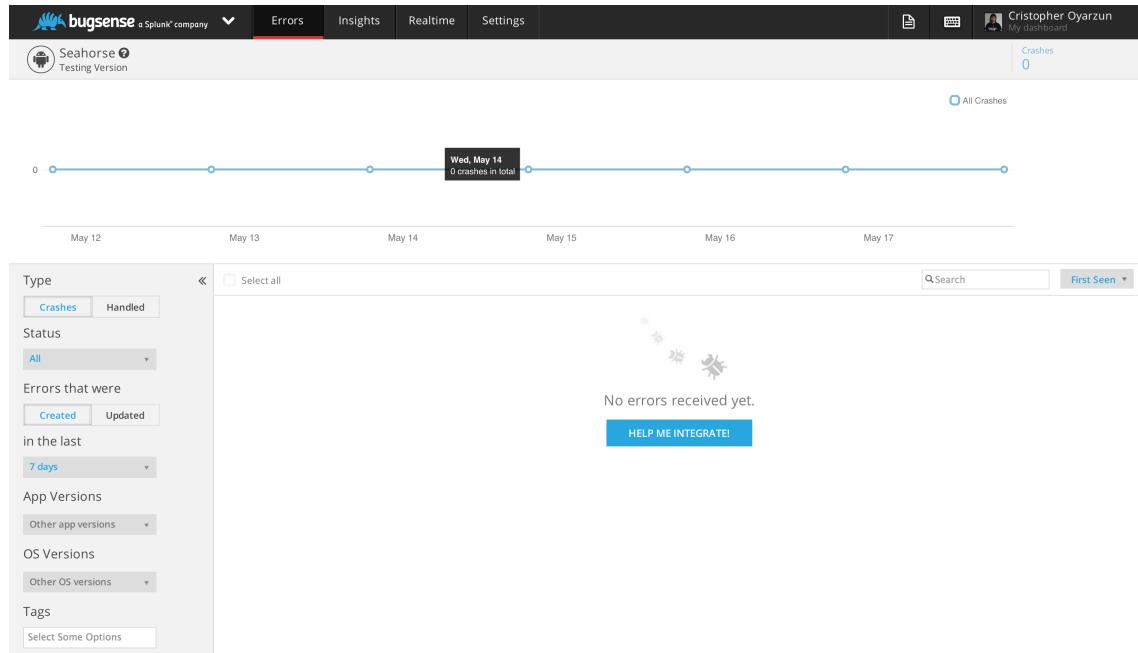


Figura 3.4 – Vista del sistema de reporte de crashes de Bugsense. Fuente: Elaboración Propia

Google Analytics [32]

Google Analytics es otra herramienta que cuenta con reportes de crashes. Si bien, la especialidad de Google Analytics es ofrecer estadísticas y hacer tracking de distintos eventos, también es posible recibir reportes de crashes y excepciones. El gran problema es que los reportes no llegan en tiempo real, ya que la información se actualiza con un día de retraso.

Google Analytics ofrece filtrar los reportes de crashes por versión de la aplicación, versión del sistema operativo, marca del dispositivo y tamaños de pantalla.

ACRA [2]

ACRA es una biblioteca gratuita y de código abierto disponible en Github [1]. Desde la última actualización de Google Forms, el uso de Google Docs como almacenamiento

para los reportes que entregaba ACRA está obsoleto. Ahora es necesario implementar una aplicación web para poder ver los reportes, aunque también es posible asociarlo a otras plataformas como Bugsense o HockeyApp.

En la figura 3.5 se muestra una vista del sistema de reportes integrado con Acralizer [23], ofreciendo filtrar los reportes por versión de Android y versión de la aplicación:

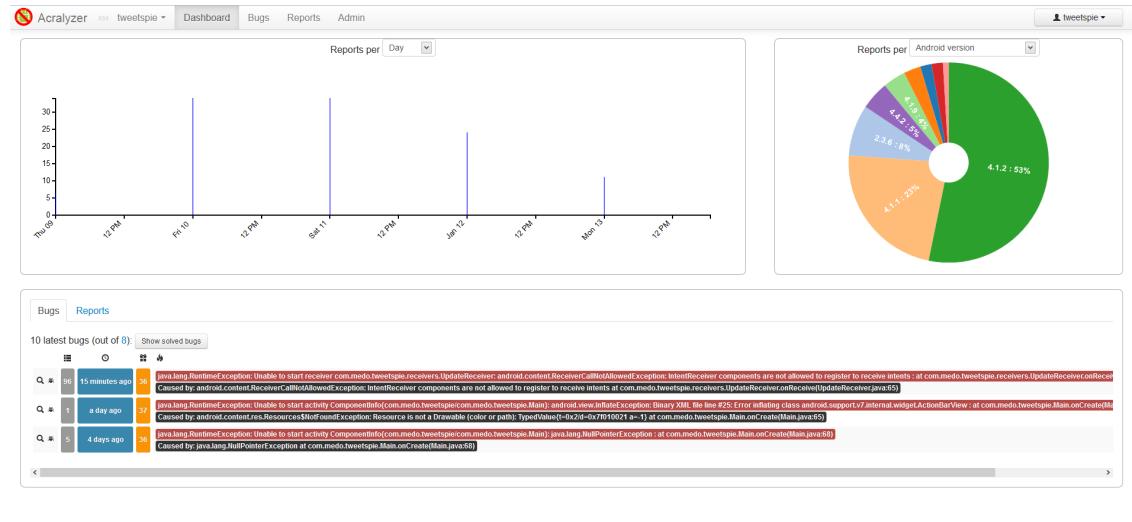


Figura 3.5 – Vista del sistema de reporte de crashes de ACRA. [23]

Capítulo 4

Análisis Comparativo

En la clasificación inicial se tenían herramientas de testing, distribución de versiones y reportes de crashes, y para llevar a cabo estos análisis comparativos, en algunos casos las herramientas se han subdividido en otras categorías. De esta forma es posible realizar comparaciones que arrojen resultados más relevantes y útiles. Las ventajas y desventajas de cada herramienta ayudarán a determinar cuál se ajusta más a las necesidades del desarrollador.

En cada sección se agregará un resumen general, el cual incluirá una tabla comparativa y conclusiones que actuarán como guía para los desarrolladores.

En el siguiente capítulo se llevará a cabo la implementación de las herramientas más destacadas en una aplicación que se encuentra disponible en la tienda oficial de Google.

4.1. Análisis Comparativo entre las Herramientas de Testing Unitario

4.1.1. Framework de Testing de Android

Ventajas

- Acceso al Framework de Android.

Desventajas

- Los test son lentos ya que se ejecuten en un dispositivo real o un emulador con la máquina virtual Dalvik.
- Usa JUnit 3, en vez del más reciente JUnit 4.

Resumen

El Framework de testing de Android es el método oficial que ofrece Google para realizar testing unitario. Primero se carga la aplicación en un dispositivo, y luego los tests basados en JUnit se ejecutan. Debido a que se ejecutan en un dispositivo que cuenta con el sistema operativo Android, se pueden usar todos los métodos que ofrece el framework de Android, por lo que los test son lo más cercano a la realidad. Aunque esto también le juega en contra, ya que compilar la aplicación, subir la aplicación a un dispositivo o emulador, y luego ejecutar los test, es un proceso muy lento.

4.1.2. Robolectric

Ventajas

- Gratuito y de código abierto.
- Los test son rápidos ya que se ejecutan en la máquina virtual de Java, por lo que no se necesita tener un dispositivo conectado.
- Comunidad que realiza mejoras de forma activa, con más de 4 años de desarrollo.

Desventajas

- Existe documentación, pero falta actualizarla.
- No todo puede realizarse con los *mocks* ofrecidos en Robolectric.

Resumen

Robolectric es un proyecto de código abierto que unifica la velocidad del testing unitario con la posibilidad de acceder el framework de Android. Esto se hace implementando todos los *stubs* con *mocks* clases. Es extremadamente rápido en comparación con lo que ofrece Android de forma nativa, ya que los test se ejecutan en una máquina virtual de Java, no siendo necesario compilar todo el proyecto y cargar la aplicación en un dispositivo. Sin embargo, debido al rápido y constante desarrollo que ha tenido Robolectric, en las últimas versiones ha cambiado bastante, dejando obsoleta mucha de la documentación existente.

4.1.3. EasyMock

Ventajas

- Gratuito y de código abierto.
- Testings a través de *mocks objects* (objetos simulados).
- Soporte para Android usando dexmaker [30].

Desventajas

- Es considerada una herramienta de apoyo, aunque puede actuar de forma independiente si es que se desea testear código puro de Java. En caso contrario, tiene que ser usada en conjunto con Robolectric o el Framework de testing de Android.
- Utiliza el viejo modelo de *Record/Replay*.

Resumen

Easymock es uno de los clásicos framework de mocking y cuenta con buena documentación. Utiliza el modelo de *Record/Replay*, en donde el desarrollador crea el mock, conecta el mock con el objeto que será testeado, invoca miembros del mock durante

la fase de *record*, luego se cambia al mock al estado de *replay*, se ejecuta el test y se verifican las invocaciones.

4.1.4. Mockito

Ventajas

- Gratuito y de código abierto.
- Testing a través de *mocks objects* (objetos simulados).
- Soporte para Android usando dexmaker [30].
- Utiliza el modelo de *Stub/Verify*
- Mayor legibilidad en los test.

Desventajas

- Es considerada una herramienta de apoyo, aunque puede actuar de forma independiente si es que se desea testear código puro de Java. En caso contrario, tiene que ser usada en conjunto con Robolectric o el Framework de testing de Android.

Resumen

Mockito en un comienzo estaba basado en EasyMock, por lo que se puede apreciar que existen varias similitudes. Aunque han realizado mejoras a nivel sintáctico, para hacerlo más entendible. El concepto detrás de los test con Mockito es el de stubbing, ejecutar y verificar, es decir, programar un comportamiento, ejecutar las llamadas y verificarlas. Este modelo es una evolución del antiguo *Record/Replay*.

4.1.5. Resumen General

Las dos grandes herramientas de testing unitario son Robolectric y el Framework de testing de Android. Hay que mencionar que tanto EasyMock como Mockito son sólo

herramientas de apoyo que están más ligadas a Java que a Android, por lo que la decisión de usar alguna de estas herramienta u otro framework de mocking dependerá del grado de afinidad que tenga el desarrollador con este tipo herramientas, ya que son absolutamente prescindibles en la mayoría de los casos. La ventaja que ofrece Robolectric sobre el Framework de testing de Android es clara, ya que sin la necesidad de ejecutar los tests en un dispositivo o emulador, hace que estos sean mucho más rápidos. A continuación se presentan los parámetros de comparación que se considerarán:

- Gratuito: No se debe pagar nada para acceder a todas sus características.
- Código abierto: Se tiene acceso al código fuente.
- Madurez: Tiempo de desarrollo y estabilidad que tiene la herramienta.
- Duración: Cantidad de tiempo que se demoran en ejecutar los tests.
- Documentación: Calidad de la documentación relacionada con la herramienta.
- Usabilidad: Nivel de complejidad en integrar y usar la herramienta en una aplicación.

En las siguientes tablas se evalúa el comportamiento de cada herramienta con respecto a los parámetros recién definidos.

Herramienta	Gratuito	Código abierto	Madurez	Duración
Framework Testing Android	Si	Si	Alta	Alta
Robolectric	Si	Si	Alta	Baja

Tabla 4.1 – a) Tabla comparativa entre herramientas que se enfocan en el testing unitario en Android.

Herramienta	Documentación	Usabilidad
Framework Testing Android	Buena	Fácil
Robolectric	Media	Fácil

Tabla 4.2 – b) Tabla comparativa entre herramientas que se enfocan en el testing unitario en Android.

4.2. Análisis Comparativo entre las Herramientas de Testing de UI

4.2.1. uiautomator

Ventajas

- Gratuito y desarrollado por Google.
- Los test son independientes del proceso en el que la aplicación testeada funciona. Esto permite realizar tests en que se llamen otras *Activities*, como por ejemplo la cámara.

Desventajas

- No tiene compatibilidad con las versiones más antiguas, ya que funciona desde la API 16 en adelante (Jelly Bean).
- Las instancias de los objetos de la UI se pueden obtener a través de los ID's sólamente desde la API 18 en adelante. En el resto de los casos, se obtienen a través del texto que poseen, por lo que si estos textos cambian, los test necesitarán una refactorización.
- No se puede obtener la *Activity* actual.
- Falta de documentación.

Resumen

Una de las principales ventajas con las que cuenta uiautomator, es que los test se ejecutan en un proceso independiente de la aplicación, por lo que se pueden realizar acciones que no son permitidas en otros frameworks, por ejemplo es posible ejecutar iniciar la cámara, o desactivar el WIFI, o activar el GPS. Si bien, esta herramienta fue lanzada el 14 de Noviembre del 2012 [7], el nivel de documentación con la que cuenta es muy bajo.

4.2.2. Robotium

Ventajas

- Gratuito y de código abierto.
- Fácil uso.
- Más de 4 años de desarrollo, por lo que es una biblioteca estable que cuenta con una comunidad activa.
- Cuenta con Robotium Recorder, para capturar videos de los test.

Desventajas

- No se sincroniza con el *thread* de la UI, por lo que en algunos casos es necesario agregar mecanismos para esperar a este *thread*, por ejemplo un sleep de 100 milisegundos.

Resumen

Robotium ha sido la herramienta más popular de testing funcional en Android, ya que lleva más de 4 años de desarrollo, con actualizaciones prácticamente de forma mensual. Existe una gran comunidad y variados ejemplos [47] que facilitan la tarea a los desarrolladores que se están iniciando en el testing, lo cual no ocurre con otras

herramientas. Por último, en Enero de este año [48], el fundador de Robotium lanzó una herramienta complementaria llamada Robotium Recorder, la cual permite grabar los tests que se realizan, aunque ésta tiene un costo de 295 dólares por una licencia anual.

4.2.3. Espresso

Ventajas

- Creado por Google basándose en Robotium.
- Fácil uso.
- Se sincroniza con el *thread* de la UI.
- Los tests corren tres veces más rápido que en Robotium *benchmarks* [27] .

Desventajas

- Falta más documentación debido a que lleva sólo un año desde que se presentó en la conferencia Google I/O del 2013.

Resumen

Es la última herramienta de testing que ha lanzado Google. Surgió por una necesidad dentro del equipo de Google de mejorar la forma en que se realizaba el testing funcional, ya que hasta ese momento ellos usaban Robotium. El desarrollo de Espresso fue influenciado en gran medida por Robotium, con la diferencia fundamental de que los tests se ejecutan sincronizados con el *thread* de la UI, lo que le da una mayor estabilidad. Además, basándose en los *benchmarks* [27] que lo comparan con Robotium, Espresso es tres veces más rápido.

4.2.4. Resumen General

Las tres herramientas mencionadas anteriormente son excelentes y cada una de ellas cuenta con ventajas sobre el resto. Por un lado está uiautomator, que es la única que permite interactuar con otras aplicaciones ya que los tests no están ligados a una *Activity*. Mientras que Robotium y Espresso opacan a uiautomator en prácticamente todo el resto de características, en especial, en el fácil uso. Todo depende de para qué se deseen usar estas herramientas, pues todas son excelentes para realizar tests de caja negra, en donde no se tiene acceso o simplemente no se revisa el código fuente de la aplicación a la cual se quieren realizar las pruebas. Por último cabe mencionar que Espresso es la más rápida de las tres, y que Robotium cuenta con la mejor documentación. A continuación se presentan los parámetros de comparación que se considerarán:

- Gratuito: No se debe pagar nada para acceder a todas sus características.
- Código abierto: Se tiene acceso al código fuente.
- Madurez: Tiempo de desarrollo y estabilidad que tiene la herramienta.
- Duración: Cantidad de tiempo que se demoran en ejecutar los tests.
- Documentación: Calidad de la documentación relacionada con la herramienta.
- Usabilidad: Nivel de complejidad en integrar y usar la herramienta en la aplicación.
- Sincronizado con *UI thread*: Los test se sincronizan con el *UI thread*.

En las siguientes tablas se evalúa el comportamiento de cada herramienta con respecto a los parámetros recién definidos.

Herramienta	Gratis	Código Abierto	Madurez	Duración
uiautomator	Si	No	Media	Alta
Robotium	Si	Si	Alta	Media
Espresso	Si	Si	Media	Baja

Tabla 4.3 – a) Tabla comparativa entre herramientas que se enfocan en el testing funcional en Android.

Herramienta	Documentación	Usabilidad	Sincronizado con <i>UI thread</i>
uiautomator	Media	Media	No
Robotium	Buena	Fácil	No
Espresso	Media	Media	Si

Tabla 4.4 – b) Tabla comparativa entre herramientas que se enfocan en el testing funcional en Android.

4.3. Otras Herramientas de Testing

4.3.1. Monkey

Ventajas

- Excelente herramienta para llevar a cabo tests de estrés.

Desventajas

- No se pueden llevar a cabo tests más complejos.

Resumen

Esta herramienta es perfecta para realizar tests de estrés. Estos consisten en enviar una gran cantidad de eventos pseudo-aleatorios a la aplicación, como clicks, toques, entre otras cosas, para verificar que la aplicación responde de forma correcta, sin crashes y sin congelarse.

4.3.2. Monkeyrunner

Ventajas

- Provee una API en Python para escribir tests, fuera del código fuente de la aplicación.

- Permite ejecutar los tests en varios dispositivos a la vez, obtener screenshots y compararlos.
- Bien documentada.

Desventajas

- Se interactúa con el dispositivo y no con la aplicación, por lo que si se desea presionar un botón dentro de la aplicación, se le deben dar las coordenadas precisas (X,Y) para presionarlo, a diferencia de otras herramientas que permiten obtener la ubicación del botón a través del ID o del texto.

Resumen

Cabe mencionar que Monkeyrunner no está relacionado con la herramienta vista anteriormente Monkey. Con Monkeyrunner es posible escribir un script en Python que instale una aplicación en un dispositivo, presionar los botones físicos del dispositivo, abrir el teclado, tomar screenshots de la interfaz y comparar estos screenshots. Está principalmente enfocado en el testing de la UI, aunque también puede ser usado para otros propósitos. Por ejemplo, en el desarrollo de una aplicación podría ser necesario que un dispositivo cuente con una gran cantidad de contactos, y la creación automatizada de contactos se puede llevar a cabo a través de un script ejecutado con Monkeyrunner.

4.3.3. Spoon

Ventajas

- Puede ser usada en conjunto con otras herramientas de testing de UI como Robotium y Espresso.
- Permite ejecutar los tests en varios dispositivos a la vez y obtener screenshots.

Desventajas

- Podría incluir video.

Resumen

Esta es una herramienta perfecta, que puede ser usada por si sola, a través de las herramientas de instrumentación que ofrece Android, o en conjunto con otras herramientas como Robotium y Espresso. Spoon ejecuta los tests en todos los dispositivos conectados al computador, incluyendo los emuladores que pueden estar ejecutándose. Al finalizar el test, se obtienen los screenshots de la aplicación durante la ejecución del test, y en cada una de las pantallas. Esto es muy útil al momento de entender por qué pueden haber fallado los test, cómo también para ver simultáneamente como se ve la aplicación en varios dispositivos.

4.4. Análisis Comparativo entre las Herramientas de Distribución de Versiones

4.4.1. Google Play Console

Ventajas

- Gratuito.
- Permite utilizar la misma tienda oficial de aplicaciones de Google para distribuir versiones betas.
- Actualización de versiones automática.
- Es posible llegar a mucha más gente, ya que se puede dejar abierta la opción de recibir versiones betas.

Desventajas

- Los usuarios testers deben ser miembros de un grupo en *Google Groups* [33] o una comunidad en *Google+ Communities*[35].
- Al ser una herramienta nativa de Android, no es multiplataforma.
- El feedback de los usuarios es a través del grupo o comunidad a la que se tuvo que unir.

Resumen

Esta es una excelente herramienta para cualquier desarrollador. Al utilizar la tienda oficial de Google, genera mucha más confianza a los usuarios testers. Además, al ser gratuita, es una de las mejores opciones con las que se cuenta, ya que es mucho más fácil llegar a más usuarios y tener un costo 0. Si se cuenta con una aplicación multiplataforma, tal vez se quiera tener a todos los usuarios de prueba en un mismo servicio por lo que eso podría ser un problema. Por último, la comunicación con los

usuarios testers no es tan directa como en otros servicios, ya que deben hacer llegar sus inquietudes a través del grupo o comunidad a la que tuvieron que unirse.

4.4.2. HockeyApp

Ventajas

- Soporte para Android, iOS, Windows Phone y Mac OS.
- Usuarios pueden enviar feedback sobre la versión de forma más directa.
- Se puede implementar el SDK para recibir reportes de crashes.
- Enfocado en grupos pequeños de testers.

Desventajas

- De pago, 30 días de prueba.
- Los usuarios testers deben crearse una cuenta en el sitio de HockeyApp [38].

Resumen

Es una muy buena herramienta para una aplicación que es multiplataforma. De esta forma, es posible tener a todos los usuarios de Android, iOS, Windows Phone y Mac registrados en el mismo servicio. Especial para grupos pequeños de testers, con los cuales se puede tener una comunicación directa a través de la plataforma. Los precios de los planes van desde los 10 dólares mensuales, en que se permite tener hasta 5 aplicaciones, hasta los 120 dólares, en que se permiten 120 aplicaciones. Además cuenta con un SDK que incluye un servicio para la recepción de crashes, aunque a éste le falta madurez en comparación con las herramientas especializadas en ello.

4.4.3. AppBlade

Ventajas

- Soporte para Android, iOS y Blackberry.

- Usuarios pueden enviar feedback sobre la versión.
- Se puede implementar el SDK para recibir reportes de crashes.
- Enfocado en grupos pequeños de testers.
- Integración con servicios externos como GitHub [29], Pivotal Tracker [42], HipChat [18], entre otros.

Desventajas

- De pago después de incluir a 25 usuarios.
- Los usuarios testers deben crearse una cuenta en el sitio de AppBlade [15].
- Herramienta aún poco madura.

Resumen

Si bien, ofrece prácticamente las mismas características que HockeyApp, le falta aún madurez. La interfaz del sitio web no genera la misma confianza y los correos con invitaciones muchas veces llegan con retraso. Las ventajas que tiene es que es multiplataforma, incluyendo a Blackberry entre sus opciones. También está enfocado en grupos pequeños de testers, con los cuales se pueda tener una comunicación más directa, y mientras sean menos de 25 usuarios, el servicio es gratuito. Se ve como una plataforma prometedora.

4.4.4. The Beta Family

Ventajas

- Soporte para Android y iOS.
- Combinan la distribución de versiones beta con test para usuarios.
- Enfocado en grupos pequeños de testers.

- Algunos usuarios reciben dinero por los test que realizan, por lo que el feedback que se entrega es más confiable.

Desventajas

- De pago, se ofrece una opción gratuita, pero es necesario anexar una tarjeta de crédito y los testers son usuarios de poca o nula reputación.
- Cuenta con un SDK sólo para iOS que permite grabar la pantalla del usuario y al usuario.

Resumen

Esta plataforma no sólo se enfoca en la distribución de versiones beta, sino que también busca realizar tests con los usuarios, incentivándolos con pagos. Estos tests constan de una serie de tareas que debe desarrollar el usuario, para corroborar que la aplicación está funcionando correctamente y que no hay problemas de usabilidad. Una vez desarrolladas estas tareas, el usuario debe responder algunas preguntas. Además, si se implementa su SDK, es posible grabar la pantalla del usuario y al usuario mientras realiza las pruebas, el problema de esta característica es que sólo está disponible para iOS. Por último, los desarrolladores pueden optar por la opción gratuita, en la que pueden crear test, pero éstos no tendrán recompensas en dinero para los testers, por lo que se deduce que ellos no estarán tan interesados en el producto. En la versión de pago, con la que se tiene acceso a testers de más confianza, se pagan 16.15 dólares mensuales, de los cuales 10 dólares van destinados al tester.

4.4.5. UserTesting

Ventajas

- Soporte para Android, iOS y sitios web.
- Combinan la distribución de versiones beta con test para usuarios.

- Enfocado en grupos pequeños de testers.
- Todos los usuarios reciben dinero por los test que realizan, por lo que el feedback es muy confiable.
- Todos los usuarios que realizan los tests, suben un video en el que graban la pantalla de su dispositivo.
- Servicio de mucha reputación, grandes empresas como Google, Apple, Microsoft, Facebook, Twitter, Dell, entre otras, lo usan.

Desventajas

- De pago, aunque es posible solicitar una prueba gratuita.

Resumen

Al igual que The Beta Family esta plataforma se enfoca en la distribución de versiones beta y en tests con los usuarios. Estos tests constan de una serie de tareas que debe desarrollar el usuario, mientras graba la pantalla del dispositivo para corroborar que la aplicación está funcionando correctamente, que no hay problemas de usabilidad y escuchar las distintas reacciones del tester al cumplir con las tareas encomendadas. Al finalizar, el usuario debe responder algunas preguntas sobre la experiencia realizada. En esta plataforma todos los tests son grabados por los usuarios y normalmente el tiempo que transcurre entre que el desarrollador crea un nuevo test y los usuarios lo realizan, no supera una hora. Debido a la alta reputación del servicio, el precio es uno de los posibles frenos dependiendo del presupuesto con el que se cuenta, ya que por cada usuario se pagan 50 dólares mensuales, por lo que es necesario enfocar bien los test que se desean realizar, y tener mucha claridad de lo que se quiere medir con cada tarea.

4.4.6. Resumen General

Es necesario hacer una distinción entre las herramientas que se enfocan en distribución y en las que además ofrecen la opción de realizar tests no automatizados, con usuarios reales, es por ello que la comparación se divide en dos tablas distintas. El concepto de madurez se refiere al nivel de desarrollo con el que cuenta la plataforma. Esto es muy importante al momento de elegir un servicio, ya que si no es maduro, robusto y de calidad, es muy probable que genere una frustración entre los usuarios, antes que éstos puedan usar la aplicación, lo cual puede influenciar la percepción que tendrán al momento de entregar feedback o de realizar los testing. A continuación se presentan los parámetros de comparación que se considerarán:

- Gratuito: No se debe pagar nada para acceder a todas sus características.
- Precio: Valor del plan más económico ofrecido por la herramienta de forma mensual.
- Madurez: Tiempo de desarrollo y estabilidad que tiene la herramienta.
- Multiplataforma: Soporte para más de una plataforma, no sólo Android.
- Feedback: Los usuarios de prueba pueden enviar retroalimentación al desarrollador.
- Distribución: Es masiva si los usuarios pueden acceder por su cuenta, sin invitaciones del desarrollador. Es limitada si el desarrollador es el que tiene que invitar a los usuarios.
- Testing incluye video: Además de realizar las tareas y responder las preguntas solicitadas por el desarrollador, el usuario adjunta un video de todo el proceso.

En la siguientes tablas se evalúa el comportamiento de cada herramienta con respecto a los parámetros recién definidos.

Herramienta	Gratis	Precio	Madurez
Google Play Console	Si	0 USD	Alta
HockeyApp	No	10 USD por usuarios ilimitados	Alta
AppBlade	No	1 USD por cada usuario	Baja

Tabla 4.5 – a) Tabla comparativa entre herramientas que se enfocan en la Distribución de Versiones en Android.

Herramienta	Feedback	Multiplataforma	Distribución
Google Play Console	No	No	Masiva
HockeyApp	Si	Si	Masiva
AppBlade	Si	Si	Masiva

Tabla 4.6 – b) Tabla comparativa entre herramientas que se enfocan en la Distribución de Versiones en Android.

La inclusión de un video en el proceso de testing es muy útil para poder ver las reacciones del usuario y la interacción que tienen con la aplicación. Además, como en Android existen muchos dispositivos, lo que el usuario ve puede variar dependiendo del tamaño de la pantalla, el sistema operativo que tiene y el fabricante del dispositivo, por lo que visualizar la aplicación en otros dispositivos es fundamental para entregar una buena experiencia a la mayor cantidad de usuarios posibles. A continuación se presenta la tabla correspondiente a las herramientas que combinan la distribución y el testing:

Herramienta	Gratis	Precio	Madurez	Multiplataforma
The Beta Family	No	16.15 USD por usuario	Baja	Si
UserTesting	No	50 USD por usuario	Alta	Si

Tabla 4.7 – a) Tabla comparativa entre herramientas que se enfocan tanto en Distribución, como Testing de Versiones en Android.

Herramienta	Testing incluye video	Distribución
The Beta Family	No	Limitada
UserTesting	Si	Limitada

Tabla 4.8 – b) Tabla comparativa entre herramientas que se enfocan tanto en Distribución, como Testing de Versiones en Android.

4.5. Análisis Comparativo entre las Herramientas de Reporte de Crashes

4.5.1. Google Play Console

Ventajas

- Gratuito.
- Permite utilizar la misma tienda oficial de aplicaciones de Google para recepción de crashes sin necesidad de implementaciones adicionales.
- Se reciben los ANRs (*Application Not Responding*).
- Los reportes de crashes vienen con un mensaje por parte del usuario.

Desventajas

- Muy pocos usuarios mandan reportes de crashes.
- No es posible recibir reportes de las excepciones.

Resumen

Al utilizar la tienda oficial de Google y no necesitar de implementaciones adicionales, esta herramienta actúa como apoyo a otros servicios, ya que para asegurar la calidad de un producto, es completamente necesario recibir los reportes de crashes de cada uno de los usuarios. Es bastante útil que los usuarios puedan incluir un mensaje en sus reportes, aunque la gran desventaja sigue siendo que depende del usuario si desea enviar el reporte al momento que la aplicación deja de funcionar correctamente. Para ejemplificar esto, es posible que a través de Google Play Console se reciba sólo un reporte de crash de un usuario, pero que a través de otras herramientas se vean más de 100 reportes distintos, ya que 99 usuarios no quisieron enviar sus reportes.

4.5.2. Crittercism

Ventajas

- Se reciben todos los crashes de los usuarios.
- Se recibe información muy detallada en cada reporte de crash como: nivel de batería, espacio en el disco, espacio en la tarjeta SD, uso de RAM, estabilidad de la red, orientación del dispositivo, idioma del dispositivo, entre otros.
- Es posible recibir las excepciones que el desarrollador desee.
- Es posible recibir notificaciones al correo sobre los crashes.
- Se puede integrar con otros servicios como GitHub [29], Pivotal Tracker [42], entre otros.
- Soporte para Android, Android NDK, iOS, Windows Phone 8, HTML5.

Desventajas

- De pago.

Resumen

Crittercism es una de las plataformas más consolidadas y con mayor reputación, ya que no sólo se enfoca en los reportes de crashes, sino que también funciona como un servicio de monitoreo de distintas métricas que ayudan a construir una mejor aplicación. La gran ventaja que comparte junto al resto de las plataformas es que la decisión de enviar o no enviar un reporte no dependa del usuario, ya que todos y cada uno de los reportes quedan a disposición del desarrollador. Además permite la integración con otros servicios que permiten asignar estos crashes a un desarrollador para que los resuelva y no vuelvan a ocurrir. El nivel de detalle en cada uno de los reportes de crashes es muy útil para que el desarrollador tenga más indicios que lo

ayuden a resolver el problema. Además, es posible recibir correos con los reportes de crashes, siendo éstos agrupados de forma inteligente para no generar SPAM en el correo del desarrollador. Por último, no sólo se enfoca en los reportes de crashes, también en la performance general de la aplicación, ya que también se puede medir la latencia y la tasa de error que tiene la aplicación al comunicarse con los servidores que le entregan información.

4.5.3. Bugsense

Ventajas

- Se reciben todos los crashes de los usuarios.
- Se recibe información útil en cada reporte de crash.
- Es posible recibir las excepciones que el desarrollador desee.
- Es posible recibir notificaciones al correo sobre los crashes.
- Se puede integrar con otros servicios como GitHub [29], Pivotal Tracker [42], entre otros.
- Integración con ACRA.
- Soporte para Android, iOS, Windows Phone 7-8, Windows 8, HTML5.

Desventajas

- De pago.

Resumen

Bugsense es una plataforma que al igual que Crittercism, cuenta con una buena reputación. La gran diferencia es la cantidad de detalle en cada uno de los reportes. Si bien Bugsense entrega información útil como la versión de la aplicación, modelo del

teléfono, versión del sistema operativo, esto no se acerca a lo ofrecido por Crittercism. Cabe mencionar que Bugsense ha presentado en Marzo de este año varias mejoras en su servicio [20], tales como la tasa de crashes que se han recibido los últimos 7 días y la página que muestra los crashes en tiempo real. Una de las características únicas es que se puede integrar con ACRA, una biblioteca de código abierto que también permite enviar los reportes de crashes de los usuarios.

4.5.4. Google Analytics

Ventajas

- Gratuito.
- Se reciben todos los crashes de los usuarios.
- Se recibe información útil en cada reporte de crash.
- Es posible recibir las excepciones que el desarrollador deseé.
- Excelente en métricas y estadísticas.
- Soporte para Android, iOS, Web.

Desventajas

- Los reportes de crashes se muestran en la plataforma con un día de retraso.

Resumen

Google Analytics es una herramienta gratuita que está más ligada al tracking de eventos, estadísticas y flujos entre las pantallas, pero también cuenta con un sistema de reporte de crashes. Por ejemplo, es posible medir tiempos de respuesta, para saber exactamente cuánto tiempo le toma a la aplicación mostrar una vista en específico. Esto ayuda a encontrar problemas de rendimiento dentro de la aplicación. Además se puede estudiar el comportamiento de los usuarios, analizando cuáles son las pantallas que los

usuarios más usan dentro de la aplicación. La gran desventaja es que la información relacionada a los reportes de crashes se actualiza en la plataforma con un día de retraso, por lo que muchas veces el desarrollador se puede sentir a ciegas después de lanzar una nueva versión de su aplicación, ya que no sabe lo que está pasando hasta después de un día.

4.5.5. ACRA

Ventajas

- Gratuito y de código abierto.
- Se reciben todos los crashes de los usuarios.
- El desarrollador decide qué información quiere recibir en cada reporte de crash.
- Es posible recibir las excepciones que el desarrollador deseé.

Desventajas

- Es necesario implementar una aplicación web para poder ver los reportes de crashes.

Resumen

La gran ventaja que ofrece ACRA sobre el resto es que es una biblioteca de código abierto, por lo que si no nos gusta algo podemos cambiarlo, o podemos agregar otras funcionalidades. Esto le juega a favor y en contra ya que también es necesario que el desarrollador implemente una aplicación web para la visualización de los reportes. Antes, el almacenamiento de los reportes de crashes se realizaba a través de Google Docs, pero desde la última actualización de Google Forms, el uso de Google Docs como almacenamiento para los reportes que entregaba ACRA quedó obsoleto.

4.5.6. Resumen General

Google Play Console es una buena herramienta de inicio, pero es absolutamente necesaria la implementación de una herramienta que asegure la recepción de todos los reportes de crashes. Crittercism y Bugsense son dos plataformas excepcionales, que permiten realizar esta tarea, aunque son de pago. ACRA cuenta con la ventaja de ser de código abierto, pero el simple hecho que el desarrollador deba implementar el servicio web para la visualización de los crashes es un paso adicional y la descarta como una alternativa de fácil integración. Por otro lado, Google Analytics ofrece variadas estadísticas e información extremadamente útil, que puede servir para mejorar problemas de performance y usabilidad, pero su punto fuerte no son los reportes de crashes, ya que tener un día de retraso significa estar un día completo sin saber que está pasando con la aplicación. A continuación se presenta una tabla que intenta simplificar las ventajas y desventajas que posee cada herramienta:

- Gratuito: No se debe pagar nada para acceder a todas sus características.
- Precio: Valor del plan más económico ofrecido por la herramienta de forma mensual.
- Madurez: Tiempo de desarrollo y estabilidad que tiene la herramienta.
- Multiplataforma: Soporte para más de una plataforma, no sólo Android.
- Usabilidad: Nivel de complejidad en integrar y usar la herramienta con la aplicación.
- Calidad del reporte: Nivel de detalle en cada reporte de crash.

En la siguientes tablas se evalúa el comportamiento de cada herramienta con respecto a los parámetros recién definidos.

Herramienta	Gratis	Precio	Multiplataforma
Google Play Console	Si	0 USD	No
Crittercism	No	24 USD	Si
Bugsense	No	19 USD	Si
Google Analytics	Si	0 USD	Si
ACRA	Si	0 USD	No

Tabla 4.9 – a) Tabla comparativa entre herramientas que se enfocan en los Reportes de Crashes.

Herramienta	Madurez	Usabilidad	Calidad del reporte
Google Play Console	Alta	Fácil	Media
Crittercism	Alta	Fácil	Alta
Bugsense	Alta	Fácil	Media
Google Analytics	Alta	Fácil	Media
ACRA	Media	Difícil	Media

Tabla 4.10 – b) Tabla comparativa entre herramientas que se enfocan en los Reportes de Crashes.

Capítulo 5

Implementación

En este capítulo se realizará la implementación de las herramientas más destacadas en base a los análisis previos. La integración de estas herramientas se llevarán a cabo en un entorno real de desarrollo, específicamente en la aplicación **Seahorse** [39]. Esta aplicación está enfocada en la creación colaborativa y privada de álbumes de fotos y videos. En la figura 5.1 es posible ver algunas de las pantallas de esta aplicación.

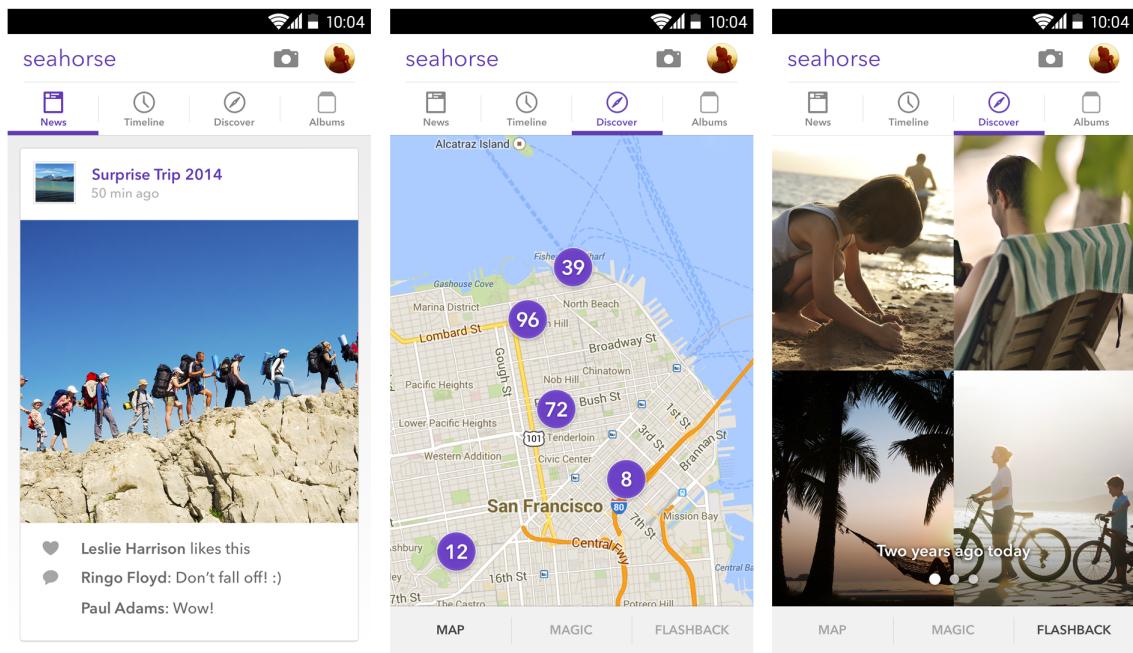


Figura 5.1 – Vistas de la aplicación Seahorse. Fuente: Elaboración Propia

Las herramientas implementadas son las que más se ajustaban a las necesidades

y a los recursos con los que cuenta Seahorse, por lo que fácilmente este conjunto de herramientas puede diferir en relación a las implementaciones que otros desarrolladores harían en sus aplicaciones.

5.1. Implementación de Herramientas de Testing

5.1.1. Robolectric

Para la implementación de Robolectric es necesario seguir los pasos de instalación que se detallan en su sitio oficial [46]. En Seahorse se usa Gradle [36] para automatizar la construcción y compilación el proyecto, aunque también es posible usar Ant[?] o Maven [?]. Para agregar Robolectric a un proyecto configurado con Gradle es necesario agregar esta línea al archivo *build.gradle* del proyecto:

```
dependencies {  
    ...  
    androidTestCompile 'org.robolectric:robolectric:2.3'  
}
```

Para más detalles sobre cómo realizar la instalación con Android Studio, se puede seguir el tutorial que presenta en su blog [28] uno de los desarrolladores de Google.

El test que se llevó a cabo consiste en verificar que el flujo inicial de la aplicación se realice de forma correcta. Básicamente se testeá que el usuario pueda acceder sin problemas a las pantallas de login y de registro. En la figura 5.2 se muestra el flujo que se someterá a prueba, el cual consta de 3 tareas básicas:

- Abrir Seahorse.
- Presionar el botón que dice *Log In* y mostrar un menú con opciones.
- Presionar el botón que dice *Sign in with email* y llevar al usuario a la vista de Login.

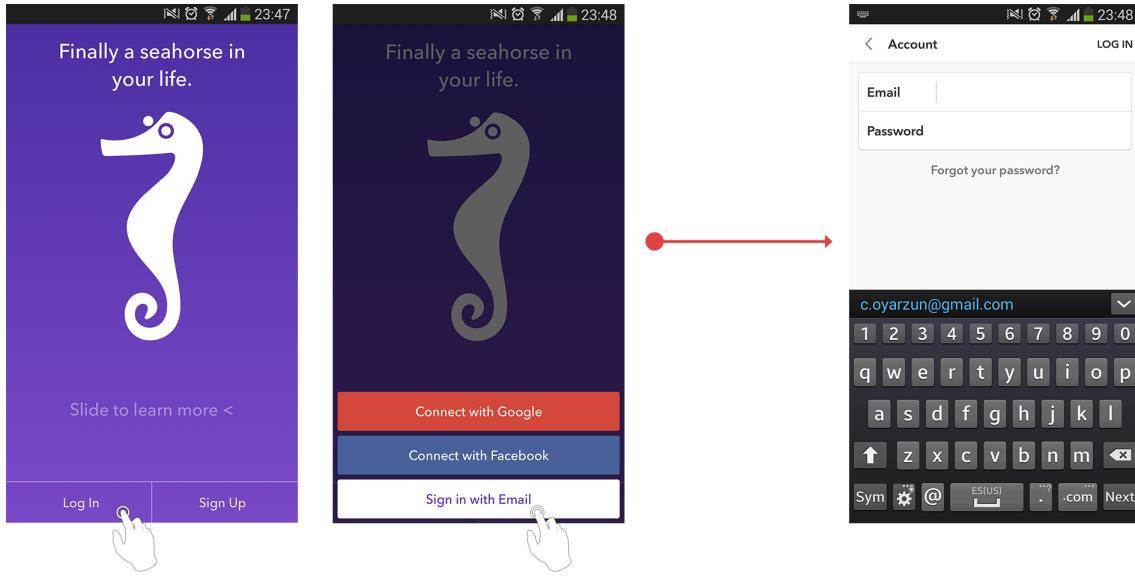


Figura 5.2 – Flujo para acceder a la pantalla de Login en la aplicación Seahorse. En la primera imagen se presiona el botón *Log In*, lo cual abre el menu de la segunda imagen, y se presiona el botón *Sign in with email*, lo que abre la vista de Login de la última imagen. Fuente: Elaboración Propia

Dentro del proyecto de Seahorse, se creó una clase en el paquete *co.seahorse.android.test*.

En este caso la *Activity* testeada es *SeahorseStartActivity* y el nombre de la clase que se encarga de realizar el test es *SeahorseStartActivityTest*.

```

@RunWith(RobolectricGradleTestRunner.class)
public class SeahorseStartActivityTest {
    private SeahorseStartActivity activity;
    private Button buttonLogin;
    private Button buttonRegister;
    private Button buttonSign;
    @Before
    public void setup() throws Exception {
        activity =
            Robolectric.buildActivity(SeahorseStartActivity.class).create().start().get();
        buttonLogin = (Button)
            activity.findViewById(R.id.buttonLogInAccount);
        buttonRegister = (Button)
            activity.findViewById(R.id.buttonRegisterAccount);
        buttonSign = (Button) activity.findViewById(R.id.buttonSignIn);
    }
}

```

El método `setup()` se encarga de crear la referencia a la *Activity* que se desea testear y obtener los botones que también serán testeados.

Es una buena práctica verificar que ninguno de los elementos obtenidos en el método anterior estén vacíos, por lo que el primer test se encargará de eso:

```
@Test
public void shouldNotBeNull() {
    assertNotNull(activity);
    assertNotNull(buttonLogin);
    assertNotNull(buttonRegister);
    assertNotNull(buttonSign);
}
```

Por último se realiza el test del flujo que lleva a la pantalla de Login:

```
@Test
public void buttonClickShouldStartLoginActivity() throws Exception {
    buttonLogin.performClick();
    buttonSign.performClick();
    Intent intent =
        Robolectric.shadowOf(activity).peekNextStartedActivity();
    assertEquals(LoginActivity.class.getCanonicalName(),
        intent.getComponent().getClassName());
}
```

A través del método `performClick()` es posible simular la lógica del botón, tal como si un usuario real lo hubiese presionado. Esto se realiza dos veces, ya que se tienen que simular dos clicks a dos botones distintos. El último de los click es el que abre la pantalla de Login por lo que se verifica que la *Activity* que se inició corresponda a *LoginActivity*.

Al ejecutar los tests, se genera un reporte en HTML, que se ubica en la raíz del proyecto, en `./build/test-report/index.html`. En este reporte se detalla la cantidad total de test ejecutados y el tiempo que se demoraron, como también los test que fallaron y la razón.

5.1.2. Robotium

Se implementó Robotium, en vez de Espresso, principalmente porque existe mayor documentación y porque cuenta con soporte para gradle de forma oficial, por lo que la integración con Android Studio es menos compleja. Además lleva más tiempo de desarrollo y es una herramienta bastante madura.

Para agregar Robotium a un proyecto configurado con Gradle es necesario agregar esta línea al archivo *build.gradle* del proyecto:

```
dependencies {
    ...
    androidTestCompile 'com.jayway.android.robotium:robotium-solo:4.3'
}
```

El test que se realizó es similar al realizado a través de Robolectric, aunque ahora de forma instrumental, es decir ejecutándolo directamente en el dispositivo. Para ello se ha creado la clase RobotiumTest, la cual consiste en lo siguiente:

```
public class RobotiumTest extends ActivityInstrumentationTestCase2 {
    private Solo solo;

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

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

    public void testRobotium() throws Exception {

        solo.assertCurrentActivity("SeahorseStartActivity Never Loaded",
            SeahorseStartActivity.class);
        solo.clickOnText(getActivity().getString(R.string.LOGIN));
        solo.waitForText(getActivity().getString(R.string.SIGN_IN_WITH_EMAIL));
        solo.clickOnText(getActivity().getString(R.string.SIGN_IN_WITH_EMAIL));
        solo.assertCurrentActivity("LoginActivity Never Loaded",
            LoginActivity.class);
        solo.takeScreenshot();
    }
}
```

}

La clase *Solo* es la principal para desarrollar tests con Robotium. En el método *setup()* se crea una referencia de *Solo* a partir de la *Activity* actual.

En el método *testRobotium()* es donde se lleva a cabo el test. Como se puede ver, todo se realiza a través de la clase *Solo*. Con ella se corrobora en un principio si la *Activity* actual corresponde a *SeahorseStartActivity*, luego se hace un click en el texto que dice “Log In”. Debido a que existe una animación para mostrar el menú, se recurre al método *waitForText* para esperar que la animación termine y luego presionar este botón. Finalmente se presiona el texto “Sign in with email” y se comprueba que la nueva *Activity* corresponda a *LoginActivity*. Todo este proceso, al ser ejecutado en un dispositivo real, demora 5.69 segundos, lo cual no es mucho, pero si se toma en cuenta que dentro de un proyecto se puede tener una gran cantidad de tests, entonces sí puede decirse que es un tiempo considerable. Debido a esto, es importante decidir de forma sabia, qué cosas son realmente necesarias testear de manera instrumental.

5.1.3. Spoon

Spoon es una herramienta que intenta simplificar la tarea de realizar tests de forma distribuida. Es posible ejecutar los mismos tests escritos con Robotium en múltiples dispositivos simultáneamente. Una vez que todos los tests se han completado, un resumen es presentado con información detallada sobre cada dispositivo y cada test en un archivo HTML.

Es necesario agregar dos archivos al proyecto, uno es *spoon-runner* que permite la ejecución simultánea, y el otro es *spoon-client* que se encarga de obtener screenshots de cada uno de los dispositivos, para luego presentarlos en el resumen general.

Para realizar el testing de forma distribuida fue usado un dispositivo Samsung Galaxy S4 y dos emuladores. Debido a que los emuladores provistos por Android son demasiados lentos se usaron los de Genymotion. Ellos ofrecen cerca de 20 emuladores

con las simulaciones de distintos dispositivos de forma gratuita. En este caso se usaron los emuladores de un Nexus 4 con Android 4.2.2 y un HTC One con 4.2.2.

El test que se realizó consistía en corroborar que en la pantalla de registro, al completar los datos y dejar uno en blanco, si se presiona el botón “SIGN UP”, debe aparecer un ícono en el campo vacío, indicando que se necesita completar toda la información. Para ello se utilizó Robotium junto con Spoon, el primero para poder interactuar con la UI, mientras que Spoon se utilizó para obtener los screenshots de los tres dispositivos durante todo el proceso. El código a continuación muestra el uso de Spoon, a través del método *Spoon.screenshot*:

```
Spoon.screenshot(getActivity(), "RegisterActivity");
solo.assertCurrentActivity("SeahorseStartActivity Never Loaded",
    RegisterActivity.class);
solo.enterText((EditText) solo.getView(R.id.editLastNameRegister),
    "Oyarzun");
solo.enterText((EditText) solo.getView(R.id.editEmailRegister),
    "c.oyerzun@gmail.com");
solo.enterText((EditText) solo.getView(R.id.editPasswordRegister),
    "123456");
Spoon.screenshot(getActivity(), "RegisterActivityWithInfo");
solo.clickOnText(getActivity().getString(R.string.SIGN_UP).toUpperCase());
Spoon.screenshot(getActivity(), "ShowMissingField");
```

Una vez que el test ya está listo, es necesario ejecutar por línea de comandos lo siguiente:

```
cristopher$ java -jar spoon-runner-1.1.1-jar-with-dependencies.jar --apk
    ../PhotoApp/build/outputs/apk/PhotoApp-debug.apk --test-apk
    ../PhotoApp/build/outputs/apk/PhotoApp-debug-test-unaligned.apk --sdk
    ../../Downloads/adt-bundle-mac-x86_64/sdk/
```

Los parámetros que se agregan son: el APK, el APK del test y la ruta del SDK. Después de unos segundos se genera una carpeta con el nombre *spoon-out* que contiene los resultados de los tests en cada uno de los dispositivos.

En la figura 5.3 se pueden ver los resultados obtenidos en el emulador de Genymotion Nexus 4, en el dispositivo Samsung S4 y en el emulador Genymotion HTC One. Todos los test funcionaron de forma correcta, aunque en la última pantalla, si bien se muestra

el ícono indicando que el campo está vacío, éste debería estar acompañado de un diálogo con un texto con el mensaje “This field is required”, por lo que se puede ver que Spoon no es capaz de capturar los diálogos que se muestran sobre la *Activity* testeada.

Genymotion Nexus 4 - 4.2.2 - API 17 - 768x1280

The screenshots show a 'Create Account' form with fields for First Name, Last Name, Email, and Password. The first screenshot shows empty fields. The second screenshot shows all fields filled with the provided values. The third screenshot shows the same data with a red error icon next to the First Name field, indicating a validation error.

samsung GT-I9505

The screenshots show a 'Create Account' form with fields for First Name, Last Name, Email, and Password. The first screenshot shows empty fields. The second screenshot shows all fields filled with the provided values. The third screenshot shows the same data with a red error icon next to the First Name field, indicating a validation error.

Genymotion HTC One - 4.2.2 - API 17 - 1080x1920

The screenshots show a 'Create Account' form with fields for First Name, Last Name, Email, and Password. The first screenshot shows empty fields. The second screenshot shows all fields filled with the provided values. The third screenshot shows the same data with a red error icon next to the First Name field, indicating a validation error.

Figura 5.3 – Resultados del test ejecutado a través de Spoon. Fuente: Elaboración Propia

5.1.4. uiautomatorviewer

Esta es una herramienta de apoyo, que puede servir para personas que están involucradas en el desarrollo de un proyecto móvil, pero que no son desarrolladores. Se puede obtener información muy detallada sobre el árbol jerárquico de vistas y de cada uno de sus elementos. A través de esta información es posible planear diferentes tipos de tests.

En la figura 5.4 se puede apreciar que a la izquierda se tiene una screenshot de la aplicación, y al lado derecho se muestra el árbol jerárquico de vistas de la aplicación. Además el desarrollador puede presionar sobre los elementos del screenshot y obtener información sobre esa vista, tales como la clase, si la vista tiene habilitado el scroll, si la vista se le puede hacer click, entre otras cosas.

5.1.5. Monkey

Esta herramienta es principalmente usada para realizar casos de testing de estrés. Esto quiere decir que la aplicación es deliberadamente sometida a una gran cantidad de eventos para determinar la estabilidad y el manejo de errores. Los eventos generados consistirán en acciones que podría realizar un usuario normal, como clicks, toques, o gestos, pero en una cantidad muy elevada.

Por ejemplo, el siguiente comando enviará 2000 eventos aleatorios a la aplicación con el nombre de paquete *co.seahorse.android*:

```
adb shell monkey -p co.seahorse.android -v 2000
```

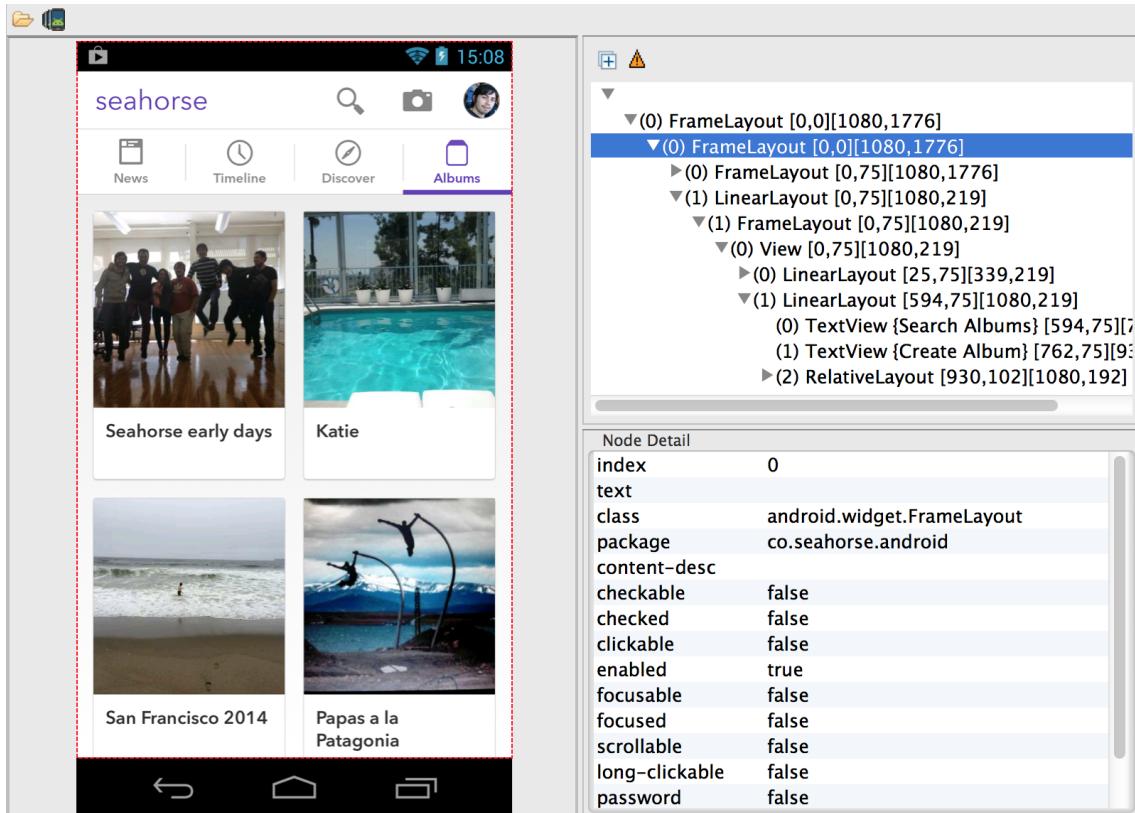


Figura 5.4 – Vista de la herramienta uiautomatorviewer en una aplicación. Fuente: Elaboración Propria

5.2. Implementación de Herramientas de Distribución

5.2.1. HockeyApp

Antes de HockeyApp, en Seahorse se usaba TestFlight [52], pero ellos dejaron de dar soporte a Android en Marzo de este año [51] ya que fueron comprados por Apple, por lo que fue necesario migrar a otro servicio. HockeyApp cumplía con prácticamente todas las necesidades que entregaba TestFlight, siendo una de ellas el soporte para aplicaciones tanto Android como iOS.

Actualmente se pagan 10 dólares por el servicio, de forma mensual, lo que corresponde al plan más económico ofrecido por HockeyApp, que permite tener hasta 5 aplicaciones distintas en la plataforma con una cantidad ilimitada de usuarios.

Para distribuir las versiones, la primera vez es necesario agregar información básica como el nombre de la aplicación y el nombre del paquete. Luego se sube el APK de la aplicación y se invita a las personas que se deseen a esta versión. Cada vez que se sube una versión nueva, es posible notificar a los usuarios que ya cuentan con una versión anterior para que la actualicen. Además, es posible invitar a nuevas personas en cualquier momento.

El desarrollador cuenta con información básica de los testers, como el modelo y dispositivo con el que cuenta, y también es posible saber si es que ya tienen la última versión.

En la figura 5.5 se puede ver la vista que tiene el desarrollador sobre una versión beta de Seahorse. La acciones más frecuente están en la parte superior, teniendo opciones para subir una nueva versión de la misma aplicación, invitar más usuarios o cambiar información básica de la aplicación. En la parte inferior es posible ver la lista de todas las últimas versiones que se han enviado de esa aplicación, con información como el peso de la aplicación, cuándo fue subida esa actualización y la cantidad de descargas que se han tenido. En Seahorse no está implementado el SDK que entrega reporte de crashes, ya que se usa otra herramienta más avanzada para ello. Por último, sobre las acciones más frecuente hay un menú que ofrece la opción de ver todas las versiones que ha tenido la beta, la lista de usuarios que tienen acceso a la beta y el posible feedback que han entregado cada uno de ellos.

5.2.2. UserTesting

UserTesting es la plataforma usada en Seahorse para lanzar versiones beta y recibir feedback de usuarios de prueba a través de videos. Esta herramienta es usada por grandes empresas como Google, Apple, Microsoft, Twitter, Facebook y Yahoo, para la distribución y testing de versiones de prueba de sitios web, aplicaciones de escritorios y aplicaciones móviles. El servicio funciona a través de la compra de créditos, en donde

The screenshot shows the HockeyApp platform interface for the 'Seahorse' application. At the top, there's a navigation bar with tabs for 'Overview', 'Versions 23', 'Crashes 0', 'Feedback 1', 'Users 12', and 'Timeline'. Below this, there are buttons for 'Add Version', 'Invite User', and 'Manage App'. The main content area starts with the app's name 'Seahorse' and its package name 'co.seahorse.android.debug'. It shows the App ID (6caf551684093e76263bef69fcab049f) and a 'Secret' link. A download and feedback section includes a 'Public Page' link. To the right, there's an icon for the latest version (0.9.7-alpha20140619) and a note that it's an 'Android beta' version. Below this is a table titled 'Latest Versions' showing five entries with columns for Name, Code, Devices, Downloads, Crashes, Storage, and Last Updated. The last row is highlighted in blue. At the bottom, there's a 'Statistics' section with tabs for 'Base Metrics' and 'Usage Time'.

Figura 5.5 – Vista de una aplicación beta de Seahorse en HockeyApp. Fuente: Elaboración Propia

por ejemplo, 1000 dólares otorgan al desarrollador 20 créditos. Cada uno de estos créditos sirve para realizar un test con un usuario, aunque también es posible realizar un mismo test a más de un usuario, pero en definitiva, por cada usuario que realice un test, un crédito es consumido. Debido al elevado valor de cada crédito, el desarrollador debe tener claro qué es lo que quiere testear dentro de la aplicación y cuáles son los resultados que espera.

En Seahorse se usa UserTesting al momento de implementar nuevas características en la aplicación, para ver temas de usabilidad y para corroborar que el usuario entiende de forma intuitiva lo implementado. Además sirve para encontrar posibles errores en otros dispositivos, ya que debido a la fragmentación existente en Android es prácticamente imposible saber como una aplicación se ve en cada uno de los dispositivos existentes.

Para la creación de un test los pasos son los siguientes:

- Seleccionar si la aplicación móvil es de Android o iOS

- Agregar instrucciones de cómo instalar la aplicación y una breve descripción para que el usuario tenga una idea de lo que va a testear.
- Agregar una serie de tareas que el usuario debe ir realizando. También se pueden incluir preguntas durante el proceso, para saber si la tarea solicitada fue complicada, o simplemente para saber qué piensa el usuario sobre esa tarea.
- Agregar un cuestionario final con preguntas más generales sobre la aplicación y el proceso completo.
- Por último es posible solicitar algunas características demográficas como el rango de edad, el país, el género, la experiencia con aplicaciones móviles, y otras restricciones como versión de Android mayor a 4.0, o que el usuario cuente con fotos en su galería.

Cuando en Seahorse se agregó el registro de usuarios a través de Facebook y de Google+, se llevó a cabo la distribución y testing a través de esta plataforma. Esto permitió corroborar que esta nueva característica, al igual que otras, funcionaban correctamente. Fue necesario hacer en total unos 8 tests, ya que en cada iteración se usaban 4 créditos, 2 para comprobar que el registro a través de Facebook funcionaba bien, y otros 2 para comprobar lo mismo, pero a través de Google+. Los tests a través de Facebook funcionaron correctamente, pero con Google+ existieron dificultades, ya que uno de los usuarios no pudo realizar el registro. Debido a esto, después de realizar algunas correcciones se llevaron a cabo nuevos tests, con los que se obtuvieron resultados positivos. Tal como se mencionó antes, es necesario aprovechar cada uno de los tests, por lo que además de testear el registro de los usuarios, también se solicitaron otras tareas, como importar albums de otras plataformas, sincronizar la cámara del dispositivo con Seahorse, invitar a otros usuarios a la aplicación, entre otras cosas.

Cada uno de los videos entregó información valiosa sobre la usabilidad dentro de la aplicación, y permitió verificar la UI en 8 dispositivos distintos, algunos con una

pantalla muy pequeña. Los videos duran entre 15 y 20 minutos, y el usuario va relatando todo el proceso, desde la instalación de la aplicación, hasta el cuestionario final.

5.3. Implementación de Herramientas de Reportes de Crashes

5.3.1. Crittercism

Crittercism ha sido la plataforma escogida para la recepción de crashes, principalmente por el nivel de detalle presente en sus reportes. Además el valor que se paga mensualmente es bastante accesible para cualquier empresa del rubro. Actualmente dentro de Seahorse, se pagan 24 dólares por este servicio, lo que da acceso a reportes muy detallados que permiten reparar de forma rápida cada crash.

Crittercism ha permitido disminuir la tasa de crashes y ofrece variadas estadísticas para ver información sobre los dispositivos en los que existen más crashes, el sistema operativo en que ocurren más crashes o la versión en la que ocurrieron más crashes. Después del lanzamiento de cada actualización se reparan una gran cantidad de bugs, pero aparecen nuevos crashes, lo que es inherente al proceso de desarrollo de software, ya que se incluye nuevo código, pero con esta herramienta es posible contar con información muy detallada, facilitando en gran medida la tarea de los desarrolladores.

La instalación es bastante simple [21]. Para agregar Crittercism a un proyecto configurado con Gradle es necesario agregar esta línea al archivo *build.gradle* del proyecto:

```
compile 'com.crittercism:crittercism-android-agent:+'
```

Luego se deben agregar los siguientes permisos en el archivo Android Manifest del proyecto:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.READ_LOGS"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.GET_TASKS"/>
```

El primer permiso es necesario para poder acceder a Internet y poder enviar los reportes. El segundo es necesario para poder obtener la información de los stack trace del usuario, y ahí saber en qué línea de código ha ocurrido el error. El tercero es para

obtener información sobre el estado de la red, por ejemplo, para saber si el usuario está conectado a Wi-Fi o a través de un carrier. El último permiso sirve para acceder a la información de las últimas dos *Activities* ejecutadas, lo que permite saber en qué pantalla ocurrió el crash.

Una vez que los permisos ya están entregados, se debe inicializar Crittercism. Esto se hace únicamente una vez por aplicación, por lo que se debe hacer en la primera *Activity* que se ejecuta dentro de la aplicación. Para iniciar Crittercism se escribe la siguiente línea en el onCreate:

```
Crittercism.initialize(getApplicationContext(), "CRITTERCISM_APP_ID");
```

Ahora ya están implementadas las características básicas, y comenzarán a llegar los reportes al sitio web de Crittercism. También es posible agregar a otros desarrolladores al servicio, para que tengan acceso a la misma información y que todo el equipo reciba los reportes de crashes al correo.

La documentación con la que cuenta es excelente para entender los alcances y todas las cosas que se pueden hacer a través de Crittercism. Por ejemplo, es posible recibir las excepciones que ocurren dentro de la aplicación. Esto se realiza agregando las siguientes líneas en la excepción que se desea recibir:

```
try {
    throw new Exception("Aquí ocurre una excepción");
} catch (Exception exception) {
    //De esta forma se captura y se envía a Crittercism
    Crittercism.logHandledException(exception);
}
```

A través de los gráficos de la figura 5.6 es posible dimensionar la utilidad de contar con una herramienta como Crittercism. El gráfico de la izquierda indica la cantidad de reportes de crashes recibidos por distintas versiones de la aplicación durante el mes de Enero. El gráfico de la derecha indica la cantidad de usuarios que han sido afectados por al menos un crash durante ese mismo período. A través de estos gráficos es posible

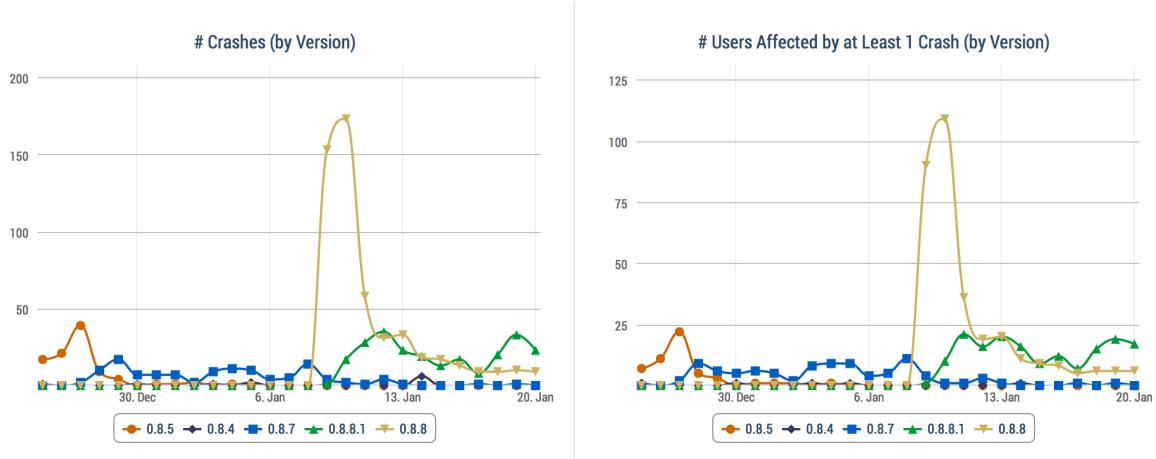


Figura 5.6 – Gráficos que entrega Crittercism sobre cantidad de crashes y usuarios afectados en un periodo de un mes. Fuente: Elaboración Propia

ver que la versión 0.8.8 de color amarillo, fue lanzada el día 9 de Enero y rápidamente comenzaron a llegar una gran cantidad de reportes de crashes, alcanzando la cifra de 173 crashes y afectando a más de 100 usuarios. Debido a ello, al día siguiente se lanzó la versión 0.8.8.1 de color verde, la cual se encargó de arreglar el error introducido en la versión previa y reduciendo este número en gran medida, con sólo 35 crashes y afectando a 20 usuarios. Si no se hubiera contado con una herramienta que entregue los reportes de crashes en tiempo real, probablemente habrían transcurrido más días con el mismo problema, afectando a muchos más usuarios.

Capítulo 6

Conclusiones

6.1. Conclusiones Generales

A través de este trabajo, se han estudiado y comparado diversas herramientas que ayudan a mejorar la calidad de las aplicaciones desarrolladas para Android.

Esto se ha llevado a cabo identificando los problemas existentes durante el desarrollo de aplicaciones móviles, dejando en evidencia que el elemento que más afecta al sistema operativo Android es la fragmentación en dos niveles: software y hardware. De esta última, se desglosan la mayoría de los problemas inherentes a la inmensa cantidad de dispositivos que existen, como por ejemplo los distintos tamaños y resoluciones de pantalla. También existen diferencias de RAM, espacio, núcleos y procesador en cada uno de los dispositivos. Por otro lado, existen problemas que están más relacionados con el proceso de desarrollo de software. Entre éstos se encuentran la distribución de versiones betas que permiten probar con usuarios reales una aplicación antes de que ésta se encuentre disponible para descargar. El último problema se enfrenta cuando la aplicación ya se encuentra en la tienda oficial de Google, y no se sabe realmente si la aplicación está respondiendo como se espera o está presentando errores.

Para atacar cada uno de estos problemas es necesario contar con herramientas que permitan darle un grado de seguridad al desarrollador de que todo va a funcionar de forma correcta, y que en cierto modo, aseguren la calidad de la aplicación desarrollada.

Es por ello que se buscaron herramientas que solucionan o al menos mitigan los problemas presentados anteriormente. La Fragmentación de Android no es un problema que se pueda solucionar con una herramienta, ya que por ejemplo es inviable probar una aplicación en los más de 11.000 dispositivos existentes, y en cada una de las 8 versiones de sistemas operativos vigentes, por lo que la única forma de combatir y mitigar la fragmentación es a través del testing, tanto automatizado como manual. Por otro lado, la distribución de versiones y los crashes son problemas que son solucionables en su totalidad con herramientas que permiten realizar estas tareas. Es por ello que las herramientas fueron clasificadas en tres categorías: Testing, Distribución de Versiones y Reportes de Crashes.

Para cada una de las categorías se realizó un análisis comparativo, a través del cual se definieron las ventajas y desventajas de cada una de las herramientas. En algunos casos fue necesario subdividir las categorías para que las comparaciones arrojen resultados más valiosos. Esto permite que los desarrolladores cuenten con información relevante para tomar mejores decisiones al momento de tener que implementar una de estas herramientas.

En base a las características de cada herramienta, se han implementado algunas de ellas en una aplicación que está disponible en la tienda oficial de Google. Cabe mencionar que las herramientas implementadas fueron las que se ajustaban más a las necesidades de la aplicación Seahorse. Es posible que otros desarrolladores, en base a las ventajas y desventajas presentadas, implementen otro conjunto de herramientas que se ajuste más a su contexto.

Entre las cosas que se evidenciaron durante este trabajo está la gran cantidad de documentación obsoleta encontrada, esto se debe principalmente al rápido desarrollo que ha tenido el sistema operativo Android, por lo que algunas bibliotecas que llevan años de desarrollo simplemente no actualizan su documentación. Otro de los puntos importantes es que hoy en día los desarrolladores están divididos entre el uso de Eclipse

como IDE, y el uso de Android Studio. Poco a poco los desarrolladores deberían ir inclinándose más por esta última, aunque este hecho genera que la documentación se encuentre aún más obsoleta, ya que la mayoría solo cubre la integración con Eclipse.

6.2. Conclusiones Específicas

- La mayoría de la documentación en las herramientas de testing sólo cubren la integración con Eclipse, por lo que si recién se está comenzando a desarrollar un proyecto, y se tiene que elegir entre Eclipse o Android Studio para iniciar el desarrollo, es recomendable usar la primera. Si bien Android Studio se basa en IntelliJ, la versión de Android lleva poco más de un año de desarrollo y aún le falta bastante para acercarse a la robustez y madurez ofrecida por Eclipse.
- Si se desea comenzar a hacer testing unitario dentro de una aplicación Android, se recomienda desarrollar la mayor cantidad de código independiente de la API de Android ya que los test se ejecutarían en la máquina virtual de Java. En los casos en que el código dependa de la API de Android, los tests igualmente pueden ser ejecutados en la máquina virtual de Java gracias a Robolectric, aunque estos tomarán un poco más de tiempo que los tests que tienen sólamente código Java. Si el código que se desea testear cuenta con muchas llamadas a la API de Android, puede que sea necesario usar el Framework de Testing de Android, y estos tests tomarán mucho más tiempo ya que es necesario ejecutarlos en un dispositivo o un emulador con Android.
- Si se desea comenzar a hacer testing de la interfaz gráfica de una aplicación Android, la herramienta a utilizar va a depender principalmente de lo que se quiera testear. Si se quieren testear casos en que una aplicación interactúa con la cámara o con otras aplicaciones, eso es posible con la herramienta nativa, uiautomator. En caso contrario, tanto Robotium como Espresso son muy buenas

herramientas. La gran ventaja de Robotium son los 4 años de desarrollo con los que cuenta, acompañado de excelente documentación. Por otro lado, si bien Espresso es una herramienta nueva y aún no cuenta con una gran comunidad, los tests ejecutados con Espresso son tres veces más rápidos que Robotium ya que están sincronizados con el *thread* de la UI.

- La elección de una de las herramientas de distribución de versiones va a depender del grado de distribución que se está buscando. Si se desea distribuir versiones beta de forma masiva, la opción para ello es Google Play Console. Para grupos pequeños, con los que se busca tener una relación más cercana y recibir feedback constante, están HockeyApp y AppBlade. Por último están UserTesting y The Beta Family, que entregan la opción de distribuir una versión beta y que los usuarios realicen testing manual de la aplicación, a través de una serie de tareas y preguntas especificadas por el desarrollador. Si bien, la mayoría de estas herramientas ofrecen una opción gratuita, para acceder a las características más relevantes es necesario pagar, por lo que la decisión puede depender del presupuesto del desarrollador.
- Para elegir una herramienta que entregue reporte de crashes el desarrollador debe fijarse en el nivel de detalle que quiere recibir. Las herramientas que entregan un mayor información son Crittercism y Bugsense, aunque son de pago. Las opciones gratuitas son Google Analytics, que entrega información básica sobre los crashes y estadísticas generales de la aplicación, aunque los reportes tienen un retraso de un día, y ACRA, que permite especificar qué información se desea recibir en cada reporte, aunque la desventaja está en que el desarrollador debe implementar su propio servicio web para visualizar los reportes.

6.3. Trabajo Futuro

Si bien se tomaron como punto de referencia herramientas que solucionan o mitigan los problemas comunes al desarrollar aplicaciones Android, existen otras herramientas que también podrían mejorar la calidad de las aplicaciones, y que están esperando a ser analizadas y popularizadas por los desarrolladores. Además, las comunidades de algunas herramientas son muy activas y se encargan de mantenerlas actualizadas, mejorándolas de forma periódica.

Dentro del conjunto de herramientas que también podrían a ser analizadas, y que son de gran importancia para asegurar productos de calidad, se encuentran:

- Herramientas de debugging y análisis de performance como Traceview, Systrace, Monitor, DDMS, todas excelentes herramientas ofrecidas por Android de forma nativa.
- Herramientas de análisis de datos y tracking de eventos como Mixpanel, Flurry, Google Analytics, etc.
- Herramientas de apoyo a bases de datos como ORMLite, GreenDAO, etc.
- Bibliotecas útiles para el desarrollo en Android como Picasso, Universal Image Loader, Volley, GSON, Jackson, entre otras.

Bibliografía

- [1] ACRA. Acra - application crash reports for android [en línea]. <<https://github.com/ACRA/acra/>>. [Consulta: 17 de Mayo].
- [2] ACRA. Acra [en línea]. <<http://acra.ch/>>. [Consulta: 17 de Mayo].
- [3] Open Handset Alliance. Industry leaders announce open platform for mobile devices [en línea]. <http://www.openhandsetalliance.com/press_110507.html>. [Consulta: 4 de Mayo].
- [4] Android. Android 2.3 platform and updated sdk tools [en línea]. <<http://android-developers.blogspot.com/2010/12/android-23-platform-and-updated-sdk.html>>. [Consulta: 4 de Mayo].
- [5] Android. Android application error reports [en línea]. <<http://android-developers.blogspot.com/2010/05/google-feedback-for-android.html>>. [Consulta: 4 de Mayo].
- [6] Android. Android dashboards [en línea]. <<http://developer.android.com/about/dashboards>>. [Consulta: 29 de Junio].
- [7] Android. Android developers blog: Android sdk tools, revision 21.[en línea]. <<http://android-developers.blogspot.com.es/2012/11/android-sdk-tools-revision-21.html>>. [Consulta: 23 de Junio].
- [8] Android. Check system version at runtime [en línea]. <<https://developer.android.com/training/basics/supporting-devices/platforms.html>>. [Consulta: 4 de Mayo].
- [9] Android. Great devices with the best of android [en línea]. <<http://www.android.com/phones-and-tablets>>. [Consulta: 20 de Abril].
- [10] Android. monkeyrunner [en línea]. <http://developer.android.com/tools/help/monkeyrunner_concepts.html>. [Consulta: 21 de Junio].
- [11] Android. Supporting multiple screens [en línea]. <https://developer.android.com/guide/practices/screens_support.html>. [Consulta: 4 de Mayo].

- [12] Android. Testing fundamentals [en línea]. <http://developer.android.com/tools/testing/testing_android.html>. [Consulta: 4 de Mayo].
- [13] Android. Ui/application exerciser monkey [en línea]. <<http://developer.android.com/tools/help/monkey.html>>. [Consulta: 4 de Mayo].
- [14] Androideity. Arquitectura de android [en línea]. <<http://androideity.com/2011/07/04/arquitectura-de-android/>>. [Consulta: 4 de Mayo].
- [15] AppBlade. Appblade [en línea]. <<https://appblade.com/>>. [Consulta: 01 de Junio].
- [16] AppBrain. Ratings of apps on google play [en línea]. <<http://www.appbrain.com/stats/android-app-ratings>>. [Consulta: 3 de Mayo].
- [17] The Atlantic. The day google had to start over on android [en línea]. <<http://www.theatlantic.com/technology/archive/2013/12/the-day-google-had-to-start-over-on-android/282479/>>. [Consulta: 4 de Mayo].
- [18] Atlassian. Hipchat [en línea]. <<https://www.hipchat.com/>>. [Consulta: 02 de Junio].
- [19] Bugsense. Bugsense android [en línea]. <<https://www.bugsense.com/docs/android>>. [Consulta: 11 de Mayo].
- [20] Bugsense. Bugsense: New insight boxes & real time goodness [en línea]. <<http://blog.bugsense.com/post/79366063424/new-insight-boxes-real-time-goodness>>. [Consulta: 14 de Junio].
- [21] Crittercism. Crittercism - android sdk documentation [en línea]. <<http://docs.crittercism.com/android/android.html>>. [Consulta: 11 de Mayo].
- [22] Crittercism. Crittercism [en línea]. <<http://www.crittercism.com/>>. [Consulta: 11 de Mayo].
- [23] Ivan Dimoski. Automated android crash reports with acra and cloudant [en línea]. <<http://www.toptal.com/android/automated-android-crash-reports-with-acra-and-cloudant>>. [Consulta: 17 de Mayo].
- [24] Ben Elgin. Google buys android for its mobile arsenal. bloomberg businessweek. bloomberg [en línea]. <<http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal>>. [Consulta: 4 de Mayo].

- [25] Erikrespo. Historical android version distribution according to android market/play store usage. from december 2009 to february 2014. [en línea]. <http://en.wikipedia.org/wiki/File:Android_historical_version_distribution_-_vector.svg>. [Consulta: 4 de Mayo].
- [26] The Beta Family. The beta family [en línea]. <<https://thebetafamily.com/>>. [Consulta: 21 de Junio].
- [27] Steven Mark Ford. Android espresso vs robotium - benchmarks.[en línea]. <<http://www.stevenmarkford.com/android-espresso-vs-robotium-benchmarks/>>. [Consulta: 23 de Junio].
- [28] Peter Friese. Android testing with robolectric [en línea]. <<http://www.peterfriese.de/android-testing-with-robolectric/>>. [Consulta: 27 de Junio].
- [29] GitHub. Github [en línea]. <<https://github.com/>>. [Consulta: 02 de Junio].
- [30] Google. dexmaker: Programmatic code generation for android.[en línea]. <<https://code.google.com/p/dexmaker/>>. [Consulta: 25 de Junio].
- [31] Google. Espressostartguide on android-test-kit [en línea]. <<https://code.google.com/p/android-test-kit/wiki/EspressoStartGuide>>. [Consulta: 4 de Mayo].
- [32] Google. Google analytics android [en línea]. <<https://developers.google.com/analytics/devguides/collection/android/v3/>>. [Consulta: 17 de Mayo].
- [33] Google. Google groups [en línea]. <<https://groups.google.com/>>. [Consulta: 01 de Junio].
- [34] Google. Google play developer console [en línea]. <<https://play.google.com/apps/publish/>>. [Consulta: 11 de Mayo].
- [35] Google. Google plus communities [en línea]. <<https://plus.google.com/communities/>>. [Consulta: 01 de Junio].
- [36] Gradle. Gradle [en línea]. <<http://www.gradle.org/>>. [Consulta: 27 de Junio].
- [37] Emir Hasanbegovic. Android device screen sizes [en línea]. <<http://www.emirweb.com/ScreenDeviceStatistics.php>>. [Consulta: 25 de Abril].
- [38] HockeyApp. Hockeyapp [en línea]. <<http://hockeyapp.net/>>. [Consulta: 01 de Junio].
- [39] Seahorse Inc. Googleplay: Seahorse. [en línea]. <<https://play.google.com/store/apps/details?id=co.seahorse.android&hl=es>>. [Consulta: 25 de Junio].

- [40] JUnit. Junit, a programmeroriented testing framework for java.[en línea]. <<http://junit.org/>>. [Consulta: 21 de Junio].
- [41] Scott Knaster. Google i/o 2013: For the developers [en línea]. <<http://googledevelopers.blogspot.com/2013/05/google-io-2013-for-developers.html>>. [Consulta: 01 de Junio].
- [42] Pivotal Labs. Pivotal tracker [en línea]. <<https://www.pivotaltracker.com/>>. [Consulta: 02 de Junio].
- [43] Joel Murach. *Murach's Android Programming*. Mike Murach & Associates, 2013.
- [44] OpenSignal. Android fragmentation visualized (july 2013) [en línea]. <<http://opensignal.com/reports/fragmentation-2013>>. [Consulta: 25 de Abril].
- [45] OpenSignal. The many faces of a little green robot (august 2012) [en línea]. <<http://opensignal.com/reports/fragmentation.php>>. [Consulta: 25 de Abril].
- [46] Robolectric. Robolectric, getting started [en línea]. <<http://robolectric.org/installation>>. [Consulta: 25 de Junio].
- [47] Robotium. Robotium, the world's leading android test automation framework.[en línea]. <<https://code.google.com/p/robotium/>>. [Consulta: 23 de Junio].
- [48] SDTimes. Robotium tech launches test-case recorder integrated with android framework .[en línea]. <<http://sdt.bz/content/article.aspx?ArticleID=67598&page=1>>. [Consulta: 23 de Junio].
- [49] Square. Spoon - distributing instrumentation tests to all your androids [en línea]. <<http://square.github.io/spoon>>. [Consulta: 4 de Mayo].
- [50] staffcreativa. Los curiosos nombres de las diferentes versiones de android [en línea]. <<http://blog.staffcreativa.pe/android-google>>. [Consulta: 4 de Mayo].
- [51] TestFlight. Android end of life [en línea]. <<http://help.testflightapp.com/customer/portal/articles/1450414>>. [Consulta: 27 de Junio].
- [52] TestFlight. Testflight, beta testing on the fly [en línea]. <<https://www.testflightapp.com/>>. [Consulta: 27 de Junio].
- [53] UserTesting. Usertesting [en línea]. <<https://www.usertesting.com/>>. [Consulta: 21 de Junio].
- [54] Ángel J. Vico. Arquitectura de android [en línea]. <<http://columna80.wordpress.com/2011/02/17/arquitectura-de-android>>. [Consulta: 4 de Mayo].

- [55] Lars Vogel. Android application testing with the android test framework - tutorial [en línea]. <<http://www.vogella.com/tutorials/AndroidTesting/article.html>>. [Consulta: 4 de Mayo].
- [56] Lars Vogel. Android user interface testing with robotium - tutorial [en línea]. <<http://www.vogella.com/tutorials/Robotium/article.html>>. [Consulta: 4 de Mayo].
- [57] Lars Vogel. Testing with easymock - tutorial [en línea]. <<http://www.vogella.com/tutorials/EasyMock/article.html>>. [Consulta: 4 de Mayo].
- [58] Lars Vogel. Unit tests with mockito - tutorial [en línea]. <<http://www.vogella.com/tutorials/Mockito/article.html>>. [Consulta: 4 de Mayo].
- [59] Lars Vogel. Using robolectric for android testing - tutorial [en línea]. <<http://www.vogella.com/tutorials/Robolectric/article.html>>. [Consulta: 4 de Mayo].