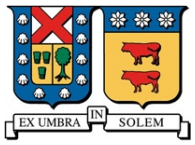
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso – Chile



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

Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso – Chile

TÍTULO DE LA MEMORIA:

ANÁLISIS DE HERRAMIENTAS PARA MEJORAR LA CALIDAD DE APLICACIONES PARA ANDROID

AUTOR:

CRISTOPHER NICOLÁS OYARZÚN ALTAMIRANO

Memoria presentada como requerimiento parcial para optar al título profesional de **Ingeniero Civil en Informática** de la **Universidad Técnica Federico Santa María**.

Profesor Guía: _____
Cecilia Reyes

Profesor Correferente: _____
Chihau Chau

Junio 2014.
Valparaíso, Chile.

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 consectetuer 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 ..."
– Autor

Resumen

El crecimiento que ha tenido el sistema operativo Android es considerable. Existen 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 ha ganado vital importancia. El objetivo de esta memoria es estudiar y comparar herramientas que ayuden a mejorar el desarrollo de aplicaciones Android de tal manera de proveer a los desarrolladores una guía práctica que les permita tomar mejores decisiones en el transcurso de un proyecto.

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 consectetuer signiferumque eu per. In usu latine equidem dolores. Quo no falli viris intellegam, ut fugit veritus placerat per.

Palabras Claves: 7 conceptos máximos separados por coma.

Abstract

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.

Keywords: 7 concepts max.

Índice de Contenidos

Índice de Tablas

Índice de Figuras

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 [24], 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 [13] 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. Normalmente si se desea tener una buena nota por parte del usuario, es necesario que la aplicación sea realmente 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 [13], por lo que es fundamental diferenciarse del resto, entregando un producto de calidad.

1.2. Objetivos

A continuación se presentan la lista de objetivos que se desean abarcar en este trabajo:

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 comúnes, 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 estas, lo que permitirá tener un panorama general de las fortalezas y debilidades que poseen; En el capítulo 4 se realizará un análisis y se compararan algunas herramientas para poder concluir que se debe usar y para qué casos; En el capítulo 5 se realizará un análisis y se compararan algunas herramientas para poder concluir que se debe usar y para qué casos. Finalmente en el capítulo 6 se presentan las conclusiones obtenidas apartir 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. A continuación se darán a conocer algunos detalles sobre sus inicios, arquitectura y evolución con el tiempo.

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, que estuvieran más enfocados en la localización del dueño y en distintas preferencias.

Google compró a Android Inc. el 17 de Agosto del 2005 [17]. Poco se sabía sobre esta compañía para ese entonces ya que estuvo funcionando de forma secreta, sin dar a conocer muchos detalles sobre lo que desarrollaban. Muchos asumían que Google estaba

planeando entrar al mercado de dispositivos móviles. De ahi en adelante los esfuerzos de Google se enfocaron en conversaciones con fabricantes y carriers, con la promesa de proveer de un sistema flexible y actualizable.

Sin embargo, la aparición del iPhone el 9 de Enero del 2007 [14] tuvo un efecto disruptivo en el desarrollo de Android. Hasta el momento se contaba con un prototipo, el cuál 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 operativa 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 [12], 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, el cuál es portable y seguro. Para cada pieza de hardware, como la cámara o el bluetooth, existe un driver dentro del kernel, que permitirá 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 cuáles

están escritas en C o C++, y son compiladas para la arquitectura específica del dispositivo. En la mayoría de los casos el fabricantes es quien se encarga de instalarla en su dispositivo. Las bibliotecas incluídas 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 compontentes que posee son:

  - **Administrado de actividades (Activity Manager):** Gestiona la pila de actividades de la aplicación, como también su ciclo de vida.
  - **Administrador de ventanas (Windows Manager):** Organiza lo que se mostrará en pantalla. Crea las superficies en la pantalla, que posteriormente estarán ocupadas por las actividades.
  - **Proveedor de contenidos (Content Provider):** Encapsula los datos que pueden ser compartidos por las aplicaciones, facilitando la comunicación entre estas.
  - **Vistas (Views):** Son los elementos que nos permitirán construir las interfaces de usuario, como listas, botones, textos, hasta otros elementos más avanzados como visores de mapas.

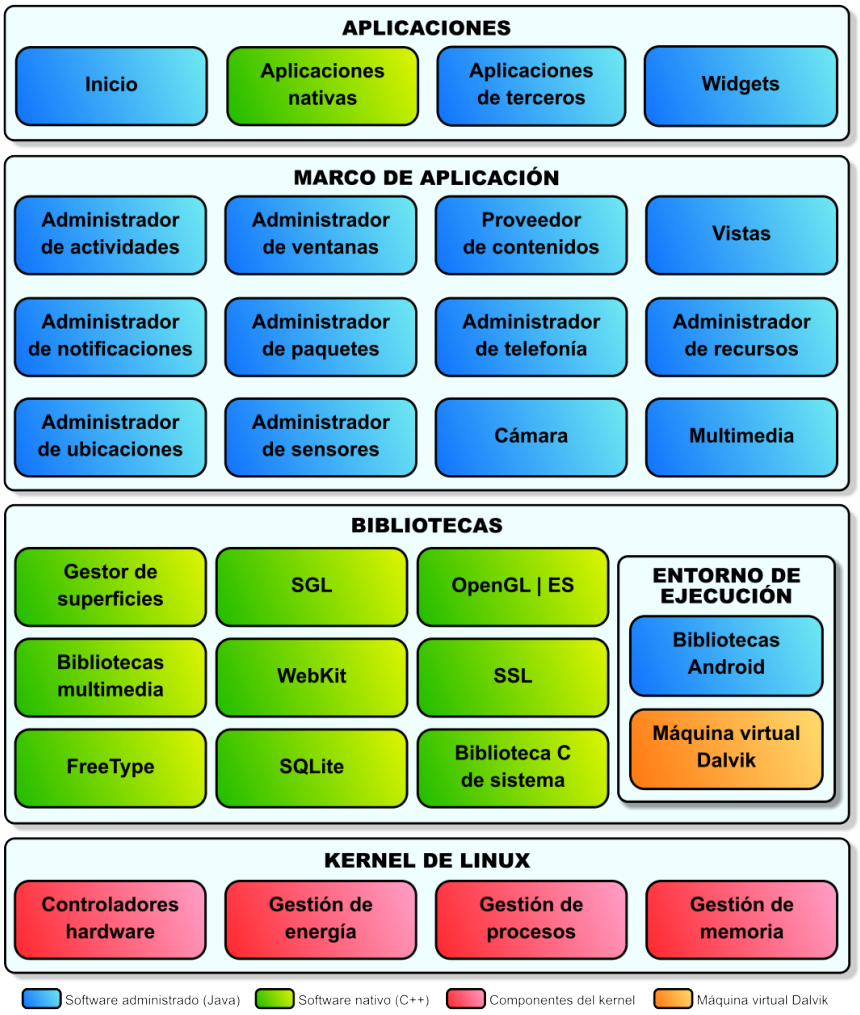


Figura 2.1 – Arquitectura de Android, compuesta por cuatro capas. [29]

- **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 [23, p. 14] lo que ocurre es que se generan los Bytecode Java (archivos .class) en base a nuestro código fuente Java (archivos .java). Luego se compilan estos archivos .class a archivos ejecutables Dalvik (archivos .dex), los cuales pueden ser ejectuados

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 un archivo llamadado APK (Android Package). Este contiene todos los archivos necesarios para ejecutar la aplicación, incluyendo los archivos .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.

2.1.4. Tipos de dispositivos

En el sitio web de Android [8], 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, este puede ser personalizado para que funcione con otros tipos de dispositivos electrónicos.

A continuación se listan los otros dispositivos que cuentan con Android:[23, p. 5]

- Lectores de libros.
- Cámaras.
- Sistemas en vehículos.

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

- Casas inteligentes.
- Consolas de videojuegos.
- Televisores inteligentes.
- Relojes inteligentes.

2.1.5. 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 alfabático. El oreden 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, el cual fue lanzado para dispositivos comerciales el 22 de Noviembre del 2013.

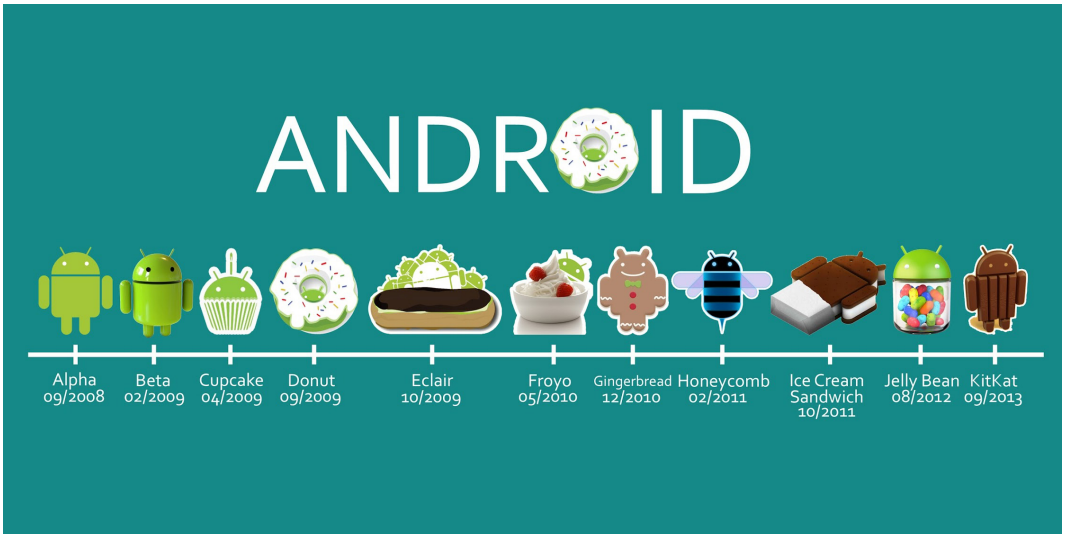


Figura 2.3 – Versiones de Android. [28]

Al comenzar el desarrollo de una aplicación Android, se debe decidir cual va a ser la API mínima a la cuál 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 ningúna 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 tienen 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. Esta información es recolectada durante los últimos 7 días de cada mes, terminando el 1 de Abril del 2014. 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%

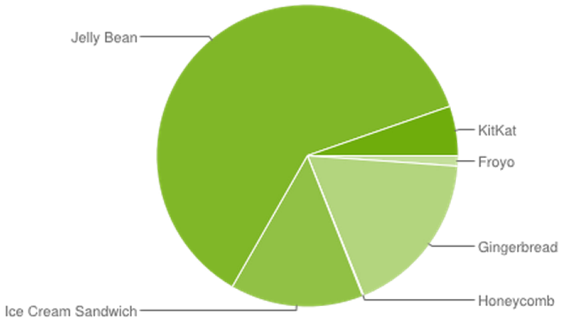


Figura 2.4 – Estadísticas relativas al número de dispositivos que tiene cada versión de Android en Abril del 2014.

2.1.6. 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 [24]. A continuación se daran a conocer los tipos de testing más conocidos.

Tipos de test

Testing unitario

Un test unitario (unit test) es una pieza de código escrito por un desarrollador que ejecuta una funcionalidad específica en el codigo 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, estos 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 UI

Además de testear los componentes individuales que permiten el funcionamiento de la aplicación, como actividades 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, 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 algun 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 unitatios. Además tienden a ser más complejos y lentos que los test unitarios.

Testing funcional o de aceptación

Normalmente estos tipos de test son creados por gente 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 requierimientos o características que debería tener la aplicación. Los testers y desarrolladores también pueden colaborar en la creación de estos.

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 clases de test como:

- Smoke test: Es un testing rápido que se lleva a cabo sobre toda la aplicación. Su objetivo no consiste en encontrar bugs, sino que es 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.

¿Qué testear?

Tan importante como saber hacer testing, es saber que cosas son necesarias de testear. A continuación se listan algunas situaciones comúnes relacionadas con Android que se deberian 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, dejandolo en *landscape* (posición horizontal) en vez de portrait (posición vertical).

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

- ¿Se dibuja de forma correcta la pantalla?
- ¿La aplicación mantiene el estado? La Actividad no debería perder nada que el usuaurio 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 también desencadenan el comportamiento por defecto de destruir y recomenzar la Actividad.

Dependencias de fuentes externas

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

Cambios en segundo plano

Si la aplicación esta inactiva y ya ha pasado a segundo plano, lo más probable es que el sistema destruya la o las actividades de la aplicación para dar memoria a otras

aplicaciones que esten corriendo actualmente. Es por ello que testear el ciclo de vida de la actividad es necesario para corroborar si se destruye y recomienza de forma exitosa, sin pérdida del estado actual.

¿Qué tests requieren un sistema con Android para ejecutarse?

Testing de clases en Java

Si las clases que se tienen no hacen llamadas a la API de Android, se puede usar el framework de JUnit sin ningúna restricción [30].

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

Si se quieren hacer tests que usen la API de Android, estos 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.[30]

2.2. Problemas al desarrollar en Android

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

2.2.1. Fragmentación

La Fragmentación es el elemento que más afecta a Android. Debido a los más de 11.000 diferentes tipos de dispositivos [24], como también a las ocho versiones vigentes del sistema operativo [6], es mucho más dificil 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 categorias, 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 muchos sistemas operativos de Android vigentes hoy en día. Esto priva al desarrollador de muchos funciones 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 [7] 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 que muchas veces el desarrollador deba programar una funcionalidad más de una vez. Actualmente muchos 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 rápidez 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 puede ver 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.

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,

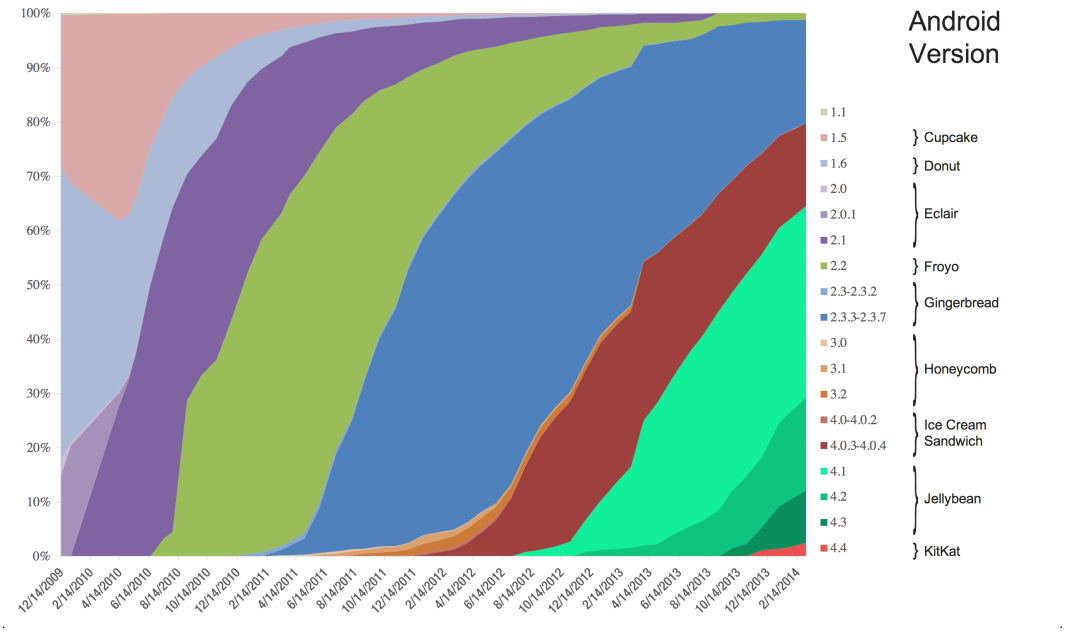


Figura 2.5 – Distribución historica de versiones de Android.[18]

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 ya que cada dispositivo con Android cuenta con diferentes características.

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 definirá 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.

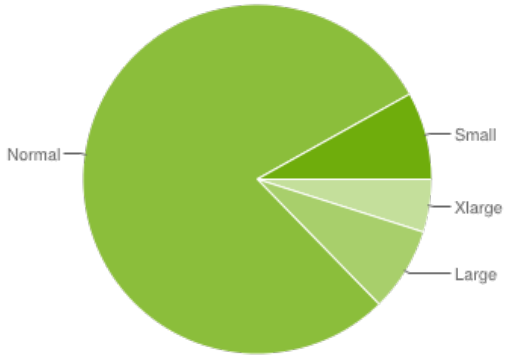


Figura 2.6 – Gráfico con la distribución de tamaños de pantalla.

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 [8], 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 Abril del 2014.

Fragmentación en las resoluciones de pantalla

Android categoriza las resoluciones de cada pantalla en base a la densidad de pixeles 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 imagén 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 pixeles, ya que si el ancho y alto se asignan en pixeles, va a ocurrir lo que se muestra en la figura 2.7. Por ello, es muy importante asignar valores que esten en la escala de densidad de pixeles.

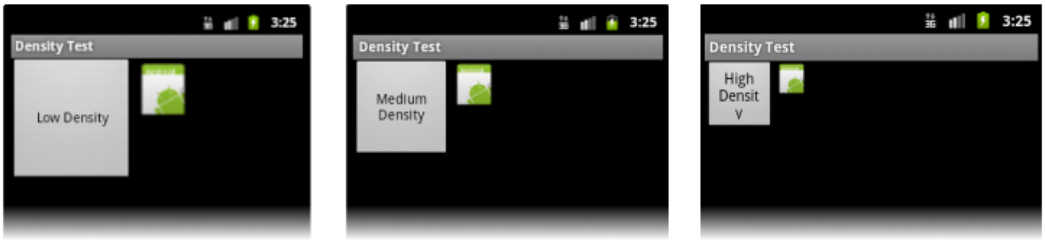


Figura 2.7 – Consecuencias de asignar alto y ancho en pixeles en vez de en densidad de pixeles.[9]

Aún con estas precauciones, es muy dificil saber cuales 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), ya que 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 [22], se puede saber cúal es la densidad de pixeles de prácticamente todos los dispositivos más populares de Android.

En el gráfico de la figura 2.8 se puede apreciar la información que entrega Google sobre la distribución de densidad existente durante el mes de Abril del 2014 [8].

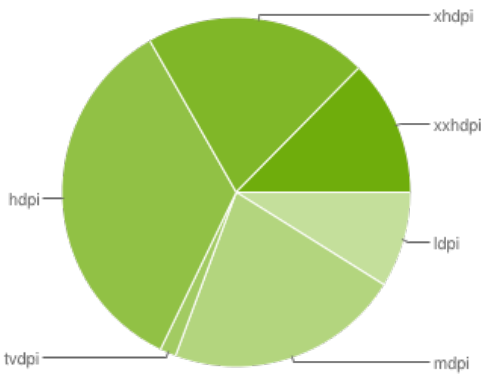


Figura 2.8 – Gráfico con la distribución de densidades de pantalla.[18]

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 cuál 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 [25] y 2013 [24]. 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 2.9 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 [24] 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!"*

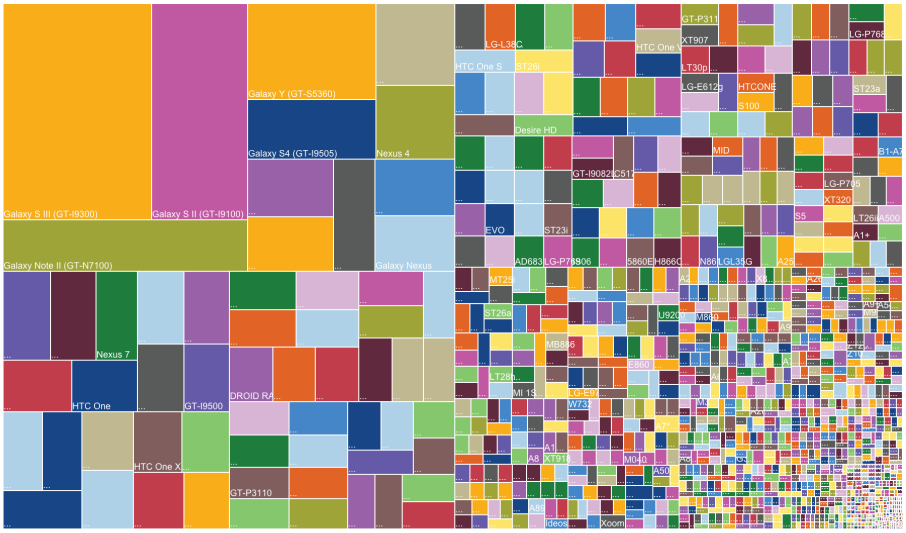


Figura 2.9 – Fragmentación de dispositivos entregado por OpenSignal el 2013.

Similar a las estadísticas de dispositivos, la gráfica de la figura 2.10 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 encuentra en esta gráfica, pero que tienen porcentajes muy bajos son Motorola, HTC, Huawei, LG y Google.

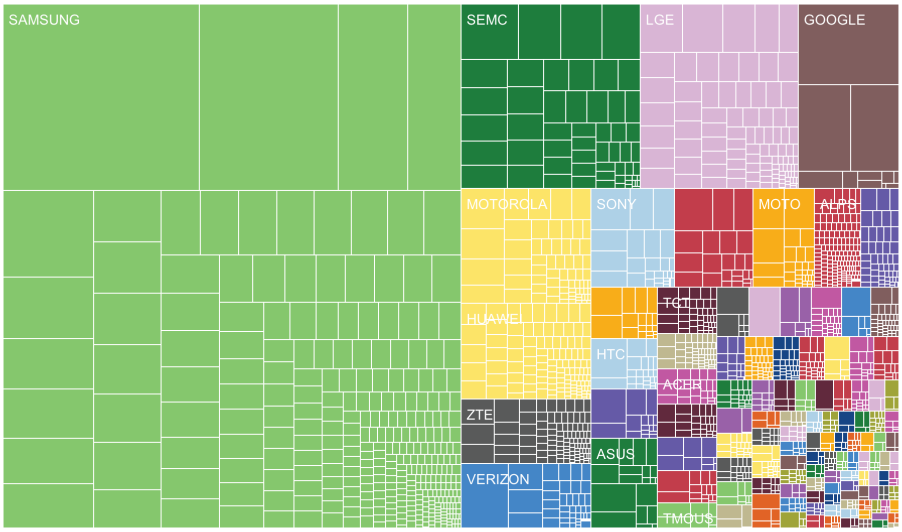


Figura 2.10 – Fragmentación a nivel de fabricantes de dispositivos Android.

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 desafios 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 desafio tiene relación con los tamaños de pantalla, ningún otro sistema operativo móvil tiene tanta diversidad. En las figuras (2.11 y 2.12) 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.

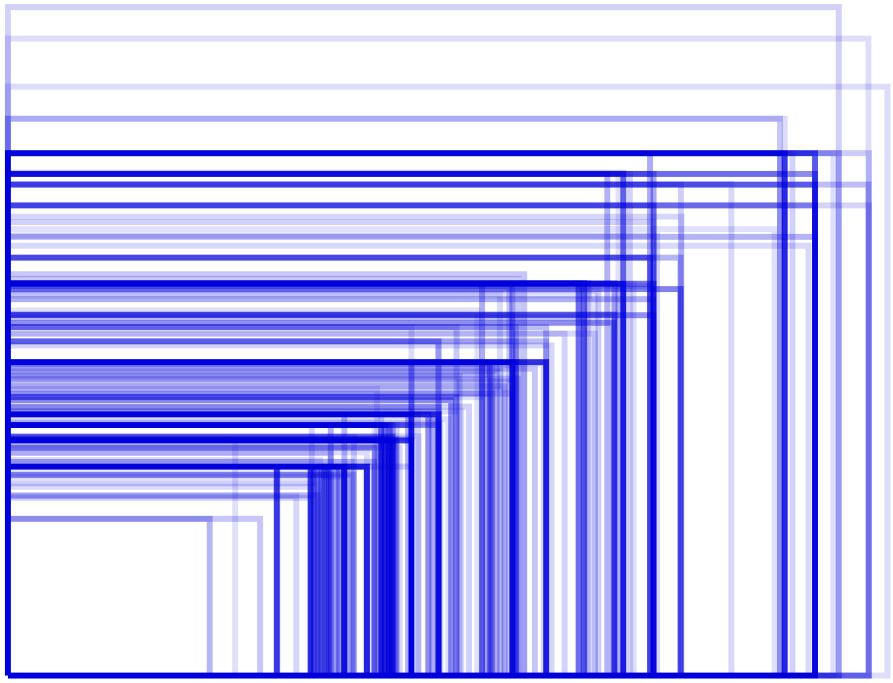


Figura 2.11 – Fragmentación de pantallas en dispositivos Android entregado por OpenSignal en Julio del 2013.

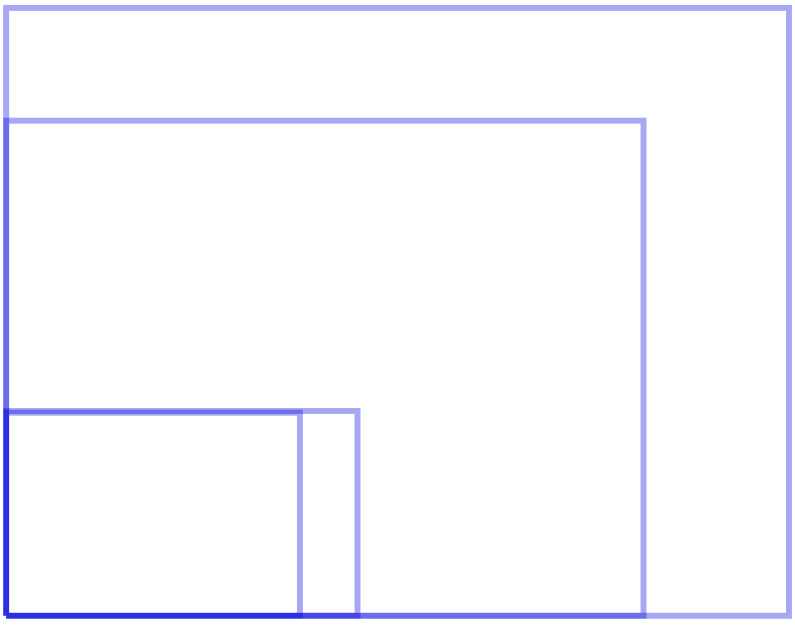


Figura 2.12 – Fragmentación de pantallas en dispositivos iOS entregado por OpenSignal en Julio del 2013.

2.2.2. 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 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, para corroborar que todas las características de la aplicación están funcionando correctamente antes de subir una actualización. [FALTA COMPLEMENTAR MAS]

2.2.3. 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 version de Android 2.2 (Froyo). Si bien esto es bastante útil, la mayoría de las veces no es suficiente ya que los reportes que se reciban van a depender de si el usuario afectado desea enviar un reporte al momento que la aplicación dejó de responder correctamente.

Este proceso es fundamental para el desarrollo y mantenimiento de una aplicación ya que gracias a los reportes de crashes es posible saber en que 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 2.13, la idea básica es recibir feedback del usuario para reparar la aplicación y que este tenga una mejor experiencia la próxima vez.

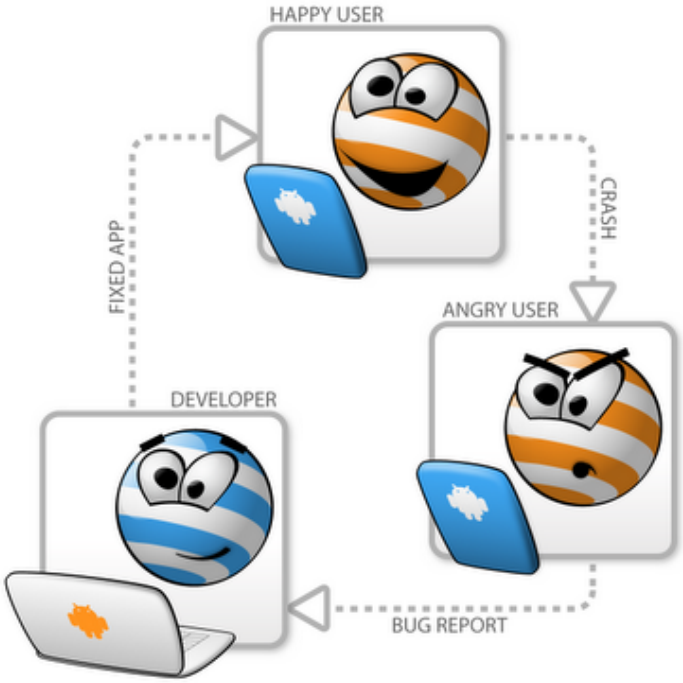


Figura 2.13 – Concepto de Google sobre como el proceso de reportar crashes hacen al usuario feliz.

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

Ahora que ya se conoce lo básico sobre testing y se han visto las herramientas que entrega el framework de testing de Android, se comenzaran a ver el resto de herramientas que se encuentran disponibles.

3.1.1. Herramientas de testing provistas por Android

Android provee un buen framework de testing [10] 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

El testing de 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.

Instrumentation

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 Instrumentation, que permite a los tests controlar el ciclo de vida 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 activado 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 Actividad llama al método *finish()*, entonces el método *onDestroy()* también es llamado. La API de Android no provee un 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 cuál comienza una actividad y retorna la actividad que está siendo testeada. Después se puede llamar al método *finish()*, seguido por un método *getActivity()* nuevamente, y así poder 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 aplciación esta ejecutándose.

Simulando objetos (Mock objects)

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 compontentes sin afectar la base de datos del sistema de archivos de un dispositivo Android.

uiautomator

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 [30].

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 apra encontrar un elemento.

uiautomatorviewer

Android también provee la herramienta *uiautomatorviewer*, que permite analizar la interfaz gráfica de una aplicación. Esta herramienta puede ser usada apra 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*. [AGREGAR SCREENSHOT]

Monkey

Monkey [11] es un herramienta de línea de comando que envia eventos aleatorios a un dispositivo. Se puede restringir a *Monkey* para que se ejecute únicamente en ciertos paquetes, por lo que se le pueden dar instrucciones para testear únicamente una aplicación.

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
```

monkeyrunner

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

A través de *monkeyrunner* se puede hacer un script para realizar un test. Este se ejecuta através 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.

3.1.2. Herramientas de testing de terceros

EasyMock

EasyMock es un framework para mocking [32], 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

Mockito [33] 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 similado con una llamada al método

MockitoAnnotations.initMocks(this).

Robolectric

Robolectric [34] es un framework que simula parte del framework de Android contenido en el archivo *android.jar*. Esta 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, este dirige estas 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, este 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.

Robotium

Robotium [31] 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 actividades de Android.

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

Espresso

Google lanzó el framework Espresso [19] 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 estan 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 este 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

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 [27].

La aplicación se ejecuta en base a los test que hayamos 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");
```

Remote Test Lab

Remote Test Lab [26] es un servicio que Samsung ofrece a los desarrolladores para que puedan probar sus aplicaciones en un dispositivo real, pero en remoto. Gracias a él podrán acceder, vía web, a diferentes smartphones y tablets con varias versiones del sistema operativo. En estos dispositivos los desarrolladores podrán instalar y testear sus apps...

3.2. 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:

3.2.1. Herramienta de reporte de crashes provista por Android

Cuando la aplicación ya esta publicada, es posible acceder a una sección dentro de la *Google Play Developer Console* [21], titulada *CRASHES & ANRS*. En este sitio es posible ver los reportes de los últimos 6 meses. Tal como se muestra en la figura 3.1, 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.

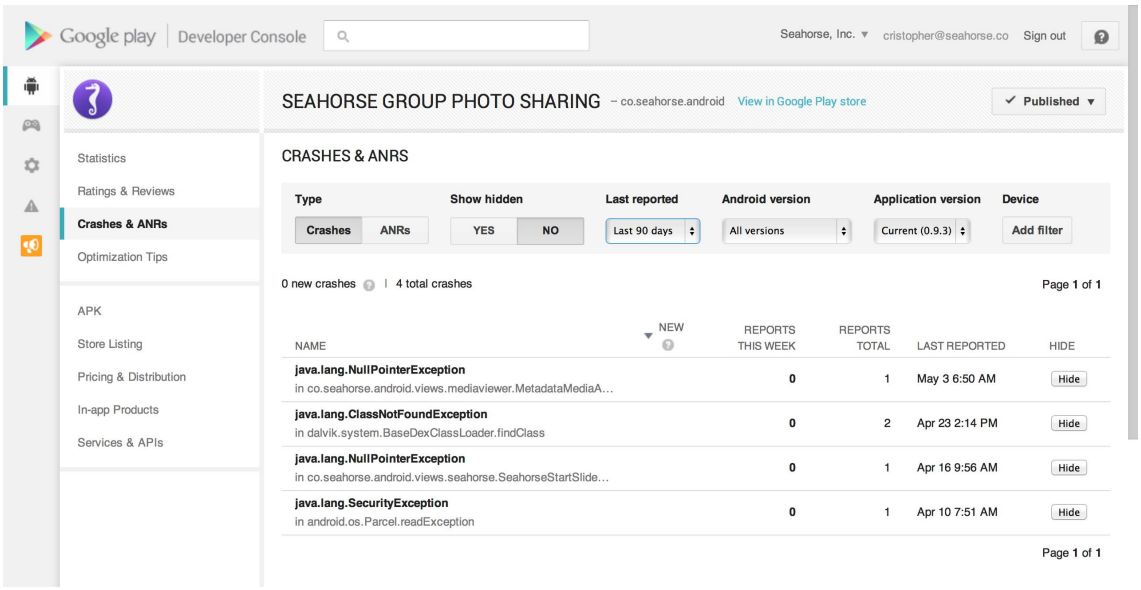


Figura 3.1 – Vista del sistema de reporte de crashes provisto por Android.

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

3.2.2. 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 estas 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 envíados, 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:

3.2.3. Crittercism

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.

La instalación es bastante simple [16]. Se debe descargar el SDK de Crittercism para Android e incluirlo al proyecto. 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 ahi saber en que 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 actividades ejecutadas, lo que permite saber en que actividad 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 actividad 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 de Crittercism. También es posible recibir cada reporte de crash al correo, configurando esto desde el sitio web.

Como se puede ver en la figura 3.2, se tienen opciones parecidas al reporte de crashes de Google. La gran diferencia está en el detalle de la información, ya que al revisar los detalles de un crash, se puede ver información muy específica como: nivel de bateria, espacio en el disco, espacio en la tarjeta SD, uso de RAM, estabilidad de la red, orientación del dispositivo, idioma del dispositivo, actividades que estaban ejecutandose, entre muchos otros puntos.

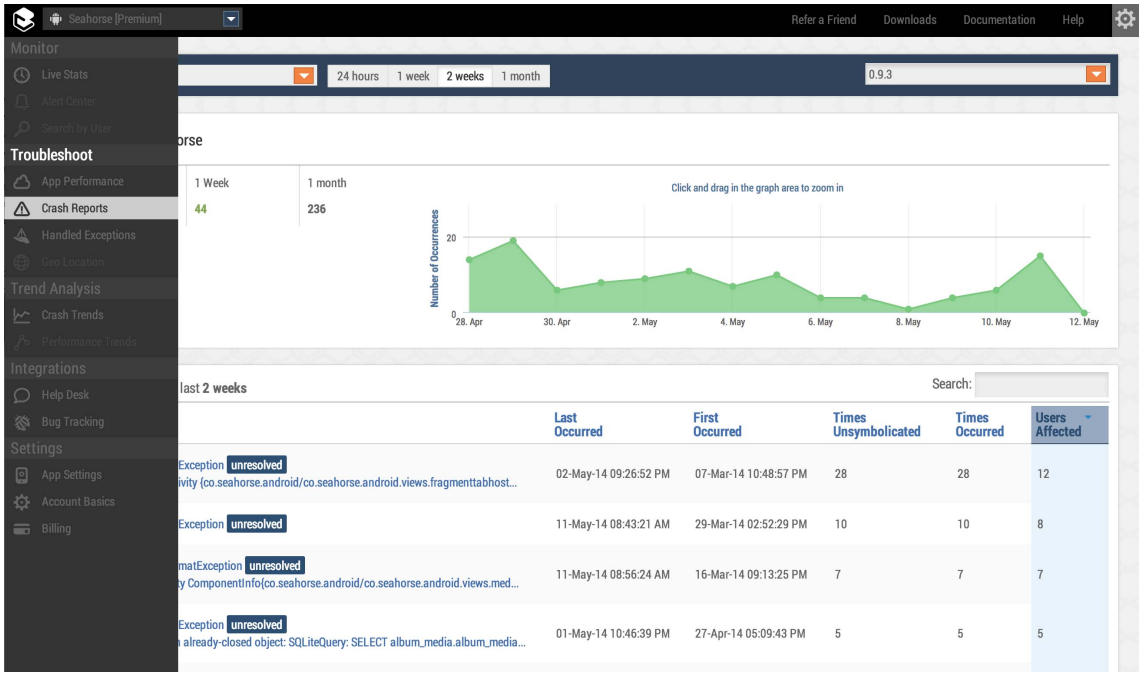


Figura 3.2 – Vista del sistema de reporte de crashes de Crittercism

3.2.4. Bugsense

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.

Para la instalación se descarga la biblioteca desde el sitio de Bugsense [15]. Después de incluirla en el proyecto, se deben pedir los permisos correspondientes en el Android Manifest. Para iniciar Bugsense se escribe la siguiente línea:

```
BugSenseHandler.initAndStartSession(Context, APIKEY);
```

Con esto, Bugsense ya se encuentra implementado. Para implementar las otras características sólo hay que seguir el tutorial que se encuentra en el sitio[15].

En la figura 3.3 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.

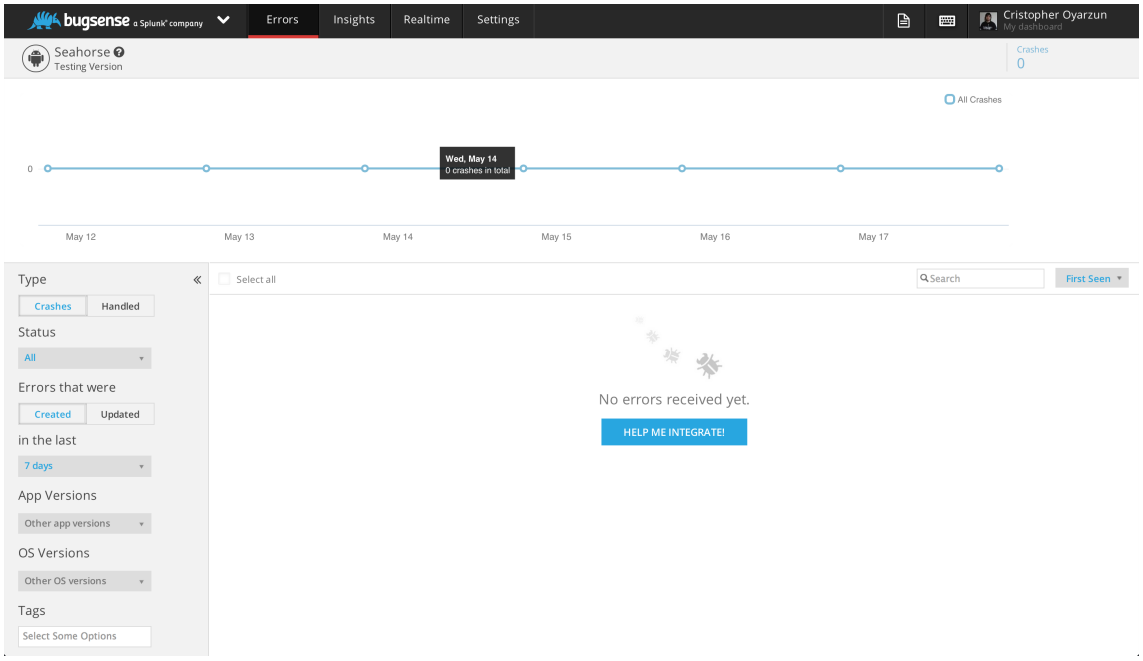


Figura 3.3 – Vista del sistema de reporte de crashes de Bugsense

3.2.5. Google Analytics

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.

Para su implementación es necesario descarga desde el sitio de Google Analytics [20] la versión 3 de su SDK. Una vez descargado el SDK, es necesario incluirlo al proyecto y dar los siguientes permisos en el archivo Android Manifest:

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

Para implementarlo a través de código es necesario agregar estas líneas en cada una de las Actividades de las cuáles se desee obtener información:

```
@Override
  public void onStart() {
    super.onStart();
    ... // El resto del código de onStart()
    EasyTracker.getInstance(this).activityStart(this); // Agregar este método
  }

  @Override
  public void onStop() {
    super.onStop();
    ... // El resto del código de onStop()
    EasyTracker.getInstance(this).activityStop(this); // Agregar este método
  }
```

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

3.2.6. ACRA

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.

Para su implementación es necesario descarga la biblioteca desde el sitio web de ACRA [2]. Una vez que ya se ha añadido al proyecto, es necesario pedir el siguiente permiso en el archivo de Android Manifest:

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

En el código del proyecto la implementación es de la siguiente forma:

```
import org.acra.*;
import org.acra.annotation.*;

@ReportsCrashes(formKey = "", formUri =
    "http://www.aqui_va_la_plataforma_web.com/reportpath")
public class MyApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();
    ACRA.init(this); // Esta es la línea que inicializa ACRA
  }
}
```

Capítulo 4

Análisis Comparativo

Para este análisis comparativo hemos filtrado las aplicaciones que pueden ser más útiles y que pueden ser comparadas, ya que muchas de las herramientas estudiadas son muy distintas una de otra, por lo que es muy complicado realizar una comparación.

Sed iusto nihil populo an, ex pro novum homero cotidieque. Te utamur civibus eleifend qui, nam ei brute doming concludaturque, modo aliquam facilisi nec no. Vidisse maiestatis constituam eu his, esse pertinacia intellegam ius cu. Eos ei odio veniam, eu sumo altera adipisci eam, mea audiam prodesset persequeris ea. Ad vitae dictas vituperata sed, eum posse labore postulant id. Te eligendi principes dignissim sit, te vel dicant officiis repudiandae.

Id vel sensibus honestatis omittantur, vel cu nobis commune patrioque. In accusata definiebas qui, id tale malorum dolorem sed, solum clita phaedrum ne his. Eos mutat ullum forensibus ex, wisi perfecto urbanitas cu eam, no vis dicunt impetus. Assum novum in pri, vix an suavitate moderatius, id has reformidans referrentur. Elit inciderint omittantur duo ut, dicit democritum signiferumque eu est, ad suscipit delectus mandamus duo. An harum equidem maiestatis nec. At has veri feugait placerat, in semper offendit praesent his. Omnium impetus facilis sed at, ex viris tincidunt ius. Unum eirmod dignissim id quo. Sit te atomorum quaerendum neglegentur, his primis tamquam et. Eu quo quot veri alienum, ea eos nullam luptatum accusamus. Ea mel causae phaedrum reprimique, at vidisse dolores ocurreret nam.

Capítulo 5

Implementación

Sed iusto nihil populo an, ex pro novum homero cotidieque. Te utamur civibus eleifend qui, nam ei brute doming concludaturque, modo aliquam facilisi nec no. Vidisse maiestatis constituam eu his, esse pertinacia intellegam ius cu. Eos ei odio veniam, eu sumo altera adipisci eam, mea audiam prodesset persequeris ea. Ad vitae dictas vituperata sed, eum posse labore postulant id. Te eligendi principes dignissim sit, te vel dicant officiis repudiandae.

Id vel sensibus honestatis omittantur, vel cu nobis commune patrioque. In accusata definiebas qui, id tale malorum dolorem sed, solum clita phaedrum ne his. Eos mutat ullum forensibus ex, wisi perfecto urbanitas cu eam, no vis dicunt impetus. Assum novum in pri, vix an suavitate moderatius, id has reformidans referrentur. Elit inciderint omittantur duo ut, dicit democritum signiferumque eu est, ad suscipit delectus mandamus duo. An harum equidem maiestatis nec. At has veri feugait placerat, in semper offendit praesent his. Omnium impetus facilis sed at, ex viris tincidunt ius. Unum eirmod dignissim id quo. Sit te atomorum quaerendum neglegentur, his primis tamquam et. Eu quo quot veri alienum, ea eos nullam luptatum accusamus. Ea mel causae phaedrum reprimique, at vidisse dolores ocurreret nam.

Capítulo 6

Conclusiones y Trabajo Futuro

Apéndice A

Apéndice A

A.1. Sección Apéndice

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: 20 de Abril].

[7] Android. Check system version at runtime [en línea]. <https://developer.android.com/training/basics/supporting-devices/platforms.html>. [Consulta: 4 de Mayo.

[8] Android. Great devices with the best of android [en línea]. <http://www.android.com/phones-and-tablets>. [Consulta: 20 de Abril].

[9] Android. Supporting multiple screens [en línea]. <https://developer.android.com/guide/practices/screens_support.html/>. [Consulta: 4 de Mayo].

[10] Android. Testing fundamentals [en línea]. <http://developer.android.com/tools/testing/testing_android.html/>. [Consulta: 4 de Mayo].

[11] Android. Ui/application exerciser monkey [en línea]. <http://developer.android.com/tools/help/monkey.html/>. [Consulta: 4 de Mayo].

[12] Androideity. Arquitectura de android [en línea]. <http://androideity.com/2011/07/04/arquitectura-de-android/>. [Consulta: 4 de Mayo].

[13] AppBrain. Ratings of apps on google play [en línea]. <http://www.appbrain.com/stats/android-app-ratings>. [Consulta: 3 de Mayo.

[14] 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].

[15] Bugsense. Bugsense android [en línea]. <https://www.bugsense.com/docs/android/>. [Consulta: 11 de Mayo].

[16] Crittercism. Crittercism - android sdk documentation[en línea]. <http://docs.crittercism.com/android/android.html/>. [Consulta: 11 de Mayo].

[17] 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].

[18] 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.

[19] Google. Espressostartguide on android-test-kit [en línea]. <https://code.google.com/p/android-test-kit/wiki/EspressoStartGuide/>. [Consulta: 4 de Mayo].

[20] Google. Google analytics android [en línea]. <https://developers.google.com/analytics/devguides/collection/android/v3//>. [Consulta: 17 de Mayo].

[21] Google. Google play developer console [en línea]. <https://play.google.com/apps/publish//>. [Consulta: 11 de Mayo].

[22] Emir Hasanbegovic. Android device screen sizes [en línea]. <http://www.emirweb.com/ScreenDeviceStatistics.php>. [Consulta: 25 de Abril].

[23] Joel Murach. *Murach's Android Programming*. Mike Murach & Associates, 2013.

[24] OpenSignal. Android fragmentation visualized (july 2013) [en línea]. <http://opensignal.com/reports/fragmentation-2013>. [Consulta: 25 de Abril].

[25] OpenSignal. The many faces of a little green robot (august 2012) [en línea]. <http://opensignal.com/reports/fragmentation.php>. [Consulta: 25 de Abril].

[26] Samsung. Remote test lab: la herramienta que todo desarrollador necesita [en línea]. <http://www.europe-samsung.com/smsdev/Home/Articulo/66/>. [Consulta: 4 de Mayo].

[27] Square. Spoon - distributing instrumentation tests to all your androids [en línea]. <http://square.github.io/spoon//>. [Consulta: 4 de Mayo].

[28] staffcreativa. Los curiosos nombres de las diferentes versiones de android [en línea]. <http://blog.staffcreativa.pe/android-google/>. [Consulta: 4 de Mayo].

[29] Ángel J. Vico. Arquitectura de android [en línea]. <http://columna80.wordpress.com/2011/02/17/arquitectura-de-android/>. [Consulta: 4 de Mayo].

[30] 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].

[31] Lars Vogel. Android user interface testing with robotium - tutorial [en línea]. <http://www.vogella.com/tutorials/Robotium/article.html/>. [Consulta: 4 de Mayo].

[32] Lars Vogel. Testing with easymock - tutorial [en línea]. <http://www.vogella.com/tutorials/EasyMock/article.html/>. [Consulta: 4 de Mayo].

[33] Lars Vogel. Unit tests with mockito - tutorial [en línea]. <http://www.vogella.com/tutorials/Mockito/article.html/>. [Consulta: 4 de Mayo].

[34] Lars Vogel. Using robolectric for android testing - tutorial [en línea]. <http://www.vogella.com/tutorials/Robolectric/article.html/>. [Consulta: 4 de Mayo].