Repatterning:

Improving the Reliability of Android Applications with an Adaptation of Refactoring

by

Bradley Christian Dennis

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 2, 2014

Keywords:Refactoring, Code Smells, Patterns, Non-functional Requirements, Verification

Committee:

David Umphress, Chair, Associate Professor of Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering
Jeffrey Overbey, Assistant Research Professor of Computer Science and Software Engineering

Abstract

Studies of Android applications show that *NullPointerException, OutofMemoryError*, and *BadTokenException* account for a majority of errors observed in the real world. The technical debt being born by Android developers from these systemic errors appears to be due to insufficient, or erroneous, guidance, practices, or tools. This dissertation proposes a re-engineering adaptation of refactoring, called *repatterning*, and pays down some of this debt. We investigated 323 Android applications for code smells, corrective patterns, or enhancement patterns related to the three exceptions. I then applied the discovered patterns to the locations suggested by the code smells in fifteen randomly selected applications. I measured the before and after reliability of the applications and observed a statistically significant improvement in reliability for two of the three exceptions. I found repatterning had a positive effect on the reliability of Android applications. This research demonstrates how refactoring can be generalized and used as a model to affect non-functional qualities other than the restructuring related attributes of maintainability and adaptability.

Acknowledgments

As all journeys must come to an end, my decade long journey towards a doctorate is finally coming to a close. The many struggles, detours and failures were too great to overcome alone, and as such, I am eternally indebted and grateful to those that supported me on this effort. First, I would like to thank my wife, Katya, and son, Josey, their patience and tolerance has proven much greater than mine. Second, without the gracious support of my father, brother, and father-in-law, John "Dad", Brian, and Vlad, and their wives, Brenda, Shannon, and Sasha, I never could have completed this work. I extend a special acknowledgement to my advisor, Dr. Umphress; his support for me never wavered and over the years he has become a valued mentor and role model. A warm thanks is also extended to my committee, Dr. Cross, Dr. Hendrix, and Dr. Overbey, as their insights and guidance have expanded and improved my research greatly.

I would like to thank Hasanur Rahaman, who helped build and collect the initial application catalog, Dr. Paul Stermer who helped with the experimental design and methodology, and finally, Joseph Douglas (aka Sylvan Kills), for his many hours editing and proofreading my dissertation.

Table of Contents

List of Tables

List of Figures

Chapter 1 – Introduction

1.1 Background

Google announced at its developer conference, Google I/O, in May of 2013, that the company hit 48 billion application downloads from the Google Play Store, and that the Android operating system had broken 900 million activations (Gundotra, 2013). Android is dominating smartphone sales, having captured 75% of the market as of first quarter 2013 (IDC, 2013).

A market of this size will attract developers and their products. There are over 1 million (Victor, 2013) applications in the Play Store and over 100 thousand registered developers (Pargianowicz, 2012). Becoming an Android developer is a simple process: it only takes a PC or a Mac, a Gmail account, and a $25 developer registration fee. Once registered, a developer can publish anything to the Play store. Google does not require any pre-certification, letting the market provide quality control for the applications. This low barrier to entry attracts a very diverse set of developers: from the hobbyist, to the moonlighter, to Android development shops and much bigger organizations such as Disney Interactive Studios. The telltale sign of the size of the development team is the number of platforms the applications support. Applications that appear in only one marketplace are most likely the product of one or two developers (Mikalajunaite, E., personal communication, March 16, 2012), while multiplatform applications generally require larger teams for production and support.

On any given day, one in five Android users will experience a crash, with up to half of those users uninstalling the offending application (Bugsense, 2012). This customer loss is a serious problem for developers. Developing an Android application is different than what most developers are used to. It feels familiar since it is based on Java, but it has some hidden complexities lurking for the programmer. When one thinks of how an application runs on a desktop, it is either running or it is not. On a mobile device, it is usually some variation of *running*, *suspended*, or *stopped*. The life cycle of an Android application expands this even further with *running*, *paused*, *stopped*, *shutdown* or in one of two transient states in between (Franke, Elsemann, Kowalewski, & Weise, 2011). To complicate matters even further, an Android application is not a single homogenous application, but a high level collection of application components that share the same resources. These components are practically complete applications by themselves, and each can act as entry points for the program. This is where a key problem lies. Up to 30% of Android application crashes occur when an application transitions from one lifecycle state to another (Popadopoulos, 2013; Levy, 2012; Maji, Arshad, Bagchi, & Rellermeyer, 2012). The complexities of the Android lifecycle compound the natural difficulties of an unpredictable environment along with resource constraints that are typical of mobile development.

1.2 Framework

Many Android developers are not just developers. They are *appreneurs*. They invent, develop, test, market, support, improve, and service their apps. They program these applications while simultaneously trying to operate and grow a business. The cognitive burden on an Android developer is immense. Many understand the architecture of an Android application, but lack the time to dive into the academic literature or the Android source code to synthesize the

impact that a multi-entry component architecture might have on their application, or to consider the nuances of resource management on an Android device. Instead, they rely on best practices and tools (Agrawal & Wasserman, 2010) to verify the quality of their application before publishing it in the marketplace.

The ultimate source of guidance for many developers is the Android developer website (Google b, n.d.). Google provides a very large body of work on the Android Developer website for a developer: beginner tutorials, dissections of the core Application Framework, best practices, frequently updated blog articles, quality checklists, and other similar resources. Many of the official guides provide some recommendations for making an application less susceptible to crashes, but they tend to be demonstrations of common use cases instead of in-depth explorations. Google's App Fundamentals are overly simplified or in many cases just factually wrong (Franke, Elsemann, Kowalewski, & Weise, 2011). Google even provides an advanced course with a series of best practices, but reliability is absent. There are several dominant themes apparent in the official guidance from Google which are also seen in other developer guidelines such as Best Practices for Mobile Developers (AQuA, 2013a) and Mobile Software Quality Model (Franke, Kowalewski, & Weise, 2012). There is a strong emphasis on usability and security while reliability is superficially addressed or completely ignored. Android usability and security dominate the academic literature as well.

There is a variety of verification tools and techniques available for the Android developer. The Android SDK comes with a fuzzing tool (fuzzer), Monkey, which generates a stream of random user events and a very limited set of system events. The Adaptive Random Testing technique improves upon Monkey by distributing the test case generation more evenly and finding the first fault 40 to 50% quicker (Liu, Gao, & Long, 2011). A third fuzzer,

Dynodroid, makes even further improvements by limiting the available events for test cases to those that are relevant to the application in its current state (MacHiry, Tahiliani, & Naik, 2012). Another tool, AndroidRipper is a model-based testing tool that will traverse an Android application in order to generate and execute context-specific tests on the fly (Amalfitano, Fasolino, Tramontana, De Carmine, & Memon, 2012). While promising, all these tools share one common shortcoming; they focus primarily on the graphical user interface (GUI), and thus are limited to user events. This excludes the entire domain of faults that are generated when system events occur. Many other tools are either too complex or too limited in scope to be useful to the diverse range of Android developers (Franke, Royé, & Kowalewski, 2012; Zhang & Elbaum, 2012).

In this dissertation I propose a new reengineering process called *repatterning* which is modeled after refactoring and attempts to overcome these shortcomings. Repatterning combines problem analysis, amelioration, verification and validation into a simple, light-weight technique suitable for the novice and advanced developer alike. It uses heuristics derived from our collective experience, developing reliable mobile systems to ensure that any recommended solutions are effective ones. An effective and pragmatic technique for improving the reliability of Android applications should be welcomed by the developer community.

1.3 Statement of the Problem

Two different commercial crash reporting services presented the results of their big data analyses of millions of crash reports in consecutive years at DroidCon in 2012 and 2013. The top root-cause exceptions in both analyses were surprisingly consistent, with *NullPointerException* being the dominant exception (Levy, 2012; Popadopoulos, 2013). The consistency of these errors were further corroborated by a robustness evaluation of Android

inter-process communication (IPC) study done by (Maji, Arshad, Bagchi, & Rellermeyer, 2012) in late 2012. Three errors made up 90% of their failures with *NullPointerException* making up 36.5%. Another error, *BadTokenException*, appeared in all three studies, and may be directly attributed to an error in the Android documentation (Levy, 2012). The second most frequent error noted by both Levy and Popadopoulos, but not appearing in the Maji study, was an *OutofMemoryError*. Levy attributes this to poor bitmap and *ListView* management, while Popadopoulos does not identify any specific cause.

This pattern of failure suggests a large technical debt being shouldered by the Android developer community. The available guidance and verification tools for reliability are insufficient, erroneous, or in some instances non-existent. There is a clear need for pragmatic and low cognitive load techniques in order for Android developers to improve the reliability of their applications.

1.4 Purpose of the Study

The overall purpose of this study was to explore the use of refactoring as a technique for improving the reliability of Android applications. More specifically, the research questions that guided this investigation are:

- How are Android applications failing?
- What is the impact of these failures?
- What guidance is available to Android developers to improve the reliability of their applications?
- What tools can an Android developer use to verify the reliability of their applications?
- How can repatterning be used as a technique by Android developers to improve the reliability of their applications?

1.5 Limitations, Assumptions, and Delimitations

Limitations

The primary limitation of this study was the availability of artifacts of the Free Open Source Software (FOSS) Android applications selected for experimentation. If the only artifact available is the source code, which is expected, it will be difficult to generalize the usefulness of repatterning in relation to other software abstractions.

Assumptions

This study made several assumptions about FOSS Android applications. First, it assumed that the quality of FOSS applications are fairly representative of Android applications as a whole. Some developers believe that open source applications tend to be more reliable based upon the colloquialism "given enough eyeballs, all bugs are shallow" known as the "Many Eyeballs" theory (Raymond, 1999) of open source development. Android markets, however, are very active. New applications get added frequently as others grow inactive and fall unsupported, which should inhibit any generalized maturing of FOSS Android applications.

The second assumption was that the reliability of these applications can be accurately measured. A simple *defects per thousand events* calculation was used as a measure of reliability. This was derived by testing the application in a test framework based upon the Monkey tool that is packaged with the Android SDK. Finally, it was assumed that Android reliability code smells are discernable, that reliability patterns exist and both of these are discoverable by the investigator.

Four assumptions about the repatterning process were also made:

- Any repatterning being applied will not alter the semantics of the feature under treatment.
- Application of the repatterning is safe and does not inject new bugs.
- The effort involved in repatterning is aligned with the resulting quality improvements. In other words, the improvement is worth the investment.
- Repatterning does not significantly degrade other quality attributes, notably, maintainability.

Delimitations

This study was delimited to FOSS Android applications that support Android 2.3.3 or greater. This allowed the investigation of an application's reliability as well as access to the source code for inspection and experimentation of a set of applications reflective of 97% of the current Android market. Another restriction of the Android applications under investigation was the feature complexity of the applications. Applications, such as widgets and live wallpapers, were excluded as they only rely upon a very limited subset of the Application Framework and are generally too simple to exhibit the error characteristics of other more complex applications. Furthermore, there are two reliability concerns that were the focus of this study; one addressing application lifecycle and the other addressing memory management. These are the prevailing concerns as suggested by the body of academic and professional literature, and determined by the investigator. Finally, the study only investigates the effect of repatterning on the reliability of the applications, and not the quality attributes of the process itself. We believe the adaptation stays true enough to refactoring to maintain its characteristics of pragmatism and ease of use.

1.6 Definition of Key Terms

Reliability – the degree to which a system, product or component performs specified functions under specified conditions for a specified period of time (ISO, 2011).

Refactoring – "a disciplined technique for restructuring an existing body of code, thereby altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations." (Fowler b, n.d.).

Repatterning - a disciplined reengineering process to improve a system's quality consisting of three steps; identification, amelioration, and verification.

Application life cycle – the process-related states of an application and the allowed transitions between these states (Franke, Elsemann, Kowalewski, & Weise, 2011).

Behavior preserving transformations – transformations that do not alter the visible behavior of the program.

Behavior correcting transformations – transformations that cure existing faults in the program.

Behavior enhancing transformations – transformations that improve a program's quality by introducing new behavior.

Chapter 2 – Literature Review

2.1 Introduction

Google Senior Vice President Vic Gundotra announced that Android hit the milestones of 900 million activations and over 48 billion application installs via Google Play at Google I/O 2013 in May 2013 (Gundotra, 2013). Any mobile developer with a PC or a Mac, a Gmail account, and $25 for a developer registration fees can release applications for 75% of the smartphone market (IDC, 2013) in as little as an afternoon. The impressive growth and low barriers to entry make Android one of the most appealing ecosystems for mobile developers today. However, this opportunity does not come without a price. As high as 3% of every application launched (Geron, 2012) will crash, with up to 20% of Android users experiencing a crash on any given day (Bugsense, n.d.). Bugsense further estimates that Android developers lose 5% of their customers due to poor application quality.

The high rate of application failure raises four major research questions. First, what reliability issues do Android applications suffer from? Second, what is the impact of these issues? Third, what guidance is available for reliability concerns? And finally, what tools are available to verify Android applications? These questions are answered by summarizing the state of Android reliability, and then by synthesizing the data from the perspective of a software developer. The answers to these questions are troublesome. Android applications are failing due

to issues unique to Android as well as failures common to the mobile domain. Reliability should be of greater concern to developers, but even if it were, they do not have the right guidance or tools to produce highly reliable applications.

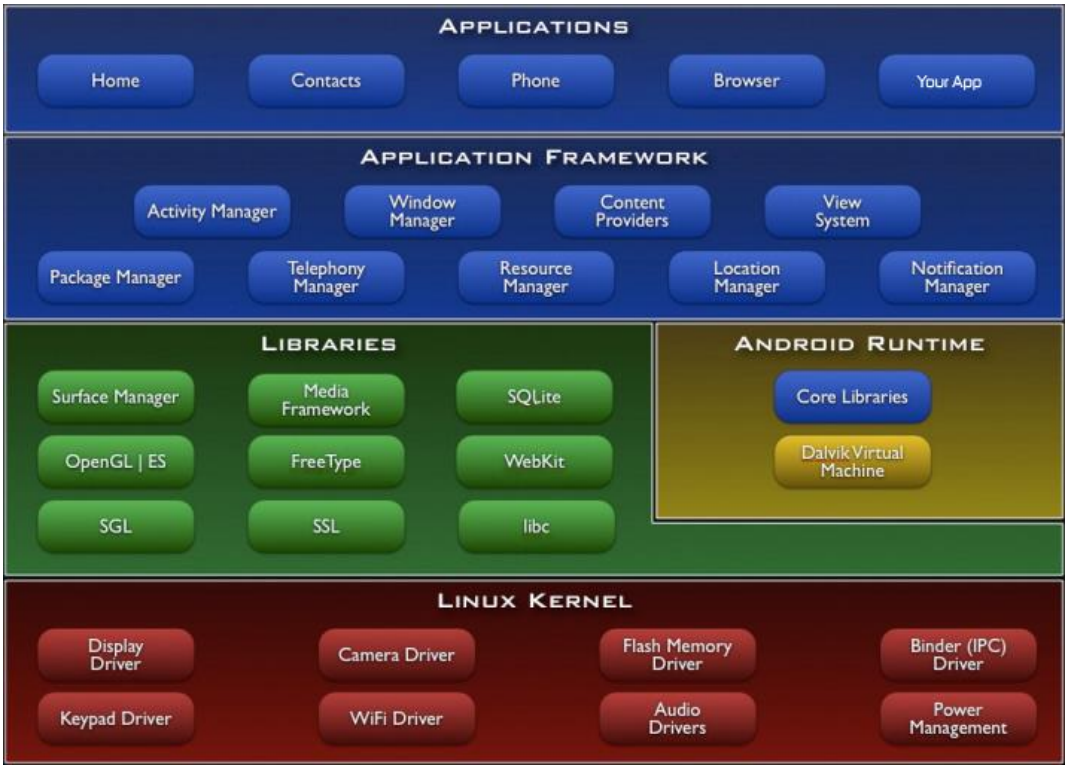

Figure 1. Android Architecture, (Google c, n.d.)

2.2 Android Mobile Operating System (MOS) and Application Architecture

Android applications are built on top of a framework that is feature-rich and use an application model that appears simple. However, as an application uses more components its lifecycle model becomes unsuspectingly complex. Developers building Android applications must be proficient at the advanced lifecycle management of a multi-entry component architecture as well as the inter-process communication methods that Android relies heavily upon in its system architecture and application model.

Android MOS

An Android application sits on top of three layers of the operating system as shown in Figure 1. The lowest level is the kernel. This is a slightly modified, stripped-down build of the Linux 2.6 Kernel. This layer provides basic kernel operations and all the essential hardware drivers. Above this resides the Android Runtime and many of the native libraries that provide modern technologies to the developer, such as a transactional database and high-end graphics. The uniqueness of the Android environment resides in the next layer, the Application Framework. The platform services provided by Android's Application Framework is "Android" for the developer. The important components are:

- Activity Manager – manages the life-cycle of applications.
- View System – manages many of the UI elements. Every Android user interface component is derived from the View or ViewGroup API classes.
- Content Providers – provides managed access to data outside an application's context, such as Calendar or custom components.
- Resource Manager – handles access to any application's resources.

Every application runs in its own Dalvik Virtual Machine. Inter-process communication (IPC) is managed by the Application Framework and occurs via three mechanisms:

- Intents – universal messaging structures that can be sent and received between processes, or even to launch processes and activities.
- Bundles – containers for passing data along with Intent.
- Binders – system level component to handle the remote method invocation between application components.

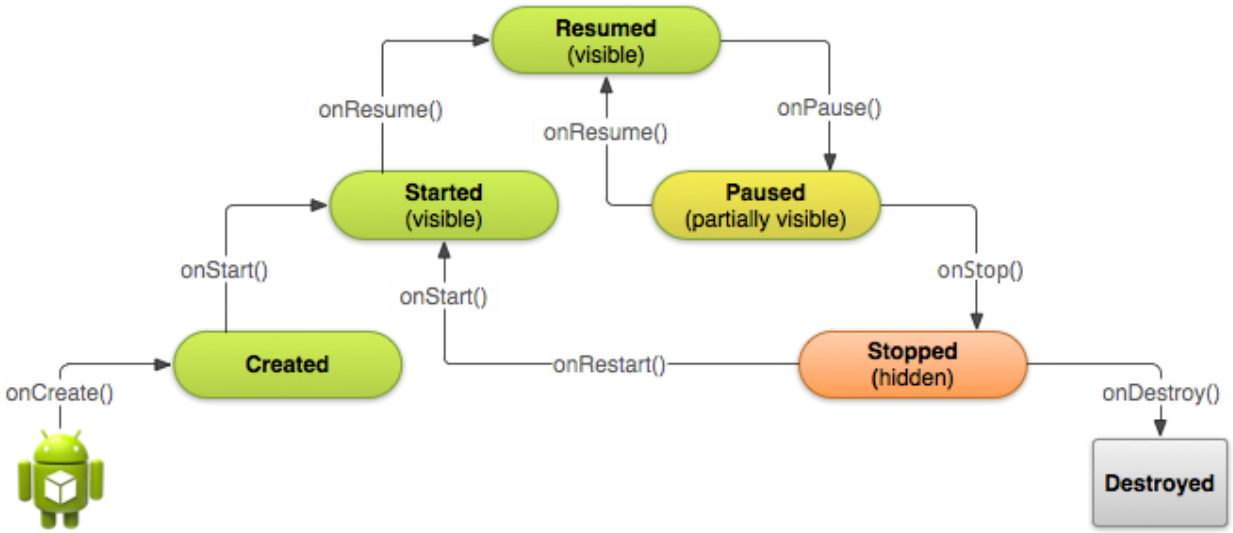

Figure 2. Simplified Android Activity Life-cycle from the Android Developer Site (Google j, n.d.)

Application Architecture

The Android system provides four specialized components in order to create an application. These components are high-level, any of which can be an application entry point and each with its own distinct life-cycle, see an example in Figure 2. Even though the components run in the same process in the same virtual machine, they interact with each other in the same manner as with an external component; via Intents, Bundles, and Binders. Logically, any of these components can be a stand-alone application. The four building blocks are:

- Activities – a single screen with a user interface.
- Services – a long-running background process without a user interface.
- Content Providers – a component that manages access to shared application data. Content providers are used for both internal and external sharing of data.
- Broadcast receivers – a component that can register to receive system event broadcasts, such as a SMS text received, or receive custom events from applications. This component can also initiate its own event broadcasts.

2.3 Android Reliability Issues

Developers have been creating applications for mobile devices since Snake, a simple game that first appeared on the Nokia 6110 in 1997 (Snake). Developers have grown familiar with the resource constraints of mobile development. Even as mobile devices have increased in power and capabilities, limited resources remain part of the domain. The development for mobile systems share this common constraint, but each individual system still presents its own set of issues.

Inter Process Communication (IPC)

In late 2012, Maji and his team worked with IBM to publish a study on the robustness of Android IPC (Maji, Arshad, Bagchi, & Rellermeyer, 2012). Their work uncovered some reliability issues, even though the research was intended to investigate security of communication between Android components. Their experiment focused on Intents, since previous research (Chin, Felt, Greenwood, & Wagner, 2011) had demonstrated that the Android Intent was a particularly vulnerable security risk. Their experiment sent over 6 million intents to more than 800 components. Their failure results for Android 4.0 were similar to the exceptions found by the analysts at Crittercism (Levy, 2012) and Bugsense (Popadopoulos, 2013):

- *java.lang.NullPointerException* – 36.50%
- *android.view.BadTokenException* – 26.83%
- *java.IllegalStateException* – 23.56%
- *java.lang.RuntimeException* – 3%

Crash Reporting

A presentation by the CEO of Crittercism, Andrew Levy, at DroidCon 2012 enumerated just how Android applications are crashing. Crittercism has developed a crash reporting tool that has monitored billions of application launches since its release. The top five Android crashes (Levy, 2012), in order, are;

- *java.lang.NullPointException* – Crittercism's logs show this error occurring most commonly in the *onResume()* life-cycle method. Secondly, it occurs most frequently on data from Intents.

- *java.lang.OutofMemoryError* – Levy's data suggests the memory problems arise out of improper use of *Bitmaps* and *ListViews* by developers.

- *android.view.BadTokenException* – this error is the result of an error in the Android documentation. Developers are creating dialogs via *getApplicationContext()* without protecting against context switches of the activity.

- *java.lang.IllegalArgumentException* & *java.lang.RuntimeException* – Crittercism's analysis doesn't show any themes to these errors, but does provide several common errors and pitfalls such as incomplete manifest files.

- *android.database.sqliteSQLiteException* - Levy does not give any indication as to a common cause to the error, but he provides some insights. He suggests some of these exceptions are occurring due to direct access of the SQLite database file as well as concurrency issues accessing the database file.

A year later the CEO of Bugsense, Panos Popadopoulos, presented the findings of an analysis of Bugsense monitoring data at DroidCon 2013. Like Crittercism, Bugsense is an application monitoring service for developers to track and log runtime errors in their deployed applications. Bugsense analyzed 40 million crashes from March 1st, 2012 to May 30th 2012 and had similar results but drew slightly different conclusions.

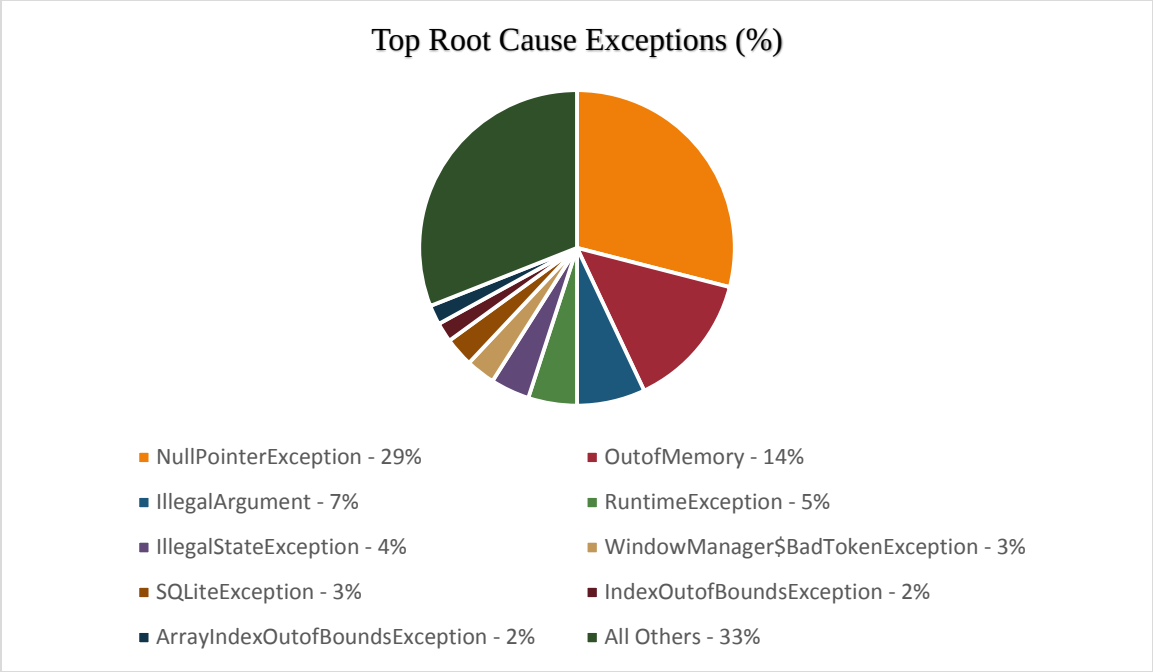

Figure 3. Top Root Cause Exceptions (Popadopoulos, 2013)

Their top 10 root cause exceptions (Popadopoulos, 2013) are:

- *java.lang.NullPointerException* – 29%
- *java.lang.OutofMemoryError* – 14%
- *java.lang.IllegalArgumentException* – 7%
- *java.lang.RuntimeException* – 5%
- *java.IllegalStateException* – 4%
- *android.view.BadTokenException* – 3%
- *android.database.sqlite.SQLiteException* – 3%
- *java.lang.IndexOutofBoundsException* – 2%
- *java.lang.ArrayIndexOutofBoundsException* – 2%
- *java.io.FileNotFoundException* – 2%

They attribute these exceptions to the following reasons:

- Memory Exhaustion – Bugsense claims out of memory errors account for 37% of the issues they collected. They suggest this is the result of poor utilization of bitmaps. .
- Race Conditions & Deadlocks – Race conditions amounted to 28% of the errors found. Their analysis suggests this is caused by concurrency issues accessing the SQLite database.
- Missing/Corrupted Resources – 14% of the errors collected by Bugsense were the result of accessing a non-existing resource such as an audio file.
- Improper Component Identification – Bugsense found 3% of the crashes to be the result of undeclared components in the Android Manifest, or attempts to access non-existent components.
- Insufficient Permissions – Another 3% of the errors collected were permissioning errors.

The Bugsense report further groups the errors by context: 86% within the application, 11% accessing an API, and 3% within the Android framework itself. It is not clear how they deduced memory exhaustion as the top cause of failures. Memory exhaustion would be a sound inference if Android were written in languages such as C or C++ where the failure of a *malloc* or *memset* could result in a null pointer, but in a memory managed system like Java or Android that is not the case. The evidence from Levy and Maji suggests that the null pointers occurred in the lifecycle callback implementations, and were not the result of low memory.

Android MOS

Although a bit dated by the speed of Android advancement, the (Maji, Hao, Sultana, & Bagchi, 2010) case study on the faults of the Android may still provide some valuable insights for the Android developer trying to develop more robust applications. Maji, Hao, Sultana, & Bagchi harvested bug reports from Android's defect tracker and reports from public developer forums for the analysis. The team evaluated 628 defects and found that 14% occurred in the Android Framework (excluding the View System), 8% in documentation or installation, 5% in the View System, 4% in the Kernel, 4% in Dalvik and Core Library, with the remaining defects

found in the developer tools or applications bundled with the OS. These present a significant quantity of external defects that directly impact the reliability of an application, notably the large amount of documentation errors.

Android Permissions

The oft-cited work on Android permissions, Android Permissions Demystified, presents an analysis of the permission levels of 940 applications from the Android market (Felt, 2011). Felt found that nearly one-third of the applications were over-privileged, meaning these applications requested more privileges than necessary. While mainly targeted at security issues, their work has some impact on application reliability. Unnecessary privileges may increase the impact of a bug as faults can cascade to exposed parts of the system. Their analysis found seven common developer errors that led to requesting too many privileges:

- Permission Name – developers request permissions for similar sounding functionality. An example they give is ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE privileges which are often requested in tandem, although they are for different cases.
- Deputies – developers tend to request permission for an action requested via intent of another component when it is unnecessary.
- Related Methods – developers frequently ask for permissions even if they are only using unrestricted methods of a class when that class contains methods requiring permission.
- Copy and Paste – developers copy and paste permission sets from examples found online.
- Deprecated Permissions – developers are using permissions that are no longer necessary or have otherwise been obsolesced.
- Testing Artifacts – developers leave permissions for test code in their production builds.
- Signature/System Permissions – developers are requesting unnecessary *Signature* or *SignatureOrSystem* permissions. The rationale was indiscernible.

(Felt, 2011) asserts that developers are requesting more privileges than necessary primarily due to simple confusion over the permission system. The Android documentation provides a permission list for 78 methods while Felt discovered a total of 1259 methods that actually required permissions. Felt claims another possible reason for over-privileging is due to an unintentional incentive built into the Android update system. Applications prior to Android 4.0 could not receive automatic updates if the permission set of the update was greater than the application being updated, leading developers to speculate about future permissioning needs. Further, they found six documentation errors adding to the technical debt born by Android developers.

Application Life Cycles

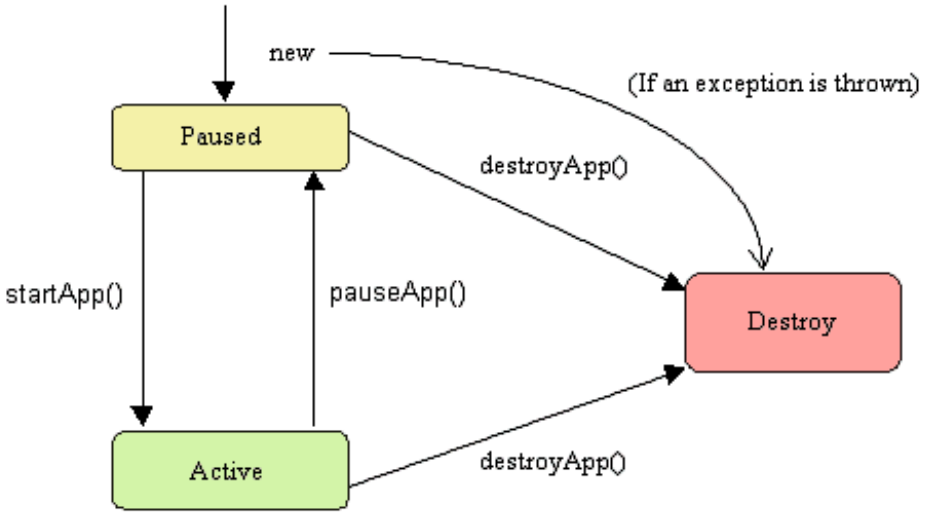

Figure 4. The Lifecyle of a MIDlet (Balani, 2004)

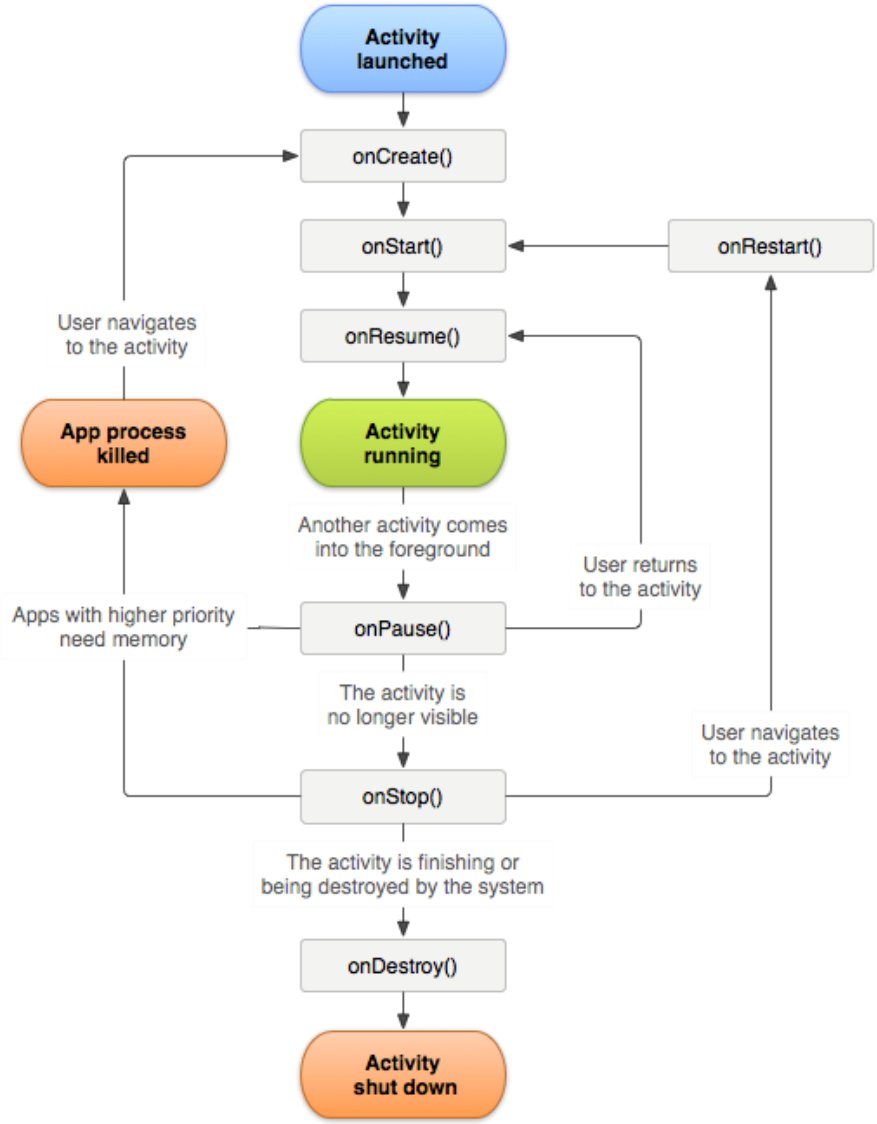

Figure 5. The Lifecycle of Android Activity (Google l, n.d.)

Dominik Franke and his team at the Embedded Software Laboratory in Aachen Germany have been doing work with respect to understanding a mobile application's lifecycle. They assert that the mobile application's life cycle, and more specifically an Android application's lifecycle, is fraught with quality concerns (Franke, Elsemann, Kowalewski, & Weise, 2011; Franke, Elsemann, & Kowalewski, 2012). They attribute these quality problems to two issues: improper understanding of an application's lifecycle, and ambiguous or incomplete

documentation. In a desktop environment resources are abundant and the operating system generally manages the state changes for an application in response to system events. This is not the case in mobile systems. Mobile frameworks define the lifecycle model, and the developer must implement the callbacks in order to manage the application's reaction to state change events. The high-level component architecture and state model can make an Android application's lifecycle complex. Figure 4 shows the lifecycle of a J2ME MIDlet, and Figure 5 the lifecycle of an Android Activity. It is easy to discern how much more complex a system with seven lifecycle callbacks is when compared to the older, and simpler MIDlet with three callbacks. This complexity is demonstrated in a case study by (Franke, Elsemann, Kowalewski, & Weise, 2011) which uncovered 26 different triggers resulting in 10 different possible state transition sequences for an application in response to trigger. They discovered that a developer seeking lifecycle guidance from the Android developer documentation finds a few simple examples, inconsistent diagrams, and incomplete documentation. Inspection of Figure 3, which comes directly from the Android developer documentation, shows the two states of *running* and *shut down* are represented, but *paused* and *stopped* are not. The associated textual documentation does not describe the *shutdown* state.

2.4 Impact on the Developer of Failures

The reality that software engineers face is that their applications will crash. Android applications, however, crash less than their Apple counterparts even though the Android MOS is more fragmented than iOS, and has a more decentralized, less regulated submission process (Viswanathan, 2012; Henneke, 2013). This doesn't mean application reliability should be ignored by Android developers. There are two key external factors that developers must

consider when developing their applications: the quality requirements of the marketplace and the expectations of the user.

Google Play, just as most other Android marketplaces, does not have any precertification requirements for submitting an application to its marketplace. This is counter to both Apple and Windows mobile marketplaces, which have published requirements and a review process. Counterintuitively, this decentralized, less regulated submission process has not led to lower quality applications in the Play Store (Geron, 2012). This does not mean there isn't risk for the developer. In February of 2013 Google purged their marketplace of 60,000 low quality applications (Perez, 2013). The speculation is that Google has been improving their algorithms for automated quality checks as their data set grows (AQuA, 2013 b).

In November of 2012, Bugsense released an infographic presenting an analysis of application uninstalls by Android and iOS users (Bugsense, 2012). They found that Android users uninstalled a free application 25% of the time if the application crashed. This ratio nearly doubles to 42% for paid applications. The category with the highest uninstalls was "Everyday" applications, like ToDo lists or alarms, at 54%. These percentages suggest that Android users are less forgiving of crashes.

2.5 Application Quality Guidance

No developer wants his or her application to crash, but the question is how do you develop software that resists crashing? Active developer communities such as Stack Overflow and the Android Developer newsgroups are great resources for debugging and finding questions to specific issues, but they don't provide general guidance or tips to improve the reliability of an

application. There are, however, other resources available ranging from official online training from Google, to university level courses and quality certification programs such as AQuA.

Guidance from Google

Google does not have any minimum standards or certification requirements to publish an application in their marketplace, nor do they establish any minimum reliability requirements in the App Content Policy or Developer Distribution Agreement. They do, however, provide developers with several artifacts to establish quality standards, provide tutorials and best practices, and verification tests to assess the base quality of an application. These artifacts fall into five categories: Training, Blog posts, API Guides, Testing, and Publishing.

Training (Google e, n.d.)

Google provides three training courses for Android developers. Each course is a themed set of classes with two to four lessons. The first course, Getting Started, is a set of introductory classes designed to make the developer familiar with Android developer tools, and to deliver a basic understanding of an Android app. Building Apps is the next course in the series. This course is designed to familiarize the developer with common technologies found in mobile applications, along with the peculiarities of Android and the mobile domain. The final course is the Best Practices set of classes. These classes present a set of best practices for the quality areas of User Experience, User Interfaces, User Input, Performance, and Security and Privacy. Each lesson generally has the same structure: some background or overview information followed by a narrated example implementation. Many of the example implementations provide a discussion of common failure cases. The Getting Started course is the only artifact that provides training on the lifecycle of an application. While broad in its coverage, the material does not inform

developers of any of the complexities identified by (Franke, Elsemann, Kowalewski, & Weise, 2011).

Blog Posts (Google a, n.d.)

The Android Developer Blog is a series of articles targeted for the developer. New articles appear about three times each month, spanning a broad range of developer-related topics. Approximately one article a month is related to some aspect of training, although there is no pattern. These articles are from different authors and do not follow the consistent format of the other training materials. Of the 24 blog posts published in the first half of 2013, only one (Dougherty, 2013) contains guidance for making an application more reliable. The article describes the best practice for handling phone call requests, discussing a common anti-pattern, as well as proper privileging. In 2012, only one article dealt with IPC (Lucas, 2012), specifically implicit Intents, and it does not provide any suggestions for making applications more reliable. 2011 follows a similar pattern with two blog posts out of sixty-eight, an article about the impact of an API change (Wilson, 2011), and another providing best practices for processing ordered broadcasts (Albuquerque, 2011), which contain developer insights for making reliable applications.

API Guides (Google d, n.d.)

The API Guides section of the developer site also contains several artifacts for a developer. This section is organized by technology domain, such as App Components, Computation, and Data Storage. Each section contains a landing page with a cross reference to related training and blog posts as well as the lead in to the overview of that particular domain. Each component and technology in the API is described with examples given of each usage.

Common cases and some counter cases are usually covered. This section includes a Best Practices subsection with links to relevant blog posts and articles on recommended techniques for general compatibility, supporting multiple screens, as well as providing support for tablets and handsets. While informative, the content tends to gloss over any complexities involved in the error prone areas of IPC and lifecycle. For example, the Activities artifact provides a high level narrative of the lifecycle targeted to a casual reader. The Intents and Intent Filters article further demonstrates this pattern. It provides a lengthy example of using Intents but never addresses any failure cases.

Testing (Google k, n.d.)

The testing guidance is organized in the same way as the other sections of the developer site: a landing page with related cross references to other areas of the site, and a series of topical articles. This section describes the testing tools in the developer's toolkit, the testing APIs available, test fundamentals, some best practices and guidance on testing, including a tutorial on testing an Activity. Google provides one article, *What to Test*, which presents some general testing guidance for Android components, then component specific guidance in each article on testing each of the main Android components. However, the instructions contained in these articles are too general to be useful to anyone but the first time Android developer.

Publishing (Google i, n.d.)

The Publishing section provides a workflow for a developer to get their application ready for publishing. The publishing section also provides guidance toward what to test and when to test by constructing a process narrative and including cross references to other areas of guidance. It doesn't provide any new information, other than the sequencing of tasks.

App Quality Alliance (AQuA)

The App Quality Alliance is a non-profit industry group formerly known as the Unified Test Initiative. The members are well-recognized in the mobile industry: AT&T, LG, and Motorola. In October of 2012, AQuA released their Quality App Directory of self-verified and member-verified apps. The directory contains 128 apps, exclusively Android, as of this writing. AQuA publishes two documents that can aid an Android developer in improving the reliability of their applications: Best Practice Guidelines for producing high quality mobile applications and Android Testing Criteria.

Best Practice Guidelines (AQuA, 2013a)

The AQuA Best Practice Guidelines are a set of generic guidelines encompassing the entire mobile application domain. The best practices AQuA recommends for stability are:

- Application stability – "The application should not crash, unexpectedly close, or otherwise behave abnormally at any time while running on any targeted device."
- Application behavior after forced close by system – "The application should preserve sufficient state information to cope with forcible close by the system. It should not lose any information that it implies would be preserved, nor become difficult to use subsequently, as a result of a forcible closure by the system."

These two guidelines should be obvious and practically irrelevant to a developer seeking guidance on reliability.

Android Testing Criteria (AQuA, 2013 b)

The Android Testing Criteria document details the sets of tests applied to Android applications in order to qualify for the AQuA quality badge. AQuA defines three classes of apps: Simple, Framework, and Complex. The core test suite is for Simple apps. This is expanded to build the Framework test suite, and expanded again for Complex apps. The

document lists 79 tests to be performed. Table 1 details the 22 negative tests required to pass and receive an AQuA quality badge for a complex application. While fairly comprehensive, the tests miss many of the triggers identified by (Franke, Elsemann, Kowalewski, & Weise, 2011) for flexing the lifecycle callbacks and discovering state management failures.

Table 1. List of negative tests from AQuA Testing Criteria

Test No.	Test	Description
2.1	Memory during run	Tests the application's functionality when the file system is filled.
2.2	Multiple Launch	Tests the application's ability to suspend and resume.
2.3	Idle	Test when the application is active, then idle long enough to cause the device to sleep.
3.2	Network delays and loss of connection	Tests the application's functionality when loses network connectivity while it is actively using the connection.
3.4	Network connectivity resource downloading	Test to evaluate the application's handling of long downloads.
4.2	Message Queuing	For applications which use SMS messages, this test repeats test 4.1, Messaging auto start, rapidly several times.
5.3	Telephone call incoming while application in use	This test checks the application's response when an inbound call is made.
6.1	Memory card operation	Tests the application's behavior when a memory card is inserted and removed. The application does not need to support memory card access.
6.2	Memory card screen behavior	For applications that access the memory, this test verifies the functionality when a card is inserted and removed.
6.3	Other Interruptions	Makes sure the application behaves correctly during 10 system interruption events, such as low battery alert and USB connection.
9.1	Suspend/resume from main menu	Verifies the application suspends correctly from its own main menu.
9.2	Suspend while executing	Verifies the application suspends correctly by pressing the phone's home button.
9.3	Resume	Verifies the application resumes normally from test 9.2.
9.4	Influence on terminal system features	Tests the application and phone behavior when the application launched, but not in the foreground.
9.5	Resource sharing database	For applications that access the Contacts, this tests to see if there is any contention between the application and the Contacts application.

13.6	Device Keys	This tests the application when the phone's hardware keys, Back, Menu, Volume, and Home are pressed. The presses include short presses and long presses.
14.1	Device Close	This tests how an application behaves on a device that supports closing.
14.2	Device Open	This tests how an application behaves on a device that supports opening.
15.1	Application Stability	If during any of the tests the application crashes, this test fails.
15.2	Application behavior after forced close	This verifies the application behavior during a force close event.
18.3	Multiplayer pause and disconnect player	For applications that have multiplayer functionality, this tests the behavior and a player is forcibly removed.
18.4	Multiplayer over Bluetooth	This repeats 18.3 but over a Bluetooth connection.

Mobile Software Quality Model

The Embedded Software Laboratory at Auchen University published its own set of quality guidance, Mobile Software Quality Model (Franke, Kowalewski, & Weise, 2012), to establish a quality framework for testing mobile applications. The researchers argue that developers cannot support all the qualities of software and must focus on key characteristics. In their model, they take the superset of all attributes from various models and distill them down to a set for mobile applications. They identify *flexibility*, *extensibility*, *adaptability*, *portability*, *usability*, *efficiency*, and *data persistence* as the seven most important quality attributes of a mobile application. They selected flexibility and extensibility due to the fragmentation of the mobile domain. An easily upgradeable application that supports multiple devices will reach the most customers and provide the most update-to-date experience for them. Their reasoning for adaptability, efficiency, and data persistence is that a mobile system isn't consistent and software must support the resource changes and shortages that routinely occur on a mobile device. Franke, Kowalewski, & Weise further emphasize usability due to the restricted input and output

capabilities generally found on handheld devices. They do not provide or suggest any metrics for these attributes. Reliability is missing from this model.

University Courses

Mobile applications are so ubiquitous that many universities today offer courses in mobile development as part of their Computer Science or Software Engineering curricula. Many universities offer college courses in Android, Auburn University included. Only fifteen different universities are part of Google's University Consortium (Google f, n.d.), each offering a programming course that involves Android in some manner. University of Washington goes so far as to offer an Android Developer Certificate Program for professionals.

University of the Free State

The University of the Free State is a public university in Bloemfontein, South Africa. It offered a one-time Android Development Short Course in 2011 (Coetzee, 2011) which was comprised of 8 lectures. The course was designed as an introduction to Android, covering a broad set of topics at a very introductory level. There were no lectures on any advanced quality issues or any other emphasis on the difficulties of developing applications for the mobile domain.

Zhejiang University

Wei Hu, et al described their Android-based Smartphone Development course at the IEEE 10th International Conference on Computer and Information Technology in 2010 (Hu, 2010). The course is divided into seven parts, the first two introducing the students to Smartphones and the various Smartphone operating systems. The next four sections cover the Android architecture and application framework components. The final part of the course

focuses on development. The course is intended to focus on practical programming over theory and includes a series of 16 hands-on labs and application projects. Although a key element of the course design is hands-on practice, there is little evidence of practical non-functional requirements being addressed.

2.6 Android Books

As the popularity of Android has risen, so has the secondary marketplace for developers. There are hundreds of books on Android programming and dozens of paid boot camps and workshops available to the moneyed programmer. For example, Amazon has an extensive catalog of Android developer books listed in their e-commerce marketplace. A simple search for 'Android' in it's Books category yields over 7,000 results. These are readily available and tend to be priced under $30. A review of the top three bestseller Android programming books (Meier, 2012; Burd, 2011; Lee, 2012) on Amazon, as of October, 2013, reveal no dedicated sections to building reliable Android applications.

2.7 Android Verification Tools

The ISO/IEC 25010:2011 software quality model, or SQuaRE, defines software reliability as the "degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time." (ISO, 2011) The high uninstall reaction to a crash (Bugsense, 2012) suggests that reliability might mean how long an application runs until it crashes to the end-user, but to a developer a more meaningful measure of reliability is how many crashes occur over a specified period of time (Meyer B. , 2008). The Android SDK comes packaged with four tools that a developer may use to verify an application. There are three tools for functional testing: unit testing with the Android Test Framework, a scripting API,

MonkeyRunner, and a UI test framework, uiautomator. The SDK comes packaged with only one non-functional testing tool: Monkey. There are several other tools and techniques available to the developer for non-functional testing and they fall into two general categories: model-based tools and fuzzers, of which the latter category Monkey falls into.

Fuzzers

Monkey (Google i, n.d.)

Monkey is a fuzzing program that exercises an application by streaming a series of pseudo-random user and system events. The tool can be configured to support different event profiles, allowing the developer more control over the types of events being generated. It also provides repeatable sequences since the random number generator seed is configurable. The tool will exit on an application crash, an unhandled exception, or an application freeze. This tool can be run on a device or in an emulator. It is considered a "dumb monkey" since the test cases are generated without regard to usage profiles. However, it is possible to configure the event generation distribution to reflect typical usage patterns of a given application if they are known.

The current documentation for Monkey lists eight factors for event distribution: touch, motion, trackball, basic navigation, major navigation, system keys, application switches, and 'any event' – a catch-all for all other types of events. Inspection of the 2.3.3 SDK source code of the *MonkeySourceRandom* class (Google g, n.d.) , the event generator component of the Monkey tool, reveals another factor, "flips", which simulates a keyboard flipping open or shut. Interestingly, inspection of the most current version, 4.3 r2.3 (Google h, n.d.), of the same class reveals two more event types; "rotation" and "pinchzoom". However, the default configuration for percent rotation is set to 0 while the default setting for pinchzoom is 2%. Since these features

are undocumented it is unlikely that developers who are using Monkey to stress test their applications are aware of them. The default event distribution for the 2.3.3 version of tool is; 15% touch, 10% motion, 15% trackball, 25% navigation, 15% major navigation, 2% system keys, 2% application switches, 1% keyboard flips, and 15% catch-all. The current default event distribution mirrors this, with the exception that 2% is reduced from the catch-all and set for pinchzoom events. No documentation was found as to how these values were derived or if they reflect any known usage patterns; moreover, an extensive search did not reveal any published studies indicating what an actual event pattern should be. It is reasonable to conclude that developers using this tool most likely use the default configuration. The tool's usefulness may be limited if these settings are not representative of the application's users.

Adaptive Random Testing

(Liu, Gao, & Long, 2011) developed a tool, Smart-Monkey, to improve upon the monkey testing tools available to mobile applications. Smart-Monkey is a plugin to their record and playback tool, MobileTest. It uses an adaptive random test (ART) case technique to generate context sensitive test cases, while keeping the "distance" between each test case uniform. Spreading the test cases as evenly as possible has been shown to expose more faults, quicker, since faults tend to cluster around each other. They define "distance" as a normalized combination of two components: sequence distance and value distance. Each event is listed in a table, along with an alpha-numeric value. This is used as a sequence distance of the event with a common string distance metric, the Levenshtein distance, used as the value distance. Test case input distances are calculated in the same manner. The researchers conducted an experiment to evaluate ART versus simple monkey testing and found that their extension significantly reduced the number of test cases generated to find first fault. It is unclear how the events were

determined, sequenced, or how they are triggered. If those determinations are made by a developer, its usefulness would be sensitive to the expertise of the developer. Further, how quickly an algorithm converges on the first fault is not a useful metric in the real world as it is the number of faults as a function of time, not how quickly it finds the first fault that is more meaningful (Meyer B. , 2008).

Dynodroid (MacHiry A. T., 2012)

Dynodroid is an input generation testing tool developed at Georgia Tech. Dynodroid uses a novel technique, coined "observe-select-execute," to create system events and inputs in order to exercise an app. The observe component, Observer, computes a set of events that are relevant to the application in its current state. The Selector chooses an event and then instructs the next component, Executor, to execute it. An empirical evaluation of the tool found code coverage was slightly better than Monkey, and converged in $1/20^{th}$ of the time. Dynodroid is novel with its inclusion of context specific system events, which provides a much broader set of events to test. This tool considers a system event relevant if the application is registered to listen to it, e.g. an SMS message. This decision eliminates other system events that trigger lifecycle callbacks thereby reducing the probability of exposing lifecycle related faults.

Model-based Tools

Exception Amplification

(Zhang & Elbaum, 2012) proposes an automated approach that detects faults in the exception handling of code that accesses external resources. This approach instruments the application under test by replacing external resources with a mocked versions of the resources. Zhang then permutes all possible exceptions thrown by a resource of the given type into a call

tree and exercise the application by triggering sequences of exceptions defined by paths through the tree. Each path is termed a *mocking pattern*. They expand the small scope hypothesis, which is the notion that a large proportion of bugs can be found by exhaustively testing an application within a narrow scope, by reinterpreting the measure of scope from the number of program operations executed under test to be depth of the test amplification tree. Their investigation shows that this approach was able to identify a majority of bugs that had been reported by users, as well as 75% of the bugs that had been fixed post-release by developers. While the technique is intriguing, the intensive white-box investment in building the mocks, as well as its concentration on verifying exception handling code, limits its usefulness to the small-shop Android developer.

JPF-Android (van der Merwe, van der Merwe, & Visse, 2012)

JPF-Android is an extension of the model checking engine Java Pathfinder (JPF) which runs on the Java Virtual Machine. JPF is extended with a model of a limited set of Android Framework Components; ActivityManager, ActivityThread, the application components, the view structure and lastly, the message queue. The developer uses the JPF Abstract Window Toolkit extension to develop scripts to model the GUI of the application under test to run on JPF. JPF is a validated framework for detecting concurrency issues, and JPF-Android is a useful tool but is limited to being able to find threading issues that would not be easily discoverable running tests on the Dalvik Virtual Machine.

AndroidRipper (Amalfitano, Fasolino, Tramontana, De Carmine, & Memon, 2012)

AndroidRipper is a model-based testing tool with a unique twist. The typical process in model-based testing is to develop a model of the GUI via a formal language, a script, or other

means and then use that to generate test cases. These models are generated manually by reverse engineering or by automated tools (*rippers*). AndroidRipper uses automated ripping techniques to traverse an application's GUI, generating and executing test cases as events are encountered. The tool only supports user events.

AndroLIFT (Franke, Royé, & Kowalewski, 2012)

AndroLIFT is another product of the work at the Embedded Software Laboratory in Germany. The prime motivation is the idea that the correct implementation of an application's life cycle is crucial to high quality applications. Their tool is an Eclipse plug-in intended to be integrated into the Android Eclipse developer suite. AndroLIFT provides an instrumented *DebugActivity* test class that emits state change data, or the developer may manually instrument an *Activity* with logging statements to provide the tool with state change data. It parses the logging stream from the Android Debug Bridge and uses the data to build a visualization of the lifecycle as the application runs. Franke, Royé, & Kowalewski believe that this visualization provides a valuable tool for the deve loper to examine the application's lifecycle and correctly implement it. AndroLIFT also provides a package of assertions which the developer may use to verify their implementation of the application's lifecycle. Like the ART tool, the usefulness of this tool appears restricted by the knowledge of the developer. This tool will verify the correctness of a lifecycle implementation only to the extent of which a developer is aware. If a developer is not familiar with a specific unique trigger, this tool would not aid him or her in discovering it. The visualization of an applications lifecycle, however, does make a major contribution to addressing the unique complexities of developing for Android.

Chapter 3 – Research Description

3.1 Introduction

The range of issues is broad, from design or architectural level problems within an application's lifecycle, to narrow, as with the management of *ListViews*. Furthermore, it is widely believed that, with the low barrier of entry to mobile development, the expertise and maturity of mobile developers are as diverse as the problems they face. These traits would suggest that a specialized tool or even new training artifacts published to the Android developer site would not be enough to overcome these reliability problems. Programmers have been re-using code since the earliest days of the craft, so is there another pre-existing solution that can repurposed to solve these problems? There is evidence that suggests a light-weight best practice for improving maintainability, refactoring, might be well suited for this task. Refactoring itself is a manifestation of a general, but still simple process, that combines problem identification, amelioration, and verification. Mobile developers, especially, tend to rely on light-weight best practices over heavier formal processes (Agrawal & Wasserman, 2010). Further, many developers already view refactoring as a best practice for improving code quality in general and not as a technique that improves maintainability alone (Kim, 2012). These points suggest that an adaptation of refactoring as a reengineering process for improving reliability might be adopted by the mobile developer community. Additionally, the identification heuristic of refactoring, code smells, is already being used on a variety of artifacts indicating this may also be preserved in a new process that does not restrict itself to source code. Given these, refactoring appears to

be a good candidate for repurposing from a restructuring activity that improves maintainability to a reengineering one that improves software reliability.

3.2 Developer Perceptions

One way to predict how developers will perceive a technique generalized from refactoring is to investigate how they perceive and use refactoring itself. Refactoring advocates believe that it improves maintainability of software, and thus, improves developer productivity (Fowler, 1999). However there has been little effort to study developer attitudes towards refactoring or how refactoring is practiced in the real world. (Kim, 2012) conducted an attitudinal survey to investigate attitudes and application of refactoring by questioning any engineer that had "refactor*" in his or her change comments on five Microsoft products over a two year span. 78% of the respondents defined refactoring differently than its original definition. They viewed it as a generalized code transformation that improved some quality aspect of the software. 46% of the developers did not mention behavior preservation in any way, while 71% said that the tool supported simple refactorings in Visual Studio, which were performed during some other higher level efforts to improve the software. Many of the negative perceptions of refactoring had to do with issues surrounding a large code base or lack of sufficient tests, and not with refactoring itself. Further, the 86% of the respondents reported that they conduct some refactorings manually, with full 51% of them manually performing all refactorings. These results suggest that a technique generalized from refactoring which does not rigidly preserve behavior but improves other quality attributes, and one that is not tool dependent, would be accepted by developers as it already conforms to their perceptions and to real-world refactoring practices.

An investigation by Murphy-Hill (Murphy-Hill, Parnin, & Black, 2012) provides further support for some of these findings. Murphy-Hill analyzed the refactoring usage from eight sources. These sources varied from commit logs, to tool usage statistics, and to developer surveys. The research determined that developers used two tactics when refactoring: "floss refactoring" and "root-canal refactoring." Floss refactoring refers to refactorings done that intertwined with other developer tasks such as adding a new feature, whereas root canal refactorings were dedicated, larger-scale efforts to address a specific code-smell. The interleaved application of refactorings identified by both Kim and Murphy-Hill suggests that any technique derived from refactoring should be designed as an augmentation of the developer's natural workflow and not as a discrete task.

3.3 Developer Adoption

A prolific market for mobile applications is a new phenomenon. The launch of Apple's *App Store* for the iPhone in 2008 is generally regarded as the beginnings of the massive mobile market we see today (Hamblen, 2011). One would expect that given the ease of developing applications for the mobile market that a wide range of developers would be drawn to it. Very few studies have been found that assess developer demographics. A small one was conducted by (Agrawal & Wasserman, 2010) at Carnegie Melon University. They surveyed developers participating in several mobile developer forums and found four key traits of mobile developers: many applications were small, team sizes were mostly one or two people, developers relied heavily on best practice recommendations and little on formal processes, and they had very little organization or tracking indicating relative immaturity. Another recent survey conducted by the analysts at research2guidance (research2guidance, 2011), a mobile industry research firm based in Germany that involves 320 companies doing mobile application development, found that the

median team size of a company doing only Android research was two. This confirms Agrawal's findings. Evidence suggests that one or two person developer teams must rely on cognitive shortcuts over high investment processes, as they have to fulfill the roles of many stakeholders while staying aware that the development cycles and the time to market are very short.

3.4 Refactoring as Part of a Larger Process

The term 'refactoring' first appeared in academic literature in a paper by William Opdyke describing his Ph.D. research on program restructuring at a symposium in 1990 (Opdyke, 1990). The product of Opdyke's early research was a framework for automating program transformations (Opdyke, 1992), and how to ensure it was behavior preserving and thus a legal restructuring. The goal of his research was to reduce maintenance costs by automating the common, but risky, developer task of restructuring code to make it easier to understand and evolve. However, Opdyke did not address other activities of the refactoring, such as where and why to apply the refactoring, and how to verify the refactoring was applied correctly. These aspects manifested themselves as part of the broader refactoring process when Martin Fowler published his book *Refactoring: improving the design of existing code* (Fowler, Refactoring: improving the design of existing code., 1999) in 1999. Kent Beck first coined the term "code smell" in a co-authored chapter of this book. "Code smells" were a simple metaphor for a symptom that there might be a quality problem somewhere in the source. Code smells became the main strategy, or heuristic, for identifying where to refactor and which refactoring to apply. The verification of the refactoring also became a dominant theme when Fowler pronounced that his first step in any refactoring was to always write tests, and in every example throughout the remainder of the book he demonstrated executing tests as the final step. In 1999 the entire

refactoring process became codified: identify the issue, apply the refactoring, and verify that transformation did not break the application, see Figure 6).

Figure 6. The Refactoring Process

Tom Mens' survey on refactoring viewed the process as six discrete steps instead of three: (Mens & Tourwe, 2004)

1.) Identify the location of refactoring.
2.) Determine which refactoring to apply.
3.) Guarantee the transformation preserves behavior.
4.) Apply the transformation.
5.) Assess the outcome in terms of quality characteristics.
6.) Synchronize changes with any other artifacts.

The identification and determination steps map to the code smell metaphor, while the preservation step mirrors Fowlers insistence on writing tests first. The final two steps, measuring and synchronizing artifacts, represent a level of maturity not found with many mobile developers. The missing step that verifies the transformation further reflects an incongruity between theory and practical application of Mens' interpretation of refactoring.

3.5 Bad Smells in Other Abstractions

There are two key characteristics of code smells: they should represent something that is easily identifiable, and they do not necessarily indicate a problem (Fowler a, n.d.). Although their dominant usage is for source code analysis, there is no logical limitation to their application. Since “a bad smell” is just a recognizable metaphor for a simple identification heuristic, there is no restriction from using the metaphor in other artifacts, such as class diagrams, architectural diagrams, or others items for software development. Further, there is no logical reason why smells even need to be limited to maintainability. Smells are being used to describe potential problems in other programming paradigms such as AspectJ (Piveta, Hecht, Pimenta, & Price, 2006) and in higher software abstractions such as software architecture (Garcia, Popescu, Edwards, & Medvidovic, 2009 a; Bourquin & Keller, 2007). In fact, there is even an effort to define bad smells in spreadsheets (Asavametha, 2012; Cunha, Fernandes, Ribeiro, & Saraiva, 2012). The ‘bad smell’ is a useful metaphor and should be preserved in any generalization or adaptation of the process.

3.6 Refactoring the Restructuring Activity

Refactoring is the "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" to make it easier to understand and extend (Fowler b, n.d.) (Fowler, 1999). Refactoring only impacts understandability and extensibility of a system at the source code level of abstraction to the exclusion of any other quality improvement because refactoring is a code-level restructuring activity. Restructuring activities do not make any other improvements (Arnold, 1989). Refactoring has "crossed the chasm" (Ambler, 2007) into mainstream adoption and thus its concepts are being applied to a broader set of software artifacts and quality attributes; such as

architecture, designs, and security improvements. These types of activities are forward engineering or reengineering acts depending upon where in the lifecycle they take place since they make observable corrections or enhancements to a system's behavior. Piecemeal and disparate extensions of the refactoring model are risky, as they introduce confusion, exclude benefits of the holistic nature of the process, and may ignore critical aspects of software modification. For any reengineering application of refactoring, which those that affect qualities other than maintainability would be, the entire process needs to be re-envisioned to define a standard lexicon and a consistent model that will work towards reducing any risks of a maladapted refactoring model.

There are four types of software modification activities: corrective, adaptive, perfective, and preventive (ISO, 1999). Corrective and adaptive are reactionary modifications driven by some observed failure or change in the environment. Perfective modifications provide enhancements for the users or quality improvements such as performance or maintainability, while preventive modifications are prescient changes to preempt some latent fault in the system. These four categories could further be discriminated by how the external behavior of the system is affected. If the modification does not alter the external behavior, the activity can be classified as a restructuring one (Chikofsky & Cross, 1990). Conversely, if the modification does alter the external behavior, it should be classified as a reengineering activity (Chikofsky & Cross, 1990) since some new enhancement is being introduced, or some observable behavior is being altered. Furthermore, that software restructuring modifications only improve understandability and extensibility. Other types of similar activities that do not improve maintainability are not a restructuring acts (Arnold, 1989). While there is some confusion as to whether refactoring is preventive or perfective (Stroulia & Kapoor, 2001) (Leitch & Stroulia, 2003) (Higo, Kamiya,

Kusumoto, & Inoue, 2004) (Tsantalis & Chatzigeorgiou, 2011), it is clear that in this context it is a form of perfective restructuring. A perfective or preventive activity that makes any other enhancement, or improves any other quality attribute, is not restructuring and therefore is not refactoring.

3.7 Refactoring Limitations

The refactoring process is unique in its completeness. It has a means to help identify a potential problem, it provides a mechanism to transform the system to fix the problem, and it informs the developer how to verify that the transformation did not otherwise break the system. In other words, refactoring is a single process that diagnoses, ameliorates, and verifies any treatment applied. In the refactoring lexicon, these steps are called *code smells* (Fowler, n.d.), *behavior preserving transformations* (Fowler, 1999) as well as *refactorings* (Fowler, 1999), and finally, *verification*. Verification of refactorings is straight forward, as the transformations are behavior preserving. Functional tests can be run before and after to verify any transformations applied were not destructive. The refactoring verification step, independently, is well studied and not unique to refactoring, and thus is not being maladapted to reengineering activities. Code smells, as previously discussed, are also neutral to the domain and suffer no limitations. However transformations are limited.

Behavior preserving transformations, the amelioration step of the process, are synonymous with and are the heart of refactoring (Fowler b, n.d.). The behavior preserving aspect of the transformations is less of a feature, and more of a recognition of a characteristic of the underlying engineering activity that was formalized and automated by refactoring (Opdyke, 1992). Here, behavior preserving was described as the same set of inputs producing the same set of outputs before and after the transformation. In other words, behavior preservation was not a

formalism applied to constrain refactoring, but a description of the nature of the transformation itself. In this definition, any transformation that leads to different outputs, even when that difference is a corrective one, is not behavior preserving and thus is not refactoring. Since these types of transformations make improvements other than maintainability, they should classified as reengineering and not restructuring.

There are several adaptations of the refactoring transformations, notably aspect-oriented refactorings (AOP) (Laddad, 2003) (Hanenberg, Oberschulte, & Unland, 2003) (Monteiro & Fernandes, 2005) and security refactorings (Maruyama & Tokoda, 2008) (Hafiz, Adamczyk, & Johnson, 2009). The aspect-oriented refactoring adaptations are models of how the object-oriented refactoring can be applied to another programming paradigm while staying within the domain of restructuring. Many of these adaptations include AOP-specific smells while only (Hanenberg, Oberschulte, & Unland, 2003) recognizes that aspect aware refactorings necessarily lead to new behavior and takes special considerations to avoid it. This effort ensures these adaptations are a true restructuring activity, and does not create a special burden to identify any new verification procedures. On the other hand, the security transformations take a different tactic. In (Maruyama & Tokoda, 2008) refactoring is redefined by removing the references to maintainability and replacing them with security. While (Hafiz, Adamczyk, & Johnson, Systematically eradicating data injection attacks using security-oriented program transformations, 2009) re-envisions the behavior preserving characteristic of refactoring and broadens it from a purist definition to being the preservation of the before and after *correct*, or *good path* behavior. Hafiz et al. characterizes these types of transformations as *behavior enchancing transformations* (Hafiz, Overbey, Behrang, & Hall, 2013) further distinguishing them from the behavior preserving transformations of refactoring. These security

transformations are similar to refactoring in that they both involve some form of source code manipulation, and are generally agnostic to the system to which they are being applied. However, they are not a restructuring activity. These types of transformations are a reengineering one since they alter existing, or introduce, new behavior. We may miss any new verification or validation actions made necessary by these modifications if we do not explicitly recognize that these adaptations alter the behavior of the system. Further, since both of these efforts closely mirror the original focus of refactoring, automated program transformations, the full potential of the refactoring model to improve security is being overlooked. An inclusive process that combines the identification of security vulnerabilities, with their amelioration, and verification for any level of abstraction and against any software artifact, would provide useful additions to the secure programming body of knowledge.

Refactoring is a restructuring process that combines diagnostic, amelioration, and verification activities. Restructuring activities only affect maintainability and preserve the external behavior of a system. Behavior enhancing adaptations of refactoring that affect other characteristics of a system that do not explicitly recognize the shift from a restructuring activity to a reengineering one inject risk, confusion, and undermine the potential benefit of the new process. However, behavior enhancing or corrective adaptations of refactoring that do recognize this shift provide an opportunity to bring a unique and holistic process to the notoriously difficult set of non-functional requirements other than maintainability. Refactoring must be carefully and deliberately reimagined to extend its usefulness to non-functional attributes that are addressed by reengineering actions.

3.8 Repatterning

Chikofsky and Cross (Chikofsky & Cross, 1990) mark the differences between the maintenance activities of *restructuring* and *reengineering* in their taxonomy of reverse engineering. Restructuring is a horizontal transformation of an artifact to improve some maintainability attributes while preserving external behavior, both functional and semantic. Reengineering is the examination and transformation of an artifact to correct, improve, or evolve it without restriction as to behavior or levels of abstraction. While often confused, this distinction is important. In order to improve the reliability of an application one may need to alter the external behavior to accommodate some unforeseen interaction, or to provide new feedback to the user. That is not the case for maintainability. The immediate benefactor of a maintainability improvement is the programmer, not the user. These activities would never need to introduce new external behavior. Refactoring is always a restructuring task. Conversely, the immediate benefactor of a reliability improvement is the user. It is most likely that the end user experience will need to be altered in some way to accommodate any reliability improvement. This makes any process for improving reliability a reengineering one and not a restructuring one and as such should be distinguished from refactoring. The simplest way to make a distinction is with terminology, therefore we suggest the term *repatterning* to define this new process.

Refactoring as a model for Repatterning

The simplest distillation of the refactoring process is to implement a three step sequence. The first task is the identification of a potential weakness. The second task is to apply the transformation indicated by the heuristic. The final task is to verify the transformation was applied correctly. This three-step core process is used as the model for repatterning. Repatterning is *a disciplined reengineering process to improve a system's quality consisting of*

three steps; identification, amelioration, and verification and validation. Like refactoring, repatterning uses smells as the metaphor for identification and analysis. A repatterning bad smell should describe how to identify the weakness and suggest a transformation to be applied. At the source code level of abstraction, that may be just an algorithm or technique. However, at higher levels of abstraction the transformation should be a documented design or architectural pattern. The application of known solutions should minimize any risk of applying an incorrect one, as even the most inexperienced developer should be able ascertain some degree of contextual fit and appropriateness. Unlike refactoring, repatterning transformations are not restricted by behavior limitations or horizontal abstraction levels. A problem identified at the lowest abstraction, like the source code, may suggest a necessary transformation at a higher level, such as the architecture. There are two types of repatterning transformations; behavior correcting and behavior enhancing. Behavior correcting transformations are intended to cure an existing fault in the system, while behavior enhancing transformations are those that improve the quality of a system by adding new features. Any repatterning transformation should preserve the meaning of the feature or component under change and to keep the purpose of the feature cohesive. Further, a behavior correcting transformation should define its own verification process appropriate to its level of abstraction, while a behavior enhancing transformation should be validated that the new features introduced do not alter the stated intent of the system. The repatterning process is shown in Figure 7. In short, repatterning is refactoring without the

behavior preservation.

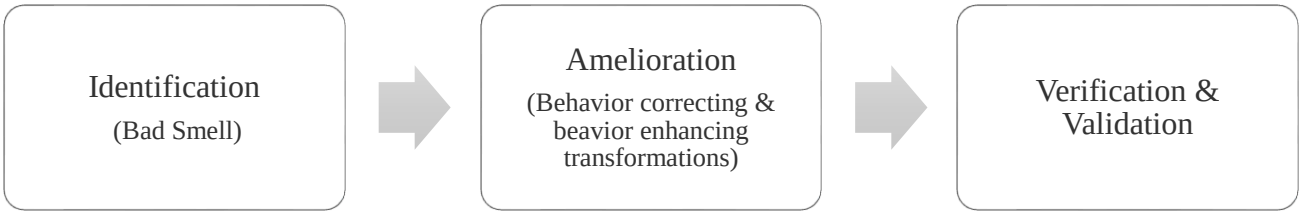

Figure 7. The Repatterning Process.

3.9 Conclusion

Android developers do not need to continue making interest payments on the technical debt imposed by the shortcomings of the Android domain. Refactoring is a lightweight restructuring practice that integrates assessment, amelioration, and verification to improve maintainability and extensibility. It is already viewed by many developers as a generalized strategy for improving quality without the behavior preserving trait of restructuring. Mobile developers tend to adopt best practices when given, so even with the conflicting evidence of refactoring's perceived and actual value it is still a good candidate for adaptation. Repatterning models the simple and pragmatic refactoring process to address the other complex quality issues, such as reliability, that can only be solved with a re-engineering activity. These qualities make

repatterning a viable solution toward paying down the principal incurred by these Android technical debts.

Chapter 4 – Methods & Results

4.1 Introduction

Four separate studies were conducted to validate the repatterning process as it applies to ameliorating the problems identified by three predominate Android exceptions: *NullPointerException*, *OutOfMemoryError*, and *BadTokenException*. One study was performed for each of the repatterning steps and addressed the following four questions:

- What code smells exist that can identify a deeper problem as suggested by the three exceptions?
- What corrective patterns are being used to guard against the failure conditions associated with the three exceptions?
- What enhancement patterns are being used as alternative paths when failure conditions associated with the three exceptions have been identified?
- How does the application of these patterns, and recipes to any problems identified by a code smell, improve the reliability of an application?

Of the four studies, three were exploratory mixed-method investigations and one was an experiment. These studies were the Smell Study, the Corrective Pattern Study, the Enhancement Pattern Study, and the Reliability Experiment. The Smell Study, Corrective Pattern Study, and Enhancement Pattern Study were conducted currently using the same context, subjects, selection process, instrumentation, and data collection procedures.

4.2 Exploratory Studies Framework

Context

The context of the studies were consumer-facing, open-source Android applications of unknown quality, written by developers with unknown levels of experience.

Subjects

The subjects of the studies are 323 prequalified FOSS Android applications selected from 749 FOSS Android applications cataloged from three public repositories: (F-Droid, n.d.), (AndroidFreeSoftware , n.d.), and (AOpenSource.com, n.d.). A prequalified application is one with ready access to the source code, supports Android version 2.3.3, is complex and is compatible with the Intel Atom (x86) emulator. A complex Android application is defined as an application that is not a live wallpaper, widget, or plug-in to any other application. See Appendix A for the catalog of FOSS Android collected from the repositories.

Selection

The subjects initially selected for the qualitative portion of the study were prequalified applications that exhibited one of the three signal exceptions during a twenty thousand event monkey pre-test. Once the research objective was able to be uniquely characterized, the full body of prequalified applications was then investigated. The subjects for the quantitative portion of the study were the entire set of prequalified applications. See Appendix B for the twenty thousand event monkey pre-test results.

Process

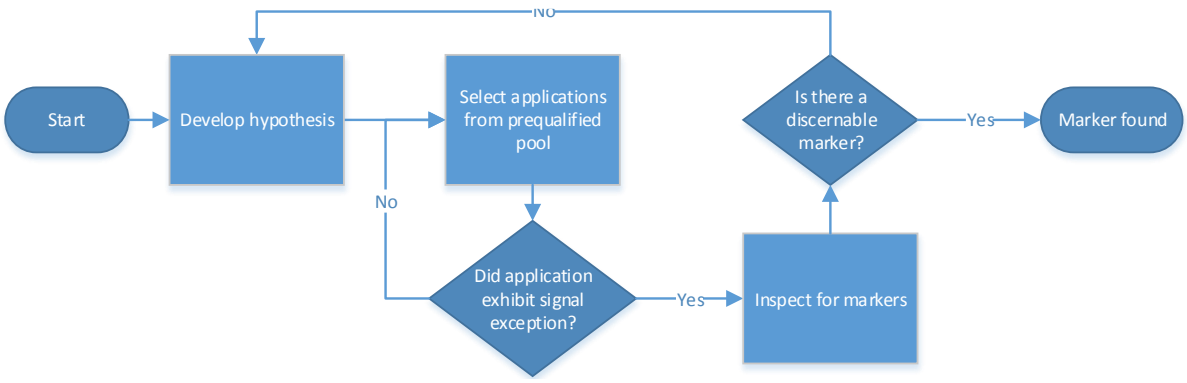

Figure 8. Exploratory Study marker discovery.

The exploratory studies were dependent on the identification of a distinct marker in the source code to identify inspection locations. The first step was to research developer issues, reports, queries, or complaints regarding a specific signal exception to develop a hypothesis as to the root cause of the exception. Next, the source code for applications that had thrown signal exceptions during the pre-test were inspected for any cues or markers that could be used to inspect the body of source code. This process was repeated until a specific code signature, a marker, was identified. These markers were translated to regular expressions and used as inputs to a Python script that scanned the available source code files, and extracted a snippet beginning ten lines before the marker and ten lines beyond and including the marker. These snippets were compiled into a single file. In the judgment of the investigator, the limited twenty line code window was enough to evaluate the code, but not enough for an in-depth analysis of any potential failures. This constraint retained the ease of discernibility characteristic of a code smell, while being sufficient to identify any patterns in use. The three regular expression patterns used were:

getExtras\(– This regular expression identified all uses of the *getExtras()* method call on an *intent*. Inspection of *intents* and their associated *bundles* was chosen as the candidate for the *NullPointerException* studies, as the investigation revealed this to be a particular concern to developers, and the risk potential was readily apparent in the initial code inspections. Further, the conclusion of (Levy, 2012) that directly implicated member variable access in the *onResume* method as a root cause of this exception was not supported by the initial investigation.

(new|extends|implements) \w(Thread|AsyncTask|Runnable|Handler)* – This regular expression identified uses of Threads, AsyncTasks, Runnables, and Handlers in the source code. In Java, inner and anonymous classes maintain an implicit reference to their parent classes, while in the Android framework, *Handlers* and *Runnables* are processed by the *Looper* object in its message queue that is associated with the application's main thread (Lockwood, 2013). This reference will prevent the activity from being garbage collected and thus leak the activity's resources. In a similar fashion, *Threads* and *AsyncTasks* also maintain references to their parent activities and can leak them as well (Lockwood b, 2013). These characteristics of Java and the Android Framework being a root cause of some *OutofMemoryErrors* were suggested as early as 2009 on the Android Developer Blog (Guy, 2009).

\.show\(, showDialog, onCreateDialog – These three regular expressions identified usage of dialogs. The initial investigation revealed that the *BadTokenException* was associated with dialogs being activated while the parent activity was in finishing state and, thus, did not have the proper Android context for the dialog. This is further supported the (Levy,

2012) assertion of a documentation error leading to the incorrect context being used;: *getApplicationContext*() as opposed to the activity itself.

The application of these expressions was limited to classes of type *Activity*. This filter allowed for a comprehensive review of the developer implementations while keeping the scope manageable for a manual inspection. The results were further filtered to remove inspection of commented code and uses of *Toasts*. *Toasts* were excluded as they are a framework dialog and were not implicated in the initial investigation. Further, third party libraries such as ActionBarSherlock were included once, while any sample code or demonstrations were excluded from consideration for smells and patterns.

NullPointerException Candidates

The *NullPointerException* marker was found in 168 apps from 162 developers. 782 markers were found in 576 activities with approximately 16,500 lines of code identified for smells and patterns inspection.

OutOfMemoryError Candidates

The *OutOfMemoryError* marker was found in 213 apps from 197 developers. 2021 markers were found in 645 activities with approximately 42,000 lines of code identified for smells and patterns inspection.

BadTokenException Candidates

The *BadTokenException* marker was found in 268 apps from 254 developers. 3150 markers were found in 887 activities with approximately 66,000 lines of code identified for smells and patterns inspection.

4.3 Smell Study Methodology

Research Question

What code smells exist that can identify a deeper problem as suggested by *NullPointerException, OutofMemoryError,* or *BadTokenException*?

Goals

The goal of this survey was to determine if discernible smells exist that correlate to deeper problems signaled by the observation of a *NullPointerException*, *OutofMemoryException*, or *BadTokenException*. A mixed-methods exploratory study was chosen as the research design as it provides a flexible research model built upon the strength of both qualitative and quantitative research methods.

Analysis procedure

Four *frequency of occurrence* metrics were calculated. The first frequency of occurrence measure was the total number of smells per signal exception across the body of applications. This indicates the prevalence of the smell in FOSS Android applications. The second frequency of occurrence was the ratio of smells present in code segments with respect to the total number of at-risk code segments. This value indicates the overall risk associated with the smell. The third frequency of occurrence measure was the number of smells detected per application. This value indicates the relative risk the smell poses. The final ratio was the frequency of applications that contained the smell. This value provides another indicator to the exposure of fault risk as suggested by the smell.

4.4 Smell Study Results

NullPointerException Smell Study

A snippet was considered to contain a *NullPointerException* smell if there was an observable risk of a *NullPointerException* being thrown. The *getExtras()* method and non-primitive accessors on *bundles* return null if the bundle or the data element does not exist. These two characteristics were evaluated to determine if the snippet was classified as containing a smell. Further, there were 12 observed cases of developers checking the returned intent from the *getIntent()* method call for a null value. The documentation does not state this value can be null and (Hackborn, 2009) states *getIntent()* only returns null in the case of the intent being set to null with *setIntent()*, so this was not considered code at-risk. An example of a snippet categorized as having a smell is in Figure 9, while an example of a snippet that was not classified as having a smell is in Figure 10.

```
58:         title.selectAll();
59:     }
60:
61:     /** Called when the create button is pushed */
62:     public void onClick(View v)
63:     {
64:         // prepare the insert request - get title from the widget
65:         EditText title = (EditText) findViewById(R.id.name);
66:         ContentValues values = new ContentValues();
67:
68:         values.put(Sessions.PROJECT_ID,
getIntent().getExtras().getLong("project_id"));
69:
70:         values.put(Sessions.TITLE, title.getText().toString());
71:         if(v == findViewById(R.id.create_and_start))
72:               values.put(Sessions.START_TIME,
System.currentTimeMillis());
73:
74:         // insert the result and go to the record activity
75:         Uri result = getContentResolver().insert(getIntent().getData(),
values);
76:         startActivity(new Intent(Intent.ACTION_EDIT, result));
77:         finish();
```

Figure 9. An example NullpointerException snippet that contains a smell. The getExtras() call may return null, resulting in a NullPointerException being thrown on the chained getLong() method call.

```
55:         pulseAnim = AnimationUtils.loadAnimation( this, R.anim.pulse );
56:
57:         setContentView(R.layout.converter);
58:
59:         ipAddress = (EditText) findViewById(R.id.ipaddress);
60:         ipBinary = (EditText) findViewById(R.id.ipbinary);
61:         ipHex = (EditText) findViewById(R.id.iphex);
62:
63:         currentIP = icicle != null ? icicle.getString(Converter.
EXTRA_IP) : null;
64:         if (currentIP == null) {
65:             Bundle extras = getIntent().getExtras();
66:             currentIP = extras != null ?
extras.getString(Converter.EXTRA_IP) : null;
67:         }
68:         if (debug) Log.d(TAG,"onCreate: currentIP="+currentIP);
69:
70:         if ((currentIP!=null) && (currentIP.length()>0)) {
71:               ipAddress.setText(currentIP);
72:               convertDecimal();
73:         }
74:
```

Figure 10. An example of a NullPointerException snippet that does not contain a smell. The getExtras() call and subsequent data accesses are guarded against a null being returned.

The following results were collected using the *NullPointerException* smell classification scheme:

Table 2. NullPointerException Smell Study Metrics

Metric	Value
Number of Markers	782
Number of Smells	429
Number of Apps with Smells	121

Table 3. NullPointerException Smell Study Results

Frequency of Occurrence	Result
Number of Smells	429
Smells to Marker Ratio	55%
Smells per App	2.56
At-risk Apps to Apps Ratio	37%

OutofMemoryError Smell Study

A snippet was considered containing an *OutOfMemoryError* smell if static inner classes were not used for *Threads, AsyncTasks, Handlers,* or *Runnables,* or they were used but accessed the parent activity without a *WeakReference* object to manage access to the activities resources. Examples of smelly and non-smelly snippets are in Figure 11 and Figure 12.

```
726:             hasNextArticle = historyItem.hasNext();
727:         }
728:         final Button nextButton = (Button)
findViewById(R.id.NextButton);
729:         if (hasNextArticle) {
730:             if (nextButton.getVisibility() == View.GONE) {
731:                 nextButton.setVisibility(View.VISIBLE);
732:             }
733:             currentHideNextButtonTask = new TimerTask() {
734:                 @Override
735:                 public void run() {
736:                     runOnUiThread(new Runnable() {
737:                         public void run() {
738:                             if (useAnimation) {
739:
nextButton.startAnimation(fadeOutAnimation);
740:                             } else {
741:                                 nextButton.setVisibility(View.GONE);
742:                             }
743:                             currentHideNextButtonTask = null;
744:                         }
745:                     });
```

Figure 11. A snippet containing an OutofMemoryError smell. The anonymous runnable may leak the Activity.

```
910:              }
911:
912:        }
913:
914:  };
915:
916:  /**
917:   * Geocoder handler class. Receives a message from geocoder thread
and displays "Add Waypoint" dialog even if
918:   * geocoding request failed
919:   */
920:  private static class GeocoderHandler extends Handler {
921:
922:        private final WeakReference<MainActivity> weakReference;
923:
924:        GeocoderHandler(MainActivity ma) {
925:              weakReference = new WeakReference<MainActivity>(ma);
926:        }
927:
928:        /**
929:         * Processing message from geocoder thread
```

Figure 12. An example of a snippet that does not contain an OutofMemoryError smell. The Handler is a static inner class and uses a WeakReference to access the parent activity.

The following results were collected using the *OutOfMemoryError* smell classification scheme:

Table 4. OutOfMemoryError Smell Study Metrics

Metric	Value
Number of Markers	2020
Number of Smells	2009
Number of Apps with Smells	213

Table 5. OutOfMemory Smell Study Results

Frequency of Occurrence	Result
Number of Smells	2009
Smells to Marker Ratio	99%
Smells per App	9.43
At-risk Apps to Apps Ratio	66%

BadTokenException Smell Study

A snippet was considered containing a *BadTokenException* smell if the state of the activity was not checked prior to showing the dialog, or if the parent activity was not being

used for the dialog's context. It was often the case that the context was unable to be determined in the inspection window. Examples of smelly and non-smelly snippets are in Figure 13 and Figure 14, respectively.

```
302:
303:          layout.addView(logo);
304:          layout.addView(textViewLayout);
305:
306:                  AlertDialog.Builder dialogBuilder = new
AlertDialog.Builder(this);
307:
dialogBuilder.setTitle(R.string.titleAbout).setView(layout).setNeutralButton
(R.string.btnDismiss, new OnClickListener() {
308:              public void onClick(DialogInterface dialog, int which) {
309:                  dialog.dismiss();
310:              }
311:          });
312:          dialogBuilder.show();
313:          }
314:
315:      @Override
316:      void onDictionaryServiceReady() {
317:          updateTitle();
318:          Intent intent = getIntent();
319:          if (intent != null && intent.getAction() != null &&
intent.getAction().equals(Intent.ACTION_SEARCH)) {
320:              final String word = intent.getStringExtra("query");
321:              editText.setText(word);
```

Figure 13. An example of a snippet containing a BadTokenException smell. The state of the activity could be finishing when this dialog is shown. The context for the dialog was unable to be determined for this snippet.

```
555:                            finish();
556:              }});
557:
558:        if (!isFinishing()) {
559:              try {
560:                    //
561:                    // Catch errors resulting from 'back' being pressed
multiple times so that the activity is destroyed
562:                    // before the dialog can be shown.
563:                    // See
http://code.google.com/p/android/issues/detail?id=3953
564:                    //
565:                    alertDialog.show();
566:              } catch (Exception e) {
567:                    Logger.logError(e);
568:              }
569:        }
570:  }
571:
572:  /**
573:   * Update all (non-existent) thumbnails
574:   *
```

Figure 14. An example of a snippet that does not contain a BadTokenException smell. The activity state is checked prior to showing the dialog.

The following results were collected using the *BadTokenException* smell classification scheme:

Table 6. BadTokenException Smell Study Metrics

Metric	Value
Number of Markers	3150
Number of Smells	2664
Number of Apps with Smells	258

Table 7. BadTokenException Smell Study Results

Frequency of Occurrence	Result
Number of Smells	2664
Smells to Marker Ratio	85%
Smells per App	9.94
At-risk Apps to Apps Ratio	80%

4.5 Corrective Pattern Study Methodology

Goals

The goal of this survey was to determine what corrective patterns were being used by FOSS Android developers in order to protect against the *NullPointerException*, *OutofMemoryError*, and *BadTokenException* failures in their applications. An exploratory examination was chosen as the research design, as it provides a flexible research model and is well suited for informal pattern discovery. The Corrective Pattern Study was done in conjunction with the Smell Study and the Enhancement Pattern Study.

Analysis procedure

Three *frequency of occurrence* metrics were calculated. The first frequency of occurrence measure was the total number of corrective patterns identified across the body of applications. This indicates the prevalence of the pattern in use. The second frequency of occurrence was the ratio of corrective patterns present in code segments to the total number of potential used in the code segments. This value is an indicator of adoption. The final frequency of occurrence measure was the number of developers observed using the pattern. Individual developers were identified by package naming schemes.

4.6 Corrective Pattern Study Results

NullPointerException Corrective Pattern Study

A snippet was considered containing a *NullPointerException* corrective pattern if it met three criteria. First, there was no observable risk of a *NullPointerException* being thrown when accessing the *bundle* or its data. Next, if primitives were accessed then it was considered a pattern if a default value was passed into the accessor. The Android Framework will return this

default value if the key does not exist in the *bundle*. A developer that passed a default value in would be able to explicitly identify and handle a fault condition, which is a sign of developer maturity. Finally, a specific signature was observed frequently during the initial investigation. This signature was to null test the *bundle*, then to test if a data key existed using the *containsKey*() method. In the case of primitives, if either the default value or the key test was used it was considered a pattern. An example of a snippet categorized as having a correct pattern is in Figure 15, while an example of a snippet that was not classified as having a pattern, but did not contain a smell is in Figure 16. The observed pattern was a derivative of the guard pattern (Jones, 1997). It is expressed as an Android recipe, NullPointer Guard, using the Portland Form (Ward, n.d.) in Figure 17.

```
70:   /**
71:    * Called when the activity is first created.
72:    */
73:   @Override
74:   public void onCreate(Bundle savedInstanceState) {
75:         try {
76:               super.onCreate(savedInstanceState);
77:               mDbHelper = new CatalogueDBAdapter(this);
78:               mDbHelper.open();
79:               setContentView(R.layout.administration_functions);
80:               Bundle extras = getIntent().getExtras();
81:               if (extras != null && extras.containsKey(DOAUTO)) {
82:                     try {
83:                           if (extras.getString(DOAUTO).equals("export"))
{
84:                                 finish_after = true;
85:                                 mExportOnStartup = true;
86:                           } else {
87:                                 throw new RuntimeException("Unsupported
DOAUTO option");
88:                           }
89:                     } catch (NullPointerException e) {
```

Figure 15. An example of the NullPointerException corrective pattern. The signature null test and containsKey sentinel is apparent.

```
83:
84:         // Set up the Action Bar
85:         ActionBar actionBar = getSupportActionBar();
86:         actionBar.setHomeButtonEnabled(true);
87:         actionBar.setDisplayHomeAsUpEnabled(true);
88:
89:         mListView = (ListView) getListView();
90:
91:         mDB = new TramHunterDB();
92:
93:         Bundle extras = getIntent().getExtras();
94:         if (extras != null) {
95:               destinationId = extras.getLong("destinationId");
96:               searchQuery = extras.getString("search_query");
97:         }
98:
99:         // Are we looking for stops for a route, or fav stops?
100:        if (searchQuery != null) {
101:              final CharSequence title =
getString(R.string.search_results, searchQuery);
102:              actionBar.setTitle(title);
```

Figure 16. An example of a snippet that does not contain a NullPointerException smell or corrective pattern. In this case, the signature key check is missing and defaults are not used for the primitive accessor.

```
NullPointer Guards in Recipes
Author: Brad Dennis, dennibc@auburn.edu
This material free of Copyright or other restrictions.

This recipe is derived from my dissertation research on repatterning
Android applications to improve reliability.

-------------------------------------------------------------------------
In this one and only section I offer one recipe for protecting against the
NullPointerException often observed in Android Activities.
      1. NullPointer Guard
-------------------------------------------------------------------------

1. NullPointer Guard
Inbound intents, both intra- and extra-application maybe contain erroneous
or incomplete bundle data.

Therefore:  Bundles and their packaged data should be explicitly affirmed
as existing prior to accessing the bundle or retrieving the data.

When a non-existent bundle, or key is accessed on a bundle, the Android
Framework responds with a NullPointerException.  This recipe guards against
that event.
```

Figure 17. The NullPointer Guard Android recipe.

The following results were collected using the *NullPointerException* corrective pattern identification scheme:

Table 8. NullPointerException Corrective Pattern Study Metrics

Metric	Value
Number of Markers	782
Number of Patterns	75
Number of Apps with Patterns	38

Table 9. NullPointerException Corrective Pattern Study Results

Frequency of Occurrence	Result
Number of Patterns	75
Patterns to Markers Ratio	10%
Number of Developers	38

OutofMemoryError Corrective Pattern Study

A snippet was considered containing an *OutOfMemoryError* corrective pattern if static inner classes were used for *Threads*, *AsyncTasks*, *Handlers* or *Runnables*, and they contained a *WeakReference* to the parent activity if necessary. An example of the *OutOfMemoryError* corrective pattern is in Figure 18. There are no examples of snippets that did not contain a smell or a corrective pattern. The observed pattern was an application of the singleton pattern (Gamma, Helm, Johnson, & Vlissides, 1995). It is expressed as an Android recipe, Avoid Activity Leaks, using the Portland Form in Figure 19.

```
119:
120:     /** ImageFetcher for (large) album cover art. */
121:     private ImageFetcher mImageFetcher;
122:
123:     /** ImageCache parameters for the album art. */
124:     private ImageCacheParams mImageCacheParams;
125:
126:     /** Is this the first time the app has run? */
127:     private boolean mFirstRun = false;
128:
129:     private final Handler uiThreadHandler = new UiThreadHandler(this);
130:
131:     private final static class UiThreadHandler extends Handler {
132:         WeakReference<SqueezerActivity> mActivity;
133:
134:         UiThreadHandler(SqueezerActivity activity) {
135:             mActivity = new WeakReference<SqueezerActivity>(activity);
136:         }
137:
138:
```

Figure 18. An example OutOfMemoryError corrective pattern. The handler is static and a WeakReference is used.

```
Avoid Activity Leaks in Recipes
Author: Brad Dennis, dennibc@auburn.edu
This material free of Copyright or other restrictions.

This recipe is derived from my dissertation research on repatterning
Android applications to improve reliability.

------------------------------------------------------------------------
In this one and only section I offer one recipe for avoiding Activity leaks
often observed in Android applications.
      2. Avoid Activity Leaks
------------------------------------------------------------------------

1. Avoid Activity Leaks
Handlers, Runnables, AsyncTasks, and Threads do not share the same
lifecycle as their parent activity.  In Java, anonymous classes and non-
static inner classes hold a reference to their outer class.

Therefore:  When Handlers, Runnables, AsyncTasks, and Threads are used as
inner classes, they should be implemented statically with a WeakReference
to the parent activity to avoid leaking it.

Static inner classes do not keep an implicit reference to their outer class
so the mismatched life-cycle of these objects and their parents will not
cause an inadvertent leak.
```

Figure 19. The Avoid Activity Leaks Android recipe.

The following results were collected using the *OutOfMemoryError* corrective pattern identification scheme:

Table 10. OutOfMemoryError Corrective Pattern Study Metrics

Metric	Value
Number of Markers	2020
Number of Patterns	4
Number of Apps with Patterns	3

Table 11. OutOfMemoryError Corrective Pattern Study Results

Frequency of Occurrence	Result
Number of Patterns	4
Patterns to Markers Ratio	0%
Number of Developers	3

BadTokenException Corrective Pattern Study

A snippet was considered containing a *BadTokenException* corrective pattern if the state of the activity was checked prior to showing the dialog and the context for the dialog was the parent activity. If the state was checked but the context was unable to be evaluated in the snippet, the original file was inspected. An example of the *BadTokenException* corrective pattern is in Figure 20. There are no examples of snippets that did not contain a smell and also not a pattern. The observed pattern was an application of the guard pattern. It is expressed as two Android recipes, The Orphaned Dialog and Dialog Context, using the Portland Form in Figure 21.

```
178:
179:        // If we are in the UI thread, update the progress.
180:        if (Thread.currentThread().equals(mUiThread)) {
181:              // There is a small chance that this message could be set
to display *after* the activity is finished,
182:              // so we check and we also trap, log and ignore errors.
183:              // See
http://code.google.com/p/android/issues/detail?id=3953
184:              if (!isFinishing()) {
185:                    try {
186:                          mProgress.setMessage(message);
187:                          if (!mProgress.isShowing())
188:                                mProgress.show();

189:                    } catch (Exception e) {
190:                          Logger.logError(e);
191:                    }
192:              }
193:        } else {
194:              // If we are NOT in the UI thread, queue it to the UI
thread.
195:              mHandler.post(new Runnable() {
196:                    @Override
197:                    public void run() {
```

Figure 20. An example of the BadTokenException corrective pattern. This class was inspected and the appropriate context was also used by the developer.

```
Android BadTokenExceptions in Recipes
Author: Brad Dennis, dennibc@auburn.edu
This material free of Copyright or other restrictions.

These recipes are derived from my dissertation research on repatterning
Android applications to improve reliability.

-------------------------------------------------------------------------
In this one and only section I offer two recipes for avoiding
BadTokenExceptions often observed in Android applications.
      3. The Orphaned Dialog
      4. Dialog Context
-------------------------------------------------------------------------

1. The Orphaned Dialog
Due to the nature of the Android Framework, code that executes dialogs may
called even though the parent Activity is in the finishing state.

Therefore:  The state of the activity should be evaluated before showing
dialogs.  This can be done with an isFinishing() call.

2. Dialog Context
In Android, dialogs require a context for resources.  This dialog should
share the same context as its parent activity.

Therefore:  The context for a dialog should always be the parent activity
itself and never the application context.
```

Figure 21. The Orphaned Dialog and Dialog Context Android recipes.

The following results were collected using the *BadTokenException* corrective pattern identification scheme:

Table 12. BadTokenException Corrective Pattern Study Metrics

Metric	Value
Number of Markers	3150
Number of Patterns	4
Number of Apps with Patterns	3

Table 13. BadTokenException Corrective Pattern Study Results

Frequency of Occurrence	Result
Number of Patterns	4
Patterns to Markers Ratio	0%
Number of Developers	3

4.7 Enhancement Pattern Study Methodology

Goals

The goal of this survey was to determine what behavior strategies, or enhancement patterns, were being used by FOSS Android developers when a failure condition from *NullPointerException*, *OutofMemoryError*, and *BadTokenException* was detected in their applications. An exploratory examination was chosen as the research design, as it provides a flexible research model and is well suited for informal pattern discovery. The Enhancement Pattern Study was done in conjunction with the Smell Study and the Corrective Pattern Study.

Analysis procedure

Three *frequency of occurrence* metrics were calculated. The first frequency of occurrence measure was the total number of enhancement patterns identified across the body of applications. This indicates the prevalence of the pattern in use. The second frequency of occurrence was the ratio of enhancement patterns present in code segments with respect to the total number of potential use segments. This value is an indicator of adoption. The third frequency of occurrence measure was the number of developers observed using the pattern. Individual developers were identified by package naming schemes.

4.8 Enhancement Pattern Study Results

NullPointerException Enhancement Pattern Study

Neither initial investigation nor the subsequent studies led to any discoveries of enhancement patterns. The most common behavior observed was to do nothing and rely on defaulted data. An example of this do-nothing strategy can be seen in Figure 22.

```
782:        // Call the super method only after we have the searchManager
set up
783:        super.onRestoreInstanceState(inState);
784:  }
785:
786:  @Override
787:  protected void onSaveInstanceState(Bundle inState) {
788:        super.onSaveInstanceState(inState);
789:
790:        // Saving intent data is a kludge due to an apparent Android
bug in some
791:        // handsets. Search for "BUG NOTE 1" in this source file for a
discussion
792:        Bundle b = getIntent().getExtras();
793:        if (b != null) {
794:              if (b.containsKey("isbn"))
795:                    inState.putString("isbn", b.getString("isbn"));
796:              if (b.containsKey(BY))
797:                    inState.putString(BY, b.getString(BY));
798:        }
799:
800:        inState.putParcelable("LastBookIntent", mLastBookIntent);
801:        // Save the current search details as this may be called as a
result of a rotate during an alert dialog.
```

Figure 22. An example of the do-nothing strategy when a NullPointerException is guarded against. If the data is non-existent, no action is taken by the developer.

Table 14. NullPointerException Corrective Pattern Study Metrics

Metric	Value
Number of Markers	782
Number of Patterns	0
Number of Apps with Patterns	0

Table 15. NullPointerException Corrective Pattern Study Results

Frequency of Occurrence	Result
Number of Patterns	0
Patterns to Markers Ratio	0%
Number of Developers	0

OutofMemoryError Enhancement Pattern Study

The initial investigation and subsequent studies did not lead to the discovery of any enhancement patterns. There was a noticeable difference in implementation between one

developer and the remaining two that had used the corrective pattern. This developer had tested and shown that the weak reference to the activity was not null before accessing it. This protected against a *NullPointerException* in the case of the activity being garbage collected. However, when that condition was detected, a do-nothing strategy was employed. This can be seen in Figure 23.

```
    private static class HandlerResetCoordinates extends
WeakReferenceHandler<CacheDetailActivity> {
        private boolean remoteFinished = false;
        private boolean localFinished = false;
        private final ProgressDialog progressDialog;
        private final boolean resetRemote;

        protected HandlerResetCoordinates(CacheDetailActivity activity,
ProgressDialog progressDialog, boolean resetRemote) {
            super(activity);
            this.progressDialog = progressDialog;
            this.resetRemote = resetRemote;
        }

        @Override
        public void handleMessage(Message msg) {
            if (msg.what == ResetCoordsThread.LOCAL) {
                localFinished = true;
            } else {
                remoteFinished = true;
            }

            if (localFinished && (remoteFinished || !resetRemote)) {
                progressDialog.dismiss();
                final CacheDetailActivity activity = getActivity();
                if (activity != null) {
                    activity.notifyDataSetChanged();
                }
            }
        }

    }
```

Figure 23. An example of the do-nothing strategy if the parent activity has been garbage collected.

Table 16. OutOfMemoryError Corrective Pattern Study Metrics

Metric	Value
Number of Markers	782
Number of Patterns	0
Number of Apps with Patterns	0

Table 17. OutOfMemoryError Corrective Pattern Study Results

Frequency of Occurrence	Result
Number of Patterns	0
Patterns to Markers Ratio	0%
Number of Developers	0

BadTokenException Enhancement Pattern Study

No enhancement patterns for the *BadTokenException* failure condition were discovered during the initial investigation or the subsequent studies. A typical behavior was to suppress the dialog and then do nothing. This can be seen in Figure 24.

```
555:                            finish();
556:              }});
557:
558:         if (!isFinishing()) {
559:              try {
560:                   //
561:                   // Catch errors resulting from 'back' being pressed
multiple times so that the activity is destroyed
562:                   // before the dialog can be shown.
563:                   // See
http://code.google.com/p/android/issues/detail?id=3953
564:                   //
565:                   alertDialog.show();
566:              } catch (Exception e) {
567:                   Logger.logError(e);
568:              }
569:         }
570:   }
571:
572:   /**
573:    * Update all (non-existent) thumbnails
574:    *
```

Figure 24. An example of the do-nothing strategy when a BadTokenException is encountered.

Table 18. BadTokenException Corrective Pattern Study Metrics

Metric	Value
Number of Markers	3150
Number of Patterns	0
Number of Apps with Patterns	0

Table 19. BadTokenException Corrective Pattern Study Results

Frequency of Occurrence	Result
Number of Patterns	0
Patterns to Markers Ratio	0%
Number of Developers	0

4.9 Reliability Experiment Methodology

Goals, context, hypotheses, and variables

Goals

The objective of this experiment was to investigate the effect on reliability of the repatterning process applied to Android applications that have discernible smells. A controlled empirical experiment was chosen as we had access to a pool of open source applications and were able to manipulate their designs deliberately and systematically.

Context

The context of the experiment is consumer-facing, open-source Android applications of unknown quality, written by developers with unknown levels of experience.

Hypotheses

The informal hypothesis can be stated as: repatterned consumer Android applications will be more reliable after repatterning than before. The formal hypothesis and null hypothesis can be stated as:

> H_{1a}– Free Open Source Software (FOSS) Android applications containing *NullPointerException* smells have a higher defect intensity before they have been repatterned than after they have been repatterned.

H_{0a} – Untreated FOSS Android applications containing *NullPointerException* smells have equivalent or lower defect intensity than their repatterned versions.

H_{1b}– FOSS Android applications containing *OutofMemoryError* smells have a higher defect intensity before they have been repatterned than after they have been repatterned.

H_{0b} – Untreated FOSS Android applications containing *OutofMemoryError* smells have equivalent or lower defect intensity than their repatterned versions.

H_{1c}– FOSS Android applications containing *BadTokenException* smells have a higher defect intensity before they have been repatterned than after they have been repatterned.

H_{0c} – Untreated FOSS Android applications containing *BadTokenException* smells have equivalent or lower defect intensity than their repatterned versions.

These defect intensity was calculated for each application. Defect intensity is defined as the number of observable defects per one thousand user or system events.

Variables

The independent variable was the repatterning process applied to the applications. The dependent variable was the number of observable defects per one thousand user or system events.

Assumptions

- Any repatterning being applied does not alter the semantics of the application at the user level of abstraction.
- Application of the repatterning is safe and does not inject new bugs.
- The effort involved in repatterning is aligned with quality improvements.
- Repatterning does not significantly degrade other quality attributes, notably maintainability.

Design

Randomization

FOSS Android applications were prequalified from a catalog of 749 FOSS Android complex applications compiled from three online sources (F-Droid, AndroidFreeSoftware, AOpenSource.com). A prequalified application was defined as one with ready access to the source code, support for Android version 2.3.3, is compatible with the Intel Atom (x86) emulator, containing each of the three smells identified in the Smell Study, and is not a live wallpaper, widget, or plug-in to any other application. Apps were selected at random from the set of applications identified by the Smell study and were built from the available source. The first fifteen apps that could be built successfully within a thirty minute timeframe were selected as candidates. There was a variety of co-factors for this experiment: the experience of the developer, the maturity of the application, the complexity of the design, the maturity of the OS and third party libraries used, and the utilization of external resources.

Grouping

Applications selected for repatterning were replicated and placed into a control group and three treatment groups. The treatment group was further replicated and categorized into three blocks based upon the treatment to be applied. This type of grouping allowed us to control any variations in reliability improvements between the treatments.

Design Type

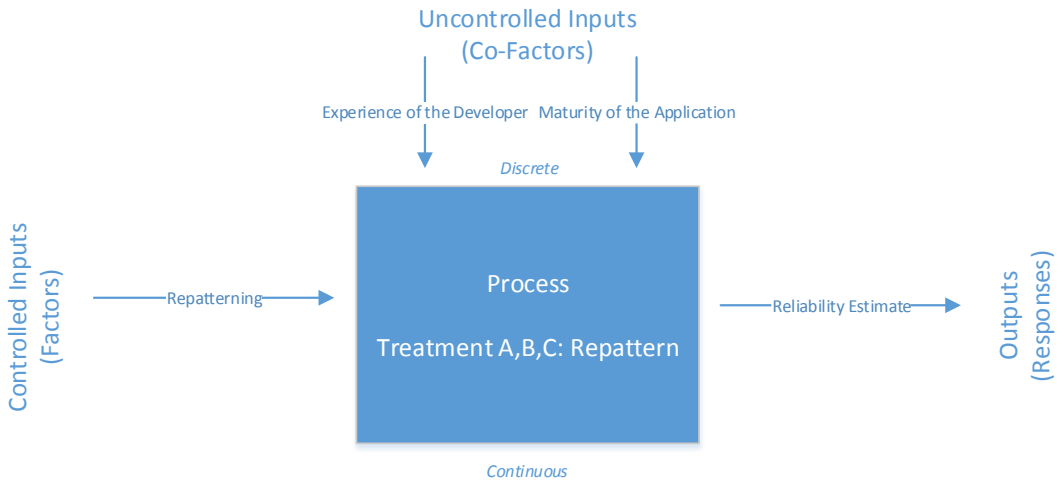

Figure 25. Experiment Design

The definition, hypotheses, and measures of the reliability evaluation established the experiment as a single factor controlled experiment. The semantic conformance, utility of the process, effect on other quality attributes, and safety evaluations of the repatterning technique were observed but not investigated.

Subjects

The subjects of this experiment were complex FOSS Android applications chosen at random from three public repositories: (F-Droid, n.d.), (AndroidFreeSoftware , n.d.), and (AOpenSource.com, n.d.).

Objects

The objects used in this experiment were three repatternings representative of real world Android application failures; *NullPointerException*, *OutofMemoryError*, and *BadTokenException*. These related specifically to application life cycle faults and memory

exhaustion. The regular expressions developed for the code smells from the Smell Study were expanded and used to identify a more comprehensive set of locations to evaluate as a smell and then to apply the patterns from the Corrective Pattern Study. The expanded markers are listed in Table 20, while the Corrective Patterns used are listed in Figures 19, 21, 24. A do-nothing strategy was used in lieu of an enhancement pattern as these were not discovered in the Enhancement Pattern Study.

Table 20. Expanded regular expressions for identifying smells.

Treatment	**Regular Expression**
NullPointerException	(getIntent|get\w*Extra|getAction)
OutOfMemoryError	(new|extends|implements) \w*(Thread|AsyncTask|Runnable|Handler)
BadTokenException	(\.show\(|\s*showDialog\(|onCreateDialog)

Instrumentation

A common test harness was developed and used for exposing faults of interest in the candidate applications. The test harness was developed in Python 2.7.6 using the interpreter available from Python Software Foundation (Python Software Foundation, 2013). Python wrapper classes were built to provide library access to the Android SDK Platform-tools Revision 22.2.1 (Android, 2013). The test harness was a client-server architecture where the server maintained a single jobs queue and reporting service for a test client to request a job and report results, see Figure 26. The tests run on two Nexus 5 (32GB and 16GB) phones running Android 4.4.2 (KitKat).

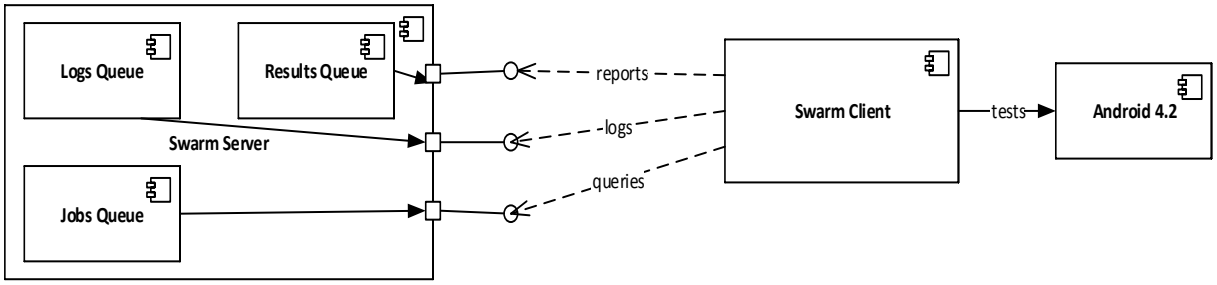

Figure 26. Test Harness Architecture

```
initialize the device
initialize ADB logcat and configure monitoring
install and start application under test

for each test configuration:
     for each test seed:
          keypress 'TAB'
          keypress 'ENTER'
          keypress 'ENTER'

          start monkey with configuration and seed

          sleep 1 second
          send fuzzed intent

          recover application if necessary

          while monkeying:
               sleep 1/10th of a second

if app is running:
     stop app
     remove any app data
     uninstall app
```

Figure 27. Reliability Test Algorithm

The test harness was comprised of three aspects; a test server, a test client, and the test algorithm itself. The primary responsibility of the server was to queue the test jobs, aggregate test results, and provide a common logging reporting mechanism for the clients. The responsibility of the client was to execute the reliability test, collect data, and report the test results to the server. The client monitored both the *logcat* and *monkey* streams for exceptions,

crashes, and test summaries. The events tracked are listed in Table 21. The test algorithm, (Figure 27) was a series of 1000 monkey tests executed in 100 event steps. This facilitated the monitoring and recovery of the application in the event of a failure with minimal loss to the 100,000 event target. The test environment had three characteristics (Table 22) to increase the fault potential: low-memory environment to encourage *OutOfMemoryErrors*, the Monkey configuration was skewed to encourage *BadTokenException*, and fuzzed intents were randomly sent to activities to encourage *NullPointerExceptions*.

Table 21. Stream Monitoring Expressions

Event	**Stream**	**Match Expression**
Monkey output	Monkey	`Not 'activityResuming'`
Monkey crashes	Monkey	`'// CRASH:'`
Device crashes	Monkey	`'pool-1-thread-1" prio=5 tid=15 WAIT'`
Short error messages	Monkey	`'// Short Msg: (.*)'`
Monkey test aborts	Monkey	`'** Monkey aborted due to error.'`
Test elapsed time	Monkey	`'## Network stats: elapsed time=(.*?)ms \('`
Test events injected	Monkey	`Events injected: (.*)`
Logcat output	Logcat	*none*
Logcat Exceptions	Logcat	`'FATAL EXCEPTION:'`

Table 22. Test Environment Configurations

Characteristic	**Configuration**
Low-memory	`dalvik.vm.heapsize =64m`

Monkey configuration	`--pct-touch 15`
	`--pct-motion 10`
	`--pct-trackball 15`
	`--pct-syskeys 25`
	`--pct-nav 2`
	`--pct-majornav 15`
	`--pct-appswitch 15`
	`--pct-flip 1`
	`--pct-anyevent 2`

Fuzzed intent type A	`adb shell am start <activity>`
Fuzzed intent type B	`adb shell am start <activity> --esn testkey`

Data collection procedure

The test harness created two reports for each application tested. The first report was the summary containing a timestamp, the device ID, the testing strategy, the APK name, the number of errors, monkey aborts, app crashes, device restarts, time elapsed, and the number of events injected. The second report contained a timestamp, the APK name, and a list of exceptions seen and each exceptions frequency.

Process

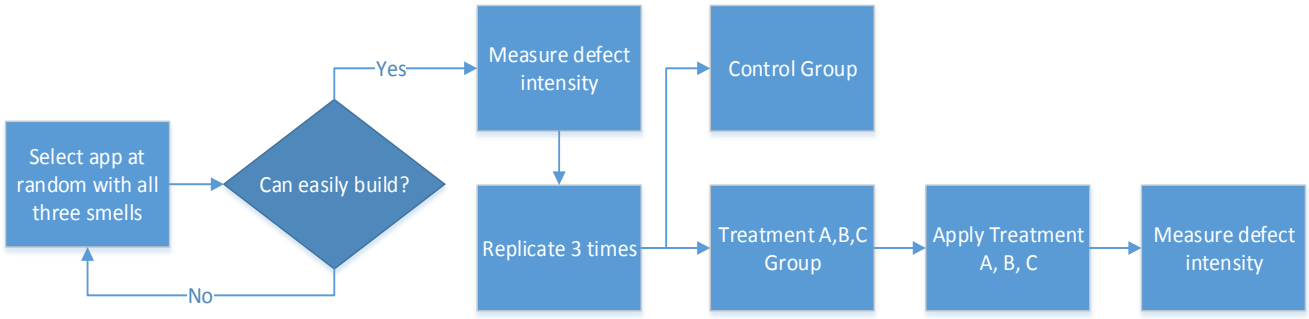

Figure 28. Experimental Process

The experimental process contained three distinct phases. The first phase was the random selection, build, and replication process. Apps were randomly selected from a list containing all three smells. An attempt was made to build the app from the collected source code. If the app could be built within thirty minutes, it was then placed into a working directory and replicated three times. Each source directory was appended with a suffix to reflect its group assignment, "_pre", "_npe", "_oom", "_bte". Once fifteen candidates were identified, the next phase was initiated. This was the treatment, or repatterning phase. The expanded markers from Table 20 were used to identify potential locations for pattern application. These locations were individually inspected and the corrective pattern was applied if they were determined to contain the smell associated with the treatment. The same criteria were used for the smell classification as discovered in the Smell Study.

4.10 Reliability Experiment Results

Results

The results of the reliability test for the Control group are shown in Tables 23 and 24.

Table 23. Control group results.

APK	Errors	Events	Defect Intensity (d/kE)
apps.babycaretimer	64	98660	0.649
com.android.keepass	43	98292	0.437
com.app2go.sudokufree	60	98324	0.610
com.beem.project.beem	76	97153	0.782
com.commonsware.android.arXiv	230	90955	2.529
com.drismo	42	98833	0.425
com.nexes.manager	219	94036	2.329
com.owncloud.android	8	99918	0.080
com.replica.replicaisland	132	95032	1.389

com.zapta.apps.maniana	11	99730	0.110
info.staticfree.android.units	14	99737	0.140
org.sixgun.ponyexpress	80	98430	0.813
org.tomdroid	14	99501	0.141
org.vono.narau	154	96065	1.603
urbanstew.RehearsalAssistant	378	82545	4.579

Table 24. Errors observed during the Control group reliability test.

Error	Number of Observations
NullPointerException	1040
OutOfMemoryError	79
BadTokenException	0
Device Crashes	14
All other exceptions	392
Total	**1525**

The results of the reliability test for the *NullPointerException* group are shown in Tables 25 and 26.

Table 25. NullPointerException group results.

APK	Errors	Events	Defect Intensity (d/kE)
apps.babycaretimer	1	99921	0.010
com.android.keepass	12	99422	0.121
com.app2go.sudokufree	36	97784	0.368
com.beem.project.beem	65	97381	0.667
com.commonsware.android.arXiv	16	98745	0.162
com.drismo	31	99020	0.313
com.nexes.manager	60	97953	0.613
com.owncloud.android	1	99902	0.010
com.replica.replicaisland	129	96043	1.343
com.zapta.apps.maniana	0	100000	0.000

info.staticfree.android.units	1	99902	0.010
org.sixgun.ponyexpress	10	99747	0.100
org.tomdroid	13	99533	0.131
org.vono.narau	45	98797	0.455
urbanstew.RehearsalAssistant	314	83001	3.783

Table 26. Errors observed during the NullPointerException group reliability test.

Error	Number of Observations	Percent Difference
NullPointerException	193	-81%
OutOfMemoryError	99	25%
BadTokenException	0	0%
Device Crashes	50	257%
All other exceptions	404	3%
Total	**746**	**-51%**

The results of the reliability test for the *OutOfMemoryError* group are shown in Tables 27 and 28.

Table 27. OutOfMemoryError group results.

APK	Errors	Events	Defect Intensity (d/kE)
apps.babycaretimer	65	98740	0.658
com.android.keepass	38	98363	0.386
com.app2go.sudokufree	61	97752	0.624
com.beem.project.beem	51	97546	0.523
com.commonsware.android.arXiv	672	69678	9.644
com.drismo	26	99026	0.263
com.nexes.manager	178	95699	1.860
com.owncloud.android	4	99602	0.040
com.replica.replicaisland	134	94187	1.423
com.zapta.apps.maniana	5	99754	0.050
info.staticfree.android.units	1	100000	0.010

org.sixgun.ponyexpress	45	98466	0.457
org.tomdroid	15	99507	0.151
org.vono.narau	41	98571	0.416
urbanstew.RehearsalAssistant	386	78905	4.892

Table 28. Errors observed during the OutOfMemoryError group reliability test.

Error	Number of Observations	Percent Difference
NullPointerException	1197	15%
OutOfMemoryError	91	15%
BadTokenException	0	0%
Device Crashes	28	100%
All other exceptions	406	4%
Total	**1722**	**13%**

The results of the reliability test for the *BadTokenException* group are shown in Tables 29 and 30.

Table 29. BadTokenException group results.

APK	Errors	Events	Defect Intensity (d/kE)
apps.babycaretimer	65	98740	0.658
com.android.keepass	38	98363	0.386
com.app2go.sudokufree	61	97752	0.624
com.beem.project.beem	51	97546	0.523
com.commonsware.android.arXiv	672	69678	9.644
com.drismo	26	99026	0.263
com.nexes.manager	178	95699	1.860
com.owncloud.android	4	99602	0.040
com.replica.replicaisland	134	94187	1.423
com.zapta.apps.maniana	5	99754	0.050
info.staticfree.android.units	1	100000	0.010
org.sixgun.ponyexpress	45	98466	0.457

org.tomdroid	15	99507	0.151
org.vono.narau	41	98571	0.416
urbanstew.RehearsalAssistant	386	78905	4.892

Table 30. Errors observed during the BadTokenException group reliability test.

Error	**Number of Observations**	**Percent Difference**
NullPointerException	700	-33%
OutOfMemoryError	81	3%
BadTokenException	0	0%
Device Crashes	38	171%
All other exceptions	336	-14%
Total	**1155**	**-24%**

Analysis

A *Wilcoxon Signed-Rank Test (one-tailed)* is the most appropriate statistical test for experiments where subjects are tested before and after a treatment is applied, and the observed data is assumed not to be normal. This allows each subject to be its own control and provides a valid test of the influence of the treatment applied.

- A Wilcoxon test using the calculator available at (Stangroom, 2014) showed that the number of defects observed after the *NullPointerException* treatment was significantly reduced by the treatment. The Z-value was -3.4078. The p-value is 0.00032. The result is significant at $p \leqslant 0.01$.
- A Wilcoxon test showed that the number of defects observed after the *OutOfMemoryError* treatment was not significantly affected by the treatment. The

Z-value is -1.3631. The p-value is 0.08692. The result is not significant at p⩽ 0.01.

- A Wilcoxon test showed that the number of defects observed after the *BadTokenException* treatment was significantly reduced by the treatment. The Z-value is -3.351. The p-value is 0.0004. The result is significant at p⩽ 0.01.

These results are summarized in Table 31.

Table 31. The statistical significance of observed results.

Treatment	**Percent Difference of Observed Exceptions**	**Z-value**	**p-value**
NullPointerException	-51%	-3.4078	0.00032
OutOfMemoryError	13%	-1.3631	0.08692
BadTokenException	-24%	-3.351	0.0004

Chapter 5 – Summary, Conclusions & Future Work

5.1 Summary

The purpose of this study was to determine if the usefulness of refactoring could be expanded to affect a broader set of non-functional requirements than the set limited by restructuring activities. A real world problem domain was selected, Android application reliability, and the refactoring model was explored in this context. To support this exploration, the refactoring model was identified, generalized, and reimagined as a reengineering activity. Once the domain was selected and the model was reimagined the research was able to proceed.

An investigation into Android reliability revealed that applications were experiencing systemic failures, and developers were being given too little, incorrect, or no guidance at all to address them. The two most dominant failures, *NullPointerException* and *OutofMemoryError*, and a third common failure, *BadTokenException,* were selected for the study. These three signal exceptions were the main organizing principle for the rest of the research.

The refactoring process was discovered to be a three-step, holistic process combining a problem heuristic, a code smell, a behavior-preserving transformation, a refactoring, and verification of the transformation. These three basic steps were able to be preserved when adapting this model from restructuring to reengineering. However, since a key characteristic of restructuring is that no observable changes are made during the activity, behavior-preserving

transformations had to be rejected from the new model. The reengineering analogs to this transformation are behavior-correcting transformations and behavior-enhancing transformations, referred to collectively as repatterning. Further, these two transformations led to an expansion of the refactoring third step, which is to verify a transformation was applied correctly. While verification of the application of the transformation is still necessary when reengineering, behavior-correcting transformations should also be verified that the fault being treated was cured. In addition, behavior-enhancing transformations should be validated to confirm that they do not alter the stated intent of the system.

Free Open Source Software Android applications were selected from three marketplaces and used as the population to investigate how repatterning could improve Android reliability. Over 900 applications were initially cataloged from the marketplaces. This information was filtered to 749 by removing duplicates and different versioned entries for the same application. The applications were then evaluated to determine if they were complex, and if the APK and source were available for download. This evaluation trimmed the 749 candidates to 471 applications. These 471 applications were then installed onto an emulator and launched to verify they were compatible, and to manually verify their complexity. This process yielded 323 complex, runnable applications that had the APK and source code.

The 323 applications were initially tested using a Monkey-based harness to collect metrics on reliability and the frequency of failures related to the exceptions of interest. A *NullPointerException* was observed in 77 applications, accounting for 37% of all the exceptions. An *OutofMemoryError* was seen in 23 applications for 18% of the total exceptions observed, while a *BadTokenException* was observed in five applications with 1% of all exceptions. These percentages loosely correspond to the data from two real-world analyses but diverges from the

observations from the IPC security investigation which observed 37% *NullPointerException*, 27% *BadTokenException*, and did not report any *OutofMemoryErrors*.

The results from the pre-test were used for the investigation into reliability smells, corrective patterns, and enhancement patterns. Specific markers in the source code were identified for detailed inspection corresponding to each of three signal exceptions. These markers were then used to extract 20 line code snippets from the pool of applications which were visually inspected and cataloged as containing a smell, a corrective pattern, an enhancement pattern, or nothing at all. 429 *NullPointerException* smells out of 782 markers were identified in 121 applications for a 55% smell to marker ratio and 37% of the applications at-risk. 75 *NullPointerException* corrective patterns were observed in use by 38 different developers, with no distinguishing enhancement pattern identified. There were 2009 *OutOfMemoryError* smells out of 2020 markers that were identified in 213 applications for a 99% smell to marker ratio and 66% of the applications at-risk. Four *OutOfMemoryError* corrective patterns were observed in use by three different developers with no distinguishing enhancement pattern identified. There were 2664 *BadTokenException* smells out of 3150 markers that were identified in 258 applications for an 85% smell to marker ratio and 80% of the applications at-risk. Four *BadTokenException* corrective patterns were observed in use by three different developers with no distinguishing enhancement pattern identified.

The Reliability Experiment was able to be conducted once the exploratory studies into smells and patterns were complete. Fourty-one applications were found to have contained smells for all three signal exceptions with no corrective or enhancement pattern implementations. An application was chosen from this pool at random and an attempt was made to build the application from source. The first 15 applications that could be built were candidates for the

experiment. Four sets of the source code were made, one for the control and one for each of the treatments. A treatment consisted of applying corrective patterns and a "do-nothing" strategy for enhancements to every location of an identified smell for a specific signal exception. The treatments were applied individually to each group and then their reliability was measured and compared against the applications built from the control group. The *NullPointerException* treatment was found to reduce observed exceptions by 51% in a statistically significant manner ($p \leqslant 0.01$). The *OutofMemoryError* treatment was found to increase the exceptions observed by 13%, however these results were not statistically significant ($p \leqslant 0.01$). The *BadTokenException* treatment reduced observed errors by 24%, and this was statistically significant ($p \leqslant 0.01$).

5.2 Conclusions

Five research questions were posed to guide this research; what issues do Android applications suffer from? Next, what is the impact of these issues? Third, what reliability guidance is available for developers? Fourth, what tools do developers have access to that can verify their applications? Finally, how can repatterning be used to improve these reliability problems?

In response to research question 1, a pattern of failures was found across two real-world analyses as well as a security focused analysis. The lifecycle related nature of the errors together with the insights from (Franke, Elsemann, Kowalewski, & Weise, 2011) suggest that developers are struggling with the complexities of the high-level, multi-entry, component architecture of the Android framework. Each of the building blocks of an Android application, activities, services, broadcast receivers, and content providers, are effectively an application in and of themselves.

This aspect is fundamental to Android development and is readily available (Google o, n.d.), yet still an apparent prime contributor to many of the reliability issues facing developers. One conclusion seems possible. We may not understand how developers are acquiring programming expertise in this modern information age. We may need to identify new strategies for transferring knowledge that are congruent with how developers are seeking to acquire new knowledge. One curiosity that was found during the exploratory studies was the use of identical code snippets by different developers. A cursory search into their source found them to be examples from developer blogs or popular developer sites such as Stack Overflow. This suggests that developers are not seeking an understanding of principles or general concepts as often presented on the official Android Developer's site, but instead are merely looking for concrete solutions to their immediate problems. Techniques such as refactoring and repatterning appear well suited to overcome this dilemma, as they re-encode abstract principles and concepts into concrete practices. For example, the Long Method smell and its related refactoring, ExtractMethod, is one way to repackage the principle of cohesion as a practice. The smells and guard patterns discovered in this research are another example of this as they synthesize the lifecycle complexities of Android into discrete, light-weight practices without the depth and broader discussion of architectures or lifecycles.

In response to research question 2, these systemic reliability issues were found to incur a real cost for developers. A 5% customer loss due to application failures (Bugsense, n.d.), with the daily revenue of the top 200 applications in Google Play being estimated at $1.1M (Agten, 2013) amounts to at least a $50,000 per day loss in opportunity for Android developers. The economic impact of these failures suggests research such as the work presented in this dissertation may make a positive improvement to developer's bottom lines. While this research

makes direct contributions to the identification of potential problems, and the guarding against faults in Android, no recommendations were able to be made with respect to the features a developer could employ when encountering these faults. It may be the case that these recommendations were not possible to derive as these enhancements would need to maintain *semantic conformance* (Meyer H. , 2008) with applications themselves and therefore difficult to generalize. However, in the studied applications there were not many enhancements observed, and the dominant strategy encountered was "do-nothing". This suggests the developers did not have the depth of knowledge or maturity to articulate and implement an error handling strategy for their application. This supports the need for more practical research in this domain.

In response to research question 3, not surprisingly, Google provides the most comprehensive guidance for Android developers, while other sources come up significantly short. However, if we consider the types of failures occurring and amount and types of information presented, we can conclude that there are still significant gaps in guidance. These gaps might be in the way existing information is presented, as intimated in the conclusions for research question 1, or missing information altogether. The findings of, and observations during, this research suggest the need to evaluate what information is being presented, and how it is being presented on the Android Developers website. One recommendation is to extend their Best Practices series with a Best Practices for Reliability guide. Another recommendation is to extend this series further with a Best Practices for Error Handling that provides some guidance on error handling strategies. This recommendation is also congruent with the conclusions from research question 2. A third recommendation is for the Android team to evaluate each of the code samples presented across the site and ensure they reflect best practices for all quality aspects, and that they are not just a demonstration of specific concept or a particular quality

concern being discussed. This recommendation incorporates the observations found in research question 1. A final recommendation is for the Android team to proactively engage the developer community, and to contribute guidance or make corrections to Android technical content being published by the community itself. The framework engineers are very active in the Android Developer newsgroup on Google Groups. This level of engagement does not appear to extend to other avenues. An engagement model could be adopted similar to how other large technical companies, such as Microsoft and Blizzard Entertainment, engage their users with active and dedicated community managers.

In response to research question 4, there is a clear lack of reliability testing tools available for the developer. Many of the tools available are designed to test functional requirements and not to test non-functional characteristics of an application. The recommendations for this problem are available in the Future Work - *Fixing Monkey* discussion.

Research question 5 was decomposed into four subquestions; what code smells exist related to the three signal exceptions? Next, what corrective patterns were being used by developers to guard against the failures? Third, what enhancement patterns were being used by developers when an error condition was encountered? Finally, does the application of these patterns to the locations suggested by the smells improve the reliability of the system?

In response to subquestion 1, code smells were found to exist that could identify potential weaknesses. The most difficult part of this aspect of the research was synthesizing the information and narrowing the scope of potential problems to a single root cause hypothesis. A conclusion was straightforward for the *NullPointerException* and *BadTokenException*, as the available evidence led to a quick conclusion that these issues were related to *Intents* and *Dialogs*. The *NullPointerException* conclusion was in contrast to the explicit statement from (Levy, 2012)

that the most frequent occurrence of this exception was in the *onResume* method. However, it was congruent with their second-most frequent observation, as well as the analysis of (Maji, Arshad, Bagchi, & Rellermeyer, 2012) and the investigator's research into developer's experiences with the exception. However, this conclusion was in direct contrast to the conclusion drawn by (Popadopoulos, 2013) that suggested the issue was related to memory exhaustion. This researcher disagrees with the memory exhaustion hypothesis as it appears to be misattributed to an impossible cause. In C programming a *NullPointerException* may often arise as the result of a failed *memset* or *malloc.* That is not the case in a memory managed environment like the Dalvik runtime.

The *BadTokenException* root cause research led to two conclusions. First, the dialogs were being shown when the activity was in a finishing state. This conclusion was drawn primarily from the investigation into the developer community. Second, dialogs were being shown with the incorrect context. The conclusion was based upon evidence from Levy that cited an error in the Android documentation and was confirmed by developer reports in the community. The documentation error has been corrected since Levy's report. The investigation into the *OutofMemoryError* was the most difficult. Both Levy and Popadopoulos directly attribute this error to poor memory management with bitmaps. However, the initial inspections of the source code, nor the research into developer reports, yielded conclusions or confirmations. The developers appeared to be using these resources appropriately. If the developers were using bitmaps normally then how could they be the cause of the *OutOfMemoryError*? Two clues helped solve this mystery. First, the work from (Franke, Elsemann, Kowalewski, & Weise, 2011) suggested developer confusion on Activity lifecycles, and the observation that the frequent inflation of bitmap resources occurred in the *onCreate* method of an Activity. If the Activity

itself was leaking, then the multiple calls to the *onCreate* method could cause an *OutofMemoryError* when any bitmap resources were inflated. A root cause of the *OutOfMemoryError* was concluded to be Activity leaks.

In response to subquestion 2, corrective patterns were observed being used by developers to guard against failures related to the three signal exceptions. The NullPointer Guard recipe was frequently used and easy to discern. There were two observations of interest. First, some developers had a distinct sentinel signature, *extras != null && extras.containsKey* that was traced back to a tutorial on an Android developer blog (Vogel, 2014). This tutorial has since been updated and the signature removed. This evidence supports some conclusions of research question 1. The second observation was a null guard used by several developers on the *getIntent* method itself. The Android documentation does not state this method returns null. Further, this question was posed to the Android developer newsgroups and an Android framework engineer stated the framework does not return null, but a programmer could set it by calling *setIntent*. It remains a curiosity as to why these developers had a null guard on this method call. The *BadTokenException* Guard recipe was also easy to discern. Either the state of the activity was checked prior to showing a dialog, or it wasn't. Lastly, any use of Avoid Activity Leaks recipe was simple to identify as there were two defining characteristics. A *Thread*, *AsyncTask*, *Handler*, or *Runnable* were implemented as static inner classes and if parent resources were used, a *WeakReference* to the parent activity was used to gain access to it.

In response to subquestion 3, there were no enhancement patterns discovered. These conclusions are discussed in research question 2 and lead to the *The Missing Enhancement Patterns* recommendation for future work.

In response to subquestion 4, qualitative studies show that the key elements of the model, smells and patterns, exist in this domain. Secondly, the empirical evidence from the Reliability Experiment suggests that repatterning can be used to make measurable improvements in application quality. The issues encountered during these studies and the results of these experiments suggest that refactoring should not be applied to reliability without careful consideration of the nature of re-engineering activities. The issues were broader than those typically experienced with refactoring, and as such a new model is needed. In the context of these experiments, repatterning has been demonstrated to have a positive impact on reliability.

Since the population under study consisted of a diverse set of real-world applications from developers of unknown experience, the results should be generalizable to other developers. Further, as there is no Android or reliability specific constraints on repatterning. The model should be generalizable to other domains and other quality attributes that involve observable behavioral changes. *The repatterning process is a reengineering adaptation of refactoring that does make observable improvements in quality.*

5.3 Future Work

The following recommendations are offered for expanding the research on repatterning and into Android reliability issues.

1. *Fidelity to Refactoring* – refactoring as we know it today was the product of research that mechanized an existing restructuring activity that developers were undertaking. In other words, refactoring was not an invention as repatterning is presented in this work. Research into how developers recognize and ameliorate quality issues in practice might

affirm repatterning's fidelity to refactoring, or reveal aspects of the repatterning model that may have been missed by the evaluation of refactoring itself.

2. *Expanding Smell and Pattern Catalogs* – The smells and corrective patterns discovered in this research are contributions to the body of knowledge on Android reliability, but they are far from comprehensive. Research should be conducted to expand these collections.

3. *The Missing Enhancement Patterns* – One of the more curious products of this work was the discovery of how often the "do-nothing" strategy was used by developers in practice. Why are these developers doing nothing when detecting these error states? Exploring this research question might identify some strategies improving the recoverability of Android applications and transferring those strategies to developers.

4. *Fixing Monkey* – The review aspect of this work led to the discovery that some configuration settings on the monkey tool packaged with the Android Developer Tools were undocumented, and the rationalization for the default configurations was not identified. Work should be done to develop configuration profiles for developers to use as models of different user types or patterns of application usage. This would improve monkey's fidelity to the real world, which in turn would improve the tools usefulness to developers. Further, the monkey tool should be extended to have a recovery mode that overcomes its first-fault weakness with respect to reliability measurement.

5. *Smells to Suggest Patterns* – Inexperienced developers might have difficulties identifying when to use a particular pattern or which pattern to use (Gamma, Helm, Johnson, & Vlissides, 1995) (Birukou, Blanzieri, & Giorgini, 2006). The potential of smells as the selection heuristic for patterns should be investigated as an aid for the inexperienced developer. As smells are mapped to potential refactorings (Fowler, 1999) (Kerievsky J. , 2005), new smells may be identified and mapped to patterns. Research into the potential of the smell metaphor to ameliorate the pattern selection problem might prove beneficial in aiding inexperienced developers.

6. *Side Effects of Android Evolution* – There was a notable disconnect between the smells identified and faults encountered for two of the three signal exceptions. For the *BadTokenException* even the pre-test observed exception profile was not congruent with the smaller population selected for the Reliability Experiment. One key difference between the pre-test and the Reliablity Experiment was the move from an emulator running Gingerbread (Android 2.3.3), and a Nexus 5 running KitKat (4.4.2). This difference might also account for the discrepancy between the smells and the observed faults. In other words, how has the speed of Android framework's evolution impacted the reliability of applications? An empirical study could be conducted using a design similar to the experiment used in this work, but the different treatments might be the major releases in the Android revision history. This research might reveal insights that could aid developers in making informed development or support decisions with respect to the Android platform fragmentation.

7. *Replicate the Research* – Reproducibility is the gold standard for any scientific experiment. The exploratory studies would be difficult to reproduce as they are subjective in nature. However, any subsequent investigation should yield similar trends in smell frequency and should identify similar patterns, or lack thereof. Replication of these studies would strengthen the key premises of repatterning. While we intuitively know that applying corrective patterns should result in quality improvements related to the patterns, the results from two of the treatments did not affirm this notion. Replication of the research might uncover the causes for this lack of affirmation and reveal new understandings about Android reliability and repatterning.

References

Python Software Foundation. (2013, Aug). *Python 2.7.6 Release.* Retrieved from https://www.python.org/download/releases/2.7.6/

Agrawal, S., & Wasserman, A. I. (2010). *Mobile Application.* submitted for publication.

Agten, T. (2013). *A Granular App Level Look at Revenues: Google Play vs. Apple App Store.* DISTIMO.

Albuquerque, B. (2011, January 20). *Processing Ordered Broadcasts.* Retrieved July 3, 2013, from Android Developers Blog: http://android-developers.blogspot.com/2011/01/processing-ordered-broadcasts.html

Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., & Memon, A. M. (2012). Using GUI ripping for automated testing of Android applications. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (pp. 258-261). ACM.

Ambler, S. (2007). Survey says... agile has crossed the chasm. *Dr. Dobbs Journal*.

Android. (2013, October). *SDK Tools.* Retrieved from Android Developer: http://developer.android.com/tools/sdk/tools-notes.html

AndroidFreeSoftware . (n.d.). Retrieved from AndroidFreeSoftware: https://wiki.koumbit.net/AndroidFreeSoftware

AOpenSource.com. (n.d.). Retrieved June 9, 2013, from AOpenSource.com: http://www.aopensource.com

AQuA. (2013 a, February). *AQuA Best Practice Guidelines for Producing High Quality Mobile Applications.* Retrieved from App Quality Alliance: http://www.appqualityalliance.org/files/AQuA_best_practices_doc_v2%202_FINAL_5_feb_2013.pdf

AQuA. (2013 b, February). *Android Testing Criteria.* Retrieved from App Quality Alliance: http://www.appqualityalliance.org/files/AQuA_testing_criteria_for_Android_for_v1.4%20final%207_feb_2013.pdf

AQuA. (2013 b, April 23). *Don't risk getting thrown off Google Play.* Retrieved June 4, 2013, from App Quality Alliance: http://www.appqualityalliance.org/blog/?p=59

Arnold, R. (1989). Software Restructuring. *Proceedings of the IEEE 77*, (pp. 607-617).

Asavametha, A. (2012). *Detecting bad smells in spreadsheets.*

Balani, N. (2004, April 08). *Accessing DB2 Everyplace using J2ME devices, part 1*. Retrieved October 15, 2013, from IBM developerWorks: http://www.ibm.com/developerworks/data/tutorials/dm0404balani/section2.html

Birukou, A., Blanzieri, E., & Giorgini, P. (2006). *Choosing the right design pattern: an implicit culture approach.*

Bourquin, F., & Keller, R. K. (2007). High-impact refactoring based on architecture violations. *In Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on* (pp. 149-158). IEEE.

Bugsense. (2012, November 26). *Android vs. iOS users: Who will stop using your app after it crashes?* Retrieved from Visual.ly: http://visual.ly/android-vs-ios-users-who-will-stop-using-your-app-after-it-crashes

Bugsense. (n.d.). *Android Errors in Real Time*. Retrieved June 9, 2013, from Visual.ly: http://visual.ly/android-errors-real-time?view=true

Burd, B. (2011). *Android Application Development All-in-One For Dummies.* For Dummies; 1 edition (December 13, 2011).

Chikofsky, E. J., & Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *Software, IEEE, (7)1*, 13-17.

Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communication in Android. *9th international conference on Mobile systems, applications, and services* (pp. 239-252). ACM.

Coetzee, S. (2011). *Android Development Short Course 2011*. Retrieved June 2, 2013, from Google Sites: https://sites.google.com/site/androidshortcourse2011/

Cunha, J., Fernandes, J. P., Martins, P., Mendes, J., & Saraiva, J. (2012). Smellsheet detective: A tool for detecting bad smells in spreadsheets. *In Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium* (pp. 243-244). IEEE.

Cunha, J., Fernandes, J. P., Ribeiro, H., & Saraiva, J. (2012). Towards a catalog of spreadsheet smells. *In Computational Science and Its Applications–ICCSA 2012* (pp. 202-216). Berlin Heidelberg: Springer.

Dougherty, D. (2013, May 29). *Handling Phone Call Requests the Right Way for Users.* Retrieved June 1, 2013, from Android Developers Blog: http://android-developers.blogspot.com/2013/05/handling-phone-call-requests-right-way.html

Faul, F., Erdfelder, E., Lang, A.-G., & Buchner, A. (2007). G*Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods, 39*, pp. 175-191.

F-Droid. (n.d.). Retrieved June 8, 2013, from F-Droid: http://f-droid.org

Felt, A. P. (2011). Android permissions demystified. *18th ACM conference on Computer and communications security* (pp. 627-638). ACM.

Fowler a, M. (n.d.). *CodeSmell*. Retrieved June 9, 2013, from martinfowler.com: http://martinfowler.com/bliki/CodeSmell.html

Fowler b, M. (n.d.). *Refactoring Home Page*. Retrieved June 9, 2013, from refactoring.com: http://www.refactoring.com/

Fowler, M. (1999). *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

Fowler, M. (n.d.). *CodeSmell.* Retrieved from martinfowler.com: http://martinfowler.com/bliki/CodeSmell.html

Franke, D., Elsemann, C., & Kowalewski, S. (2012). Reverse engineering and testing service life cycles of mobile platforms. *2012 23rd International Workshop on Database and Expert Systems Applications (DEXA)* (pp. 16 - 22). IEEE.

Franke, D., Elsemann, C., Kowalewski, S., & Weise, C. (2011). Reverse engineering of mobile application lifecycles. *18th Working Conference on Reverse Engineering (WCRE)* (pp. 283-292). IEEE.

Franke, D., Kowalewski, S., & Weise, C. (2012). A Mobile Software Quality Model. *2012 12th International Conference on Quality Software (QSIC)* (pp. 154-147). IEEE.

Franke, D., Royé, T., & Kowalewski, S. (2012). AndroLIFT: A Tool for Android Application Life Cycles. *VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle* (pp. 28-33). IARIA.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley.

Garcia, G. (n.d.). *SmellsToRefactorings.* Retrieved from Java.net: http://wiki.java.net/bin/view/People/SmellsToRefactorings

Garcia, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009 a). Identifying architectural bad smells. *In Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference* (pp. 255-258). IEEE.

Garcia, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009 b). Toward a catalogue of architectural bad smells. *In Architectures for Adaptive Software Systems* (pp. 146-162). Berlin Heidelberg: Springer.

Geron, T. (2012, February 2). *Do iOS Apps Crash More Than Android Apps? A Data Dive*. Retrieved from Forbes: http://www.forbes.com/sites/tomiogeron/2012/02/02/does-ios-crash-more-than-android-a-data-dive/

Google a. (n.d.). Retrieved June 9, 2013, from Android Developer Blog: http://android-developers.blogspot.com/

Google b. (n.d.). Retrieved June 10, 2013, from Android Developers: http://developer.android.com

Google c. (n.d.). *Android System Architecture*. Retrieved June 9, 2013, from Android Developers: http://developer.android.com/images/system-architecture.jpg

Google d. (n.d.). *App Components*. Retrieved June 10, 2013, from Android Developers: http://developer.android.com/guide/components/index.html

Google e. (n.d.). *Getting Started*. Retrieved June 7, 2013, from Android Developers: http://developer.android.com/training/index.html

Google f. (n.d.). *Google Developers University Consortium - Mobile Courses*. Retrieved July 3, 2013, from Google Developers: https://developers.google.com/university/courses/mobile

Google g. (n.d.). *MonkeySourceRandom.java*. Retrieved September 1, 2013, from Android Source Repository: https://android.googlesource.com/platform/development/+/gingerbread-release/cmds/monkey/src/com/android/commands/monkey/MonkeySourceRandom.java

Google h. (n.d.). *MonkeySourceRandom.java*. Retrieved September 1, 2013, from Android Source Repositories: https://android.googlesource.com/platform/development/+/android-4.3_r2.3/cmds/monkey/src/com/android/commands/monkey/MonkeySourceRandom.java

Google i. (n.d.). *Publishing Overview*. Retrieved June 4, 2013, from Android Developers: http://developer.android.com/tools/publishing/publishing_overview.html

Google i. (n.d.). *UI/Application Exerciser Monkey*. Retrieved September 1, 2013, from Android Developers: http://developer.android.com/tools/help/monkey.html

Google j. (n.d.). *Starting An Activity*. Retrieved June 4, 2013, from Android Developers: http://developer.android.com/training/basics/activity-lifecycle/starting.html

Google k. (n.d.). *Testing*. Retrieved June 5, 2013, from Android Developers: http://developer.android.com/tools/testing/index.html

Google l. (n.d.). *Activities*. Retrieved October 15, 2013, from Android Developers: http://developer.android.com/guide/components/activities.html

Google m. (n.d.). *Intent.* Retrieved from Android Developers: http://developer.android.com/reference/android/content/Intent.html

Google n. (n.d.). *Bundles.* Retrieved from Android Developer: http://developer.android.com/reference/android/os/Bundle.html

Google o. (n.d.). *Introduction to Android.* Retrieved from Android Developers: https://developer.android.com/guide/index.html

Griswold, W. G. (1991). *Program restructuring as an aid to software maintenance.*

Gundotra, V. (2013, May 15). *Google I/O 2013 Keynote.* Retrieved May 15, 2013, from Google Developers: https://developers.google.com/live/shows/517795853

Guy, R. (2009, Jan 19). *Avoiding memory leaks.* Retrieved from Android Developers Blog: http://android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html

Hackborn, D. (2009, Sep 09). *Android Developers.* Retrieved from Google Groups: https://groups.google.com/forum/#!topic/android-developers/zLpOGmvW2lE

Hafiz, M., Adamczyk, P., & Johnson, R. (2009). Systematically eradicating data injection attacks using security-oriented program transformations. *Engineering Secure Software and Systems*, pp. 75-90.

Hafiz, M., Overbey, J., Behrang, F., & Hall, J. (2013). OpenRefactory/C: An Infrastructure for Building Correct and Complex C Transformations.

Hamblen, M. (2011, July 26). *App Store, Android Market spur explosive app download growth.* Retrieved June 1, 2013, from Computer World: http://www.computerworld.com/s/article/9218654/App_Store_Android_Market_spur_explosive_app_download_growth

Hanenberg, S., Oberschulte, C., & Unland, R. (2003). Refactoring of aspect-oriented software. *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays)*, (pp. 19-35).

Henneke, C. (2013, July 29). *Android vs. iOS: Comparing the Development Process of the GQueues Mobile Apps.* Retrieved October 7, 2013, from GQueues: http://blog.gqueues.com/2013/07/android-vs-ios-comparing-development.html

Hermans, F., Pinzger, M., & Deursen, A. v. (2012). Detecting and visualizing inter-worksheet smells in spreadsheets. *Proceedings of the 2012 International Conference on Software Engineering* (pp. 441-451). IEEE Press.

Higo, Y., Kamiya, T., Kusumoto, S., & Inoue, K. (2004). Refactoring support based on code clone analysis. *Product Focused Software Process Improvement*, pp. 220-233.

Hu, W. C. (2010). Smartphone Software Development Course Design Based on Android. *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)* (pp. 2180-2184). IEEE.

IDC. (2013, May 16). *Android and iOS Combine for 92.3% of All Smartphone Operating System Shipments in the First Quarter While Windows Phone Leapfrogs BlackBerry, According to IDC*. Retrieved June 2, 2013, from IDC: http://www.idc.com/getdoc.jsp?containerId=prUS24108913

Industrial Logic. (2005, May). *Smells to Refactorings Cheatsheet.* Retrieved from Industrial Logic: http://www.industriallogic.com/blog/smells-to-refactorings-cheatsheet/"

ISO. (1999). *Standard 14764 on Software Engineering - Software Maintenance.* ISO/IEC.

ISO. (2011). *ISO/IEC 25010:2011: System and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models.* Geneva: ISO Copyright Office.

Jones, S. P. (1997, April). *A new view of guards.* Retrieved from Microsoft Research: http://research.microsoft.com/en-us/um/people/simonpj/Haskell/guards.html

Kerievsky, J. (2005). *Refactoring to patterns.* Pearson Deutschland GmbH.

Kerievsky, J. (2013, Aug). *Joshua Kerievsky - Google+*. Retrieved from Google+: https://plus.google.com/117748572268183690731/posts/7XiQgNpaYCQ

Kim, M. Z. (2012). A field study of refactoring challenges and benefits. *In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (p. 50). ACM.

Laddad, R. (2003). *Aspect-Oriented Refactoring, parts 1 and 2, The Server Side, 2003.*

Lee, W. (2012). *Beginning Android 4 Application Development.* Wrox; 1 edition (March 6, 2012).

Leitch, R., & Stroulia, E. (2003). Understanding the economics of refactoring. *EDSER-5 5th International Workshop on Economic-Driven Software Engineering Research*, (p. 44).

Levy, A. (2012, 10 26). *Crash Reporting Trends for Mobile App Developers.* Retrieved June 2, 2013, from DroidCon 2012: http://uk.droidcon.com/sessions/crash-reporting-trends-for-mobile-app-developers/

Li, H., & Thompson, S. (2010). Similar code detection and elimination for erlang programs. *Practical Aspects of Declarative languages*, pp. 104-118.

Liu, Z., Gao, X., & Long, X. (2011). Adaptive random testing of mobile application. *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on* (pp. V2-297). ACM.

Lockwood b, A. (2013, Apr 15). *Activitys, Threads, & Memory Leaks.* Retrieved from Android Design Patterns: http://www.androiddesignpatterns.com/2013/04/activitys-threads-memory-leaks.html

Lockwood, A. (2013, Jan 14). *How to Leak a Context: Handlers & Inner Classes.* Retrieved from Android Design Patterns: http://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html#more

Lucas, A. (2012, February 9). *Share With Intents.* Retrieved June 2, 2013, from Android Developers Blog: http://android-developers.blogspot.com/2012/02/share-with-intents.html

MacHiry, A. T. (2012). *Dynodroid: An input generation system for Android apps.* Technical Report.

MacHiry, A., Tahiliani, R., & Naik, M. (2012). *Dynodroid: An input generation system for Android apps.* Technical Report.

Maji, A. K., Arshad, F. A., Bagchi, S., & Rellermeyer, J. S. (2012). An empirical study of the robustness of inter-component communication in Android. *2012 42nd Annual IEEE/IFIP International Conference on In Dependable Systems and Networks (DSN)* (pp. 1-12). IEEE.

Maji, A. K., Hao, K., Sultana, S., & Bagchi, S. (2010). Characterizing failures in mobile oses: A case study with android and symbian. *2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)* (pp. 249-258). IEEE.

Maruyama, K., & Tokoda, K. (2008). Security-aware refactoring alerting its impact on code vulnerabilities. *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific* (pp. 445-452). IEEE.

Meier, R. (2012). *Professional Android 4 Application Development.* Wrox; 3 edition (May 1, 2012).

Mens, T. &. (2004). A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 126-139.

Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, Elsevier.

Meyer, B. (2008). Seven principles of software testing. *Computer* , 99-101.

Meyer, H. (2008). Calculating the semantic conformance of processes. *Business Process Management Workshops* (pp. 473-483). Springer Berlin: Heidelberg.

Moha, N., Gueheneuc, Y.-G., Duchien, L., & Le Meur, A.-F. (2010). DECOR: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 20-36.

Monteiro, M., & Fernandes, J. (2005). Towards a catalog of aspect-oriented refactorings. *Proceedings of the 4th international conference on Aspect-oriented software development* (pp. 111-122). ACM.

Murphy-Hill, E., Parnin, C., & Black, A. P. (2012). How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 5-18.

Opdyke, W. F. (1990). Refactoring: An aid in designing application frameworks and evolving object-oriented systems. *1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications.* SOOPPA.

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks.* Doctoral dissertation, University of Illinois.

Pargianowicz, M. (2012, November 5). *Clover Goes Mobile.* Retrieved June 1, 2013, from Atlassian: http://blogs.atlassian.com/2012/11/eclipse-plugin-android-clover-code-coverage-continuous-integration/

Perez, S. (2013, April 8). *Nearly 60K Low-Quality Apps Booted From Google Play Store In February, Points To Increased Spam-Fighting.* Retrieved June 2, 2013, from Techcrunch: http://techcrunch.com/2013/04/08/nearly-60k-low-quality-apps-booted-from-google-play-store-in-february-points-to-increased-spam-fighting/

Piveta, E., Hecht, M., Pimenta, M., & Price, R. (2006). Detecting Bad Smells in AspectJ. *J. UCS*, 811 - 827.

Popadopoulos, P. (2013, April 9). *What BigData analysis tells us about Android errors.* Retrieved June 9, 2013, from droidcon 2013: http://de.droidcon.com/2013/sessnio/what-bigdata-analysis-tells-us-about-android-errors

Ramos, R., Piveta, E., Castro, J., Araujo, J., Moreira, A., Guerreiro, P., . . . Price, T. (2007). Improving the Quality of Requirements with Refactoring. *VI Simposio Brasileiro de Qualidade de Software--SBQS2007.* 2007.

Rao, A., & Reddy, N. (2007). *Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix 1.*

Ratzinger, J., Fischer, M., & Gall, H. (2005). Improving evolvability through refactoring. *ACM SIGSOFT Software Engineering Notes*, pp. 1-5.

Raymond, E. (1999). The cathedral and the bazaar. . *Knowledge, Technology & Policy, 12(3)*, 23-49.

research2guidance. (2011). *Global mHealth Survey: The Developer View (2011-2016).* research2guidance.

Snake. (n.d.). Retrieved June 5, 2013, from Wikipedia: http://en.wikipedia.org/wiki/Snake_(video_game)

Stangroom, J. (2014). *Wilcoxon Signed-Rank Test*. Retrieved from Social Science Statistics.

Stroulia, E., & Kapoor, R. (2001). Metrics of refactoring-based development: An experience report. (pp. 113-122). Springer.

Trochim, W. (2006). *Threats to Conclusion Validity*. Retrieved July 3, 2013, from Research Methods Knowledge Base: http://www.socialresearchmethods.net/kb/concthre.php

Tsantalis, N., & Chatzigeorgiou, A. (2011). Ranking refactoring suggestions based on historical volatility. *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on* (pp. 25-34). IEEE.

van der Merwe, H., van der Merwe, B., & Visse, W. (2012). Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 1-5.

Victor, H. (2013, July 24). *Android's Google Play beats App Store with over 1 million apps, now officially largest.* Retrieved from phoneArena.com: http://www.phonearena.com/news/Androids-Google-Play-beats-App-Store-with-over-1-million-apps-now-officially-largest_id45680

Viswanathan, P. (2012, October 22). *iOS App Store Vs. Google Play Store for App Developers.* Retrieved from About.com: http://mobiledevices.about.com/od/additionalresources/a/Ios-App-Store-Vs-Google-Play-Store-For-App-Developers.htm

Ward, C. (n.d.). *About the Portland Form*. Retrieved from Cunningham & Cunningham: http://c2.com/ppr/about/portland.html

Wilson, J. (2011, December 19). *Watch out for XmlPullParser.nextText().* Retrieved July 1, 2013, from Android Developer's Blog: http://android-developers.blogspot.com/2011/12/watch-out-for-xmlpullparsernexttext.html

Zhang, P., & Elbaum, S. (2012). Amplifying tests to validate exception handling code. *International Conference on Software Engineering* (pp. 595-605). IEEE Press.

Appendix A – Application Catalog

Table 32. Free Open Source Software (FOSS) Application Catalog

Application	Version	Complex?	Source?	Can Run?
0xbench	1.1.5	Yes	Yes	Yes
24h Analog Clock	0.4.1	Yes	Yes	Yes
2Degrees Toolbox (Ad-Free)	1.5.8	Yes	No	No
920 Editor	12.11.23	Yes	No	No
A Comic Viewer	1.4.1.4	Yes	No	No
A2DP Volume	2.9.7	No	No	No
aagtl	1.0.31	Yes	Yes	Yes
Aard	1.6.4	Yes	Yes	Yes
aAuroraApp	1.04	Yes	Yes	Yes
Abstract Art	1.1.3	No	No	No
aCal	1.62	Yes	Yes	Yes
AdAway	2.3	No	No	No
AdBlock	0.5	No	No	No
Adblock Plus	1.1	No	No	No
adbWireless	1.5.4	No	No	No
Addi	1.98	Yes	Yes	Yes
ADSdroid	1.3	Yes	Yes	Yes
ADW.Launcher	1.3.6	Yes	No	No
AFWall+	1.2.0	Yes	No	No
Agit	1.34	Yes	Yes	Yes
aGrep	0.2.7	Yes	No	No
aHighScoreDemo	1	Yes	No	No
AiCiA	2012.1224.1	Yes	Yes	Yes
AirPush Detector	3.1	Yes	Yes	Yes
Alarm Klock	1.7	Yes	Yes	Yes
aLogcat	2.6.1	Yes	Yes	Yes
Amarino	not available	No	No	No
aMetro	1.1.5	Yes	Yes	Yes
Ampache Provider	1.24	No	No	No
AnaCam	1.2	No	No	No
aNarXiv	1	Yes	No	No
AnCal	1.6	Yes	Yes	Yes
And Bible	1.8.1	Yes	Yes	Yes
andFHEM	1.5.8	No	No	No
AndFML	0.5.0 Beta	No	No	No
andLess	1.3.5	Yes	No	No
Andlytics	2.4.3	No	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Andnav	N/A	Yes	No	No
Andor's Trail	0.6.12	No	No	No
AndQuote	0.3.3	Yes	Yes	Yes
andRoc	399	No	No	No
Android Music Player	4.0.4b3	Yes	No	No
Android Resources	0.0.8	No	Yes	Yes
Android SuperGenPass utility	v2.2.2	Yes	No	No
Android Terminal Emulator	1.0.52	Yes	No	No
Android Tipitaka	8.3	No	No	No
android-countdownalarm	1.0.9	Yes	No	No
Androidomatic Keyer	1	No	No	No
Android's Fortune	1.1.9	Yes	No	No
android-simple-loan-calculator	4	Yes	No	No
androidVNC	0.5.0	Yes	Yes	Yes
Androsens	1.2	Yes	Yes	Yes
AndroSS	0.4.3	Yes	No	No
Androzic	1.7.2	Yes	Yes	Yes
AndStatus	1.13.1	Yes	Yes	Yes
AndSys	0.1.l.11	Yes	Yes	Yes
AndTweet	1.2.4	No	No	No
Angulo	1.2	Yes	Yes	Yes
AnkiDroid	2	Yes	No	No
Anode Internode Widget (Beta)	0.6	No	No	No
Anstop	1.5	Yes	Yes	Yes
AntennaPod	0.9.7.2	Yes	Yes	Yes
Antikythera Simulation	0.97	Yes	Yes	Yes
Anton Widget	1.7.0	Yes	No	No
Any Cut	0.5	Yes	No	No
AnyMemo	8.3.0	Yes	No	No
AnyMemo Free: Flash Card Study	10.1	Yes	No	No
AnySoftKeyboard	20130222-skinny-eye-candy	Yes	No	No
AnySoftKeyboard: Classic PC Theme	1.0.1-201200304	No	No	No
AnySoftKeyboard: Danish	20100614	No	No	No
AnySoftKeyboard: Esperanto	20100613	No	No	No
AnySoftKeyboard: French	20111029	No	No	No
AnySoftKeyboard: Georgian	2.0.204	No	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
AnySoftKeyboard: Greek	20110717	No	No	No
AnySoftKeyboard: Hebrew	20121101	No	No	No
AnySoftKeyboard: Lithuanian	20111024	No	No	No
AnySoftKeyboard: Magyar	20120208	No	No	No
AnySoftKeyboard: Pali	0.2	No	No	No
AnySoftKeyboard: Persian	20120613-persian-numbers	No	No	No
AnySoftKeyboard: SSH	0.4.20120611-beta	No	No	No
AOKP Backup	1.51	Yes	No	No
APG	1.0.8	Yes	No	No
APKtor (public Market)	1.0.8.7	Yes	No	No
Apndroid Free	4.0.8	Yes	No	No
ApnSwitch	1.20111211	No	No	No
Apollo	1	Yes	No	No
App Tracker	1.0.9	Yes	Yes	Yes
AppAlarm Pro	1.2.7	Yes	Yes	Yes
Apple Crunch	2.2.1	Yes	No	No
Apple][	1.4.1	Yes	No	No
AppLocker	1.2	Yes	Yes	Yes
Apps Organizer	1.5.19	Yes	Yes	Yes
APV PDF Viewer	0.4.0	Yes	Yes	Yes
Ardroid	1	No	No	No
Aripuca GPS Tracker	1.3.0	Yes	Yes	Yes
Arity	1.27	Yes	Yes	Yes
arXiv mobile	2.0.20	Yes	Yes	Yes
AsciiCam	1.1	Yes	Yes	Yes
Asqare	1.3	Yes	Yes	Yes
aSQLiteManager	3.5	Yes	Yes	Yes
Authenticator	2.21	Yes	Yes	Yes
AutoAnswer	1.5	Yes	Yes	Yes
Auto-Away	1.2.1	Yes	Yes	Yes
Autostarts	1.7.7	Yes	No	No
AWOL: ArchWiki Offline	0.3.0 (Alpha III)	Yes	No	No
Baby Care Timer	1.4	Yes	Yes	Yes
Bankdroid	1.9.5.3	Yes	No	No
Barcode Box 2	1.1.1	Yes	Yes	Yes
Barcode Scanner	4.3.1	Yes	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Barnacle Wifi Tether	0.6.7 (evo)	Yes	Yes	Yes
Battery Circle	1.81	Yes	No	No
Battery Dog	0.1.1	Yes	Yes	Yes
Battery Level	0.3	Yes	No	No
Battery Widget	0.6.2	No	No	No
Beam File	1	Yes	No	No
Beem	0.1.8	Yes	Yes	Yes
Binaural Beats Therapy	2.0.3	Yes	Yes	Yes
Birthday Adapter	1.12	No	No	No
Birthdroid	0.5	Yes	Yes	Yes
Bitbeaker	2.3.1	Yes	No	No
Bitcoin Wallet	2.16	Yes	No	No
Bitcoinium Prime	1.5.0	Yes	Yes	Yes
BitcoinSpinner	0.7.3b	Yes	Yes	Yes
Bites	1.3	Yes	Yes	Yes
BlackJack Trainer	0.1	Yes	Yes	Yes
BlitzMail	0.2	Yes	No	No
Blockinger	1	Yes	Yes	Yes
Blokish	2	Yes	Yes	Yes
BlueGps	1.2.5	Yes	No	No
Bluetooth RepRap	0.3.0	No	No	No
Bluez IME	1.16	No	No	No
Blurred Lines	1.1-2	No	No	No
Boarder	0.23	Yes	No	No
BoardGameGeek	4.2	Yes	Yes	Yes
Bodhi Timer	2.1	Yes	Yes	Yes
Bomber	1	Yes	No	No
Book Catalogue	4.2	Yes	Yes	Yes
BookWorm	1.0.18	Yes	Yes	Yes
BotBrew Basil	0.0.1.24	No	No	No
BrowserQuest	1.0.1	Yes	No	No
Bubble	1.9.4	Yes	Yes	Yes
Budoist	1.3.1	No	Yes	Yes
build.prop Editor	2.0.1	Yes	No	No
Bussorakel	0.7.0	Yes	No	No
Bysyklist Oslo	1.1.2	Yes	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
c:geo	2013.02.21-fdroid	Yes	Yes	Yes
CACertMan	0.0.2.20111012	No	No	No
Calculator	3.1.2	Yes	No	No
CalDAV Sync Adapter	1.7	Yes	No	No
Call Meter 3G	3.8.4	Yes	No	No
CallerID	1.6	Yes	No	No
Campyre	1.0.1	No	No	No
Canadian Weather EmWeatherFree	0.9.8.1	Yes	No	No
Car Cast	1.0.143	Yes	Yes	Yes
Car Report	2.9.1	Yes	No	No
CatLog	1.4.3	Yes	No	No
Charmap	1.0.1	Yes	Yes	Yes
ChartDroid Core	2.0.0	No	No	No
Chess Walk	1.5.1	Yes	Yes	Yes
Chinese Checkers	0.9	Yes	No	No
Chinese French Dictionary	1.7.1	No	No	No
Chord Reader	1.0.1	Yes	Yes	Yes
Chroma Doze	1.1.1	Yes	Yes	Yes
CIDR Calculator	1.16	Yes	Yes	Yes
CMIS Browser	0.9.6	Yes	No	No
Coin Flip	5.3.2	Yes	Yes	Yes
Coin Flip Addon	1.3	No	No	No
Color Picker	3	Yes	Yes	Yes
Coloring for Kids	1	Yes	Yes	Yes
Comics Reader	1.1.612	Yes	Yes	Yes
Compass	0.1	Yes	Yes	Yes
Compass Keyboard	v1.4	Yes	No	No
Concrete Estimator	1.1	Yes	No	No
Congress	4.1.3	Yes	No	No
ConnectBot	1.7.1	No	No	No
Contact Owner	2.2	Yes	Yes	Yes
Contact Widget	1.0.2	No	No	No
Contact-sync	1.0.1	Yes	No	No
Cool Reader	3.1.2-27	Yes	Yes	Yes
Copy to Clipboard	1	Yes	No	No
Corporate Addressbook	2.0.6	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Cowsay	1.3	Yes	Yes	Yes
CPAN Sidekick	0.5.1	Yes	Yes	Yes
CPU Spy	0.4.0	Yes	No	No
Crosswords	4.4	Yes	No	No
CSipSimple	0.04-04	No	No	No
CycleStreets	1.5	Yes	Yes	Yes
DAAP	.9.7+	No	No	No
Daily Money	0.9.8-121107-freshly	Yes	No	No
Dalvik Explorer	3.6	Yes	Yes	Yes
DashClock Widget	1.4.3	No	No	No
DashClock: DashCricket	1.3	No	No	No
DashClock: K-9 Unread Count	1.1.1.0	No	No	No
Dazzle Configurable Switcher	2.9	No	No	No
Dear Future Self	0.2.6	Yes	No	No
Desk Clock	Varies with device	Yes	No	No
DesktopLabel	1.4.0	No	No	No
Device Frame Generator	3.0.3	Yes	No	No
Dialer2	2.9	Yes	Yes	Yes
Diaspora Webclient	1.6.3	No	No	No
DictionaryForMIDs	0.91.1	Yes	No	No
DieDroid	1.4.0	Yes	Yes	Yes
Diktofon	0.9.80	Yes	Yes	Yes
Diode	1.0.9	No	No	No
Diode Reddit Browser	1.0.8	No	No	No
DiskUsage	3.4	Yes	Yes	Yes
DiveDroid	0.6	Yes	No	No
DIYgenomics	1	Yes	No	No
DNCViolation	0.2	Yes	No	No
Dodge	1.4.3	Yes	Yes	Yes
Dongsa	2	Yes	No	No
Doom	fix crash2	Yes	No	No
DotDash Keyboard	1.1.5	Yes	No	No
Dots'n'Boxes	0.5	Yes	Yes	Yes
DriSMo	1.0.3	Yes	Yes	Yes
Droid Draw	1.0.0	Yes	Yes	Yes
Droid Weight	1.5.4	Yes	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
DroidAtomix	1.0.1	Yes	No	No
DroidFish	1.47	No	No	No
DroidLife	2.5	Yes	Yes	No
DroidSat	2.47	Yes	Yes	Yes
DroidSeries	0.1.5-6	Yes	Yes	Yes
DroidTracker	1.21	Yes	No	No
DroidWall - Android Firewall	1.5.5	Yes	No	No
DroidWeight	1.5.4	Yes	No	No
DroidZebra	1.4.1	Yes	No	No
DSub	3.8.2	Yes	No	No
Duck Duck Go	0.1	Yes	Yes	Yes
dynalogin	1.0.1	Yes	No	No
E numbers	1.3.0	Yes	No	No
EncPassChanger	2	Yes	No	No
EP Mobile	2.0.1	Yes	No	No
ePUBator	0.1	Yes	Yes	Yes
Ermete SMS	2.0.4	Yes	Yes	Yes
Every Locale	1.0.7	Yes	Yes	Yes
External IP	1.2-change_server-2	Yes	No	No
Faenza ADW Theme	1	No	No	No
Fake Dawn	1.2	Yes	Yes	Yes
Falling Blocks	0.3.5	Yes	Yes	Yes
FAST App Search Tool	3.7	Yes	Yes	Yes
FasterGPS	1.6	Yes	Yes	Yes
Fauxlight	1.2	Yes	No	No
FBReader	1.7.3-Honeycomb	Yes	No	No
FBReader TTS plugin	1.2	No	No	No
FBReader TTS+ Plugin	2.0.4	No	No	No
FBReaderJ	1.8.1.	Yes	No	No
F-Droid	0.42	Yes	Yes	Yes
FDroid Repository	0.21	Yes	No	No
FFvideo Live Wallpaper	0.9.0	No	No	No
File Explorer	0.1	Yes	No	No
FindMyPhone	1.22B	Yes	No	No
Firefox	20	Yes	No	No
Flashify	1.1	Yes	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Flickr Viewer HD	1.2.8	Yes	No	No
Flygtider	0.61	Yes	No	No
FON Access	1.4.7	Yes	No	No
Footguy	1.2	No	No	No
Forecast widgets	1	No	No	No
Fortune	v7	No	No	No
FOSDEM schedules	2.0.0	Yes	No	No
Free Fall	1.6.1	Yes	Yes	Yes
Friendica	8	Yes	Yes	Yes
FrostWire	1.0.5	Yes	Yes	Yes
Frozen Bubble	1.14	Yes	No	No
FTP Server	2.5.2	Yes	Yes	Yes
Funambol Sync	10.0.6	No	No	No
GAE Proxy	0.21.5	No	No	No
GameBoid	1.3.2	Yes	No	No
GameMaster Dice	0.1.5	Yes	No	No
gAppsLauncher	2.1	Yes	No	No
GBCoid	1.8.5	Yes	Yes	Yes
GCstar scanner	1	Yes	Yes	Yes
GCstar Viewer	2.8	Yes	Yes	Yes
GeoBeagle: Geocaching	1.4.15	Yes	No	No
GeoPaparazzi	3.3.0	Yes	Yes	Yes
Georgian Fonts Installer	4.1	No	Yes	Yes
Ghost Commander	1.43.3b1	Yes	Yes	Yes
Gibberbot	0.0.10-RC6	No	No	No
Giggity	1	Yes	No	No
GitHub	1.6.1	Yes	Yes	Yes
GL TRON	1.1.2	Yes	Yes	Yes
Gloomy Dungeons 3D	2013.03.07.0416	Yes	Yes	Yes
GM Dice	0.1.5	Yes	Yes	Yes
Gmote	2.0.2	Yes	Yes	Yes
GnuCash	1.1.2	Yes	Yes	Yes
gobandroid ai gnugo	0.6	No	No	No
Google I/O 2012	0.19	Yes	No	No
GPSLogger	27-fdroid	Yes	Yes	Yes
GTalkSMS	4.3	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Hacker's Keyboard	v1.33	Yes	Yes	Yes
Ham	1.5.7	Yes	Yes	Yes
HandyNotes	1.5	No	No	No
Hdhomerun Signal Meter	1.21	No	No	No
Hearing Saver	3.1	Yes	No	No
Hex	2.3	Yes	No	No
Hexiano	0.82.1 ALPHA	Yes	No	No
HFR4droid	0.7.4	No	No	No
HN	1.9.1	Yes	Yes	Yes
HNdroid	0.2.1	No	No	No
Holo Counter	1	Yes	No	No
HoloKen	1.1.1	No	No	No
Horaires TER SNCF	3.0.1 (bugfix)	Yes	Yes	Yes
Hot Death	1.0.7	Yes	Yes	Yes
Hotspot Login	0.2.0	Yes	Yes	Yes
HSTempo	1.2.1.1	Yes	Yes	Yes
httpmon	0.4.10	Yes	Yes	No
HunkyMod	0.3	Yes	No	No
HypnoTwister	1.4.1	No	No	No
iFixit	1.3.2	Yes	Yes	Yes
Import Contacts	1.3.1	Yes	No	No
Inetify	2.0.3	Yes	No	No
Intent Intercept	2.01	Yes	No	No
Irssi ConnectBot	1.7.1-irssi	Yes	Yes	Yes
Iteration (generative art)	3.15	No	No	No
iTLogger	1.0.0	Yes	Yes	Yes
Jamendo	1.0.6-legacy	Yes	Yes	Yes
Japanese Name Converter	1	Yes	Yes	Yes
Jelly Clock	1.21	No	No	No
JLyr Lyrics	1.4	Yes	Yes	Yes
Jupiter Broadcasting	2.2.3	Yes	No	No
Just A Damn Compass	1.1	Yes	No	No
Just Player	3.37	Yes	No	No
Just Player Plugin: Ampache	1.24	No	No	No
Just Player Plugin: Podcast	1.1a	No	No	No
Just Sit	0.3.3	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
K-9 Mail	4.201	Yes	No	No
Kaleidoscope	1.1.2	Yes	Yes	Yes
Kanji draw	1	Yes	Yes	Yes
Kanji Flashcards	1.5.5	Yes	No	No
KeePassDroid	1.99.5	Yes	Yes	Yes
KeepingTracks	1.02	Yes	No	No
KeepScore	1.2.3	Yes	Yes	Yes
kensa	Not Available	No	No	No
Kitchen Timer	1.1.6	Yes	Yes	Yes
Klaxon	0.27	Yes	Yes	Yes
Knave Arthur's Sword	N/A	Yes	No	No
Kontalk	2.2.6	Yes	Yes	Yes
kwaak3	Not Available	No	No	No
Lampshade	1.1.7	Yes	Yes	Yes
Language Picker Widget	1.1	No	No	No
LCARS Wallpapers	1.1.1	No	No	No
LDAP Sync	1.5	Yes	No	No
Learn Music Notes	1.4	Yes	Yes	Yes
Lesser Pad	0.33	Yes	Yes	Yes
Lexic	0.8.1	Yes	Yes	Yes
LG Touch LED	1.2.4	Yes	No	No
Libra - Weight Manager	2.7.7	Yes	No	No
Libre Droid	1.4	Yes	Yes	Yes
LifeSaver2	1	Yes	No	No
Lightning	2.4	Yes	Yes	Yes
LilDebi	0.3	Yes	No	No
Limbo PC Emulator (QEMU x86)	0.9.7	Yes	No	No
Link Shortener for TinyURL	1.0.9	Yes	No	No
Linphone	1.3.2	Yes	Yes	Yes
Littre French dictionary	0.7.5	Yes	No	No
Local Calendar	1.1	No	No	No
Location Cache Map	0.6	Yes	No	No
Location Log	1.0.5	Yes	No	No
Lock Pattern Generator	2.0.1	Yes	Yes	Yes
Log Collector	1.1.0	Yes	No	No
Logback-Android	1.0.0-3	No	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Logcat to UDP	0.5	Yes	Yes	Yes
Lumicall	1.9.10	Yes	No	No
Magic Bus	1	No	No	No
Mahjongg Builder	1.4.4	Yes	Yes	Yes
MAME4droid	1.5.2	Yes	No	No
Maniana	1.2.25	Yes	Yes	Yes
Manpages	1.51	No	No	No
Marine Compass	1.2.4	Yes	Yes	Yes
Markers	1.1.1	Yes	Yes	Yes
MathDoku	1.95d	Yes	Yes	Yes
Mathdroid	2.9.2	Yes	Yes	Yes
MemoPad	1.0.2	Yes	Yes	Yes
Memory	1.9.0	Yes	No	No
Meritous	0.0.2	Yes	No	No
MidiSheetMusic	2.5.1	Yes	Yes	Yes
Migraine Tracker	0.9b	Yes	Yes	Yes
Mileage	3.1.1	Yes	Yes	Yes
Mini vMac	1.1.0	Yes	No	No
miniNoteViewer	0.4	Yes	No	No
Ministro	7	No	No	No
Mirrored	0.2.3	Yes	No	No
Missile intercept	0.5.1	Yes	No	No
Missing Keys	1.1	Yes	No	No
Mixare	0.9.2	Yes	No	No
Mnemododo	N/A	Yes	No	No
MobileOrg	0.9.8	Yes	Yes	Yes
MobilePrint	0.5	Yes	No	No
MobileWebCam	2.98	Yes	No	No
Moloko	0.17.3b	Yes	Yes	Yes
MorbidMeter	1.2.1	No	No	No
mOTP	1.5	Yes	Yes	Yes
MPDroid	1.05	Yes	Yes	Yes
MrWhite	1.1	Yes	No	No
MTG Familiar	1.8.1	No	No	No
Multi Sms	2.3	Yes	Yes	Yes
Multipicture Wallpaper	0.6.11	No	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Munich Transit Map	0.3	Yes	No	No
MuPDF	1.2	Yes	Yes	Yes
Mupen64 Plus AE	2.1.3	No	No	No
Mustard	0.4.0beta8	No	No	No
My Expenses	1.6.8	Yes	Yes	Yes
My Location Widget	6	No	No	No
My Tracks	2.0.4	Yes	No	No
My Wallpaper	1.1.9	No	No	No
myCityBikes bysykkel	1.0.1	Yes	No	No
myLock utilities	42	Yes	Yes	Yes
MythDroid	0.6.2	No	No	No
mythmote	1.8.6	No	Yes	Yes
MyTracks	N/A	Yes	No	No
NagMonDroid	1.5.3	No	No	No
Nagroid	0.0.7	No	No	No
nanoConverter	0.8.98	Yes	Yes	Yes
Napply	1.2	No	No	No
Narau	0.8.1	Yes	Yes	Yes
NDKmol	0.92	Yes	No	No
nds4droid	26	Yes	Yes	Yes
Nectroid	1.2.4	Yes	Yes	Yes
Neko Project II for Android	20120217	Yes	No	No
Nesoid	2.5	Yes	Yes	Yes
NetCounter	0.14.1	Yes	No	No
NetHack	1.3.3	Yes	No	No
NetTTS	0.2.1	Yes	Yes	Yes
Network	1.02	Yes	Yes	Yes
Network Discovery	0.3.5	Yes	No	No
Network Log	2.17.0	Yes	Yes	Yes
Network Tester	1.1	Yes	Yes	Yes
NewsBlur	1.1	Yes	Yes	Yes
Newton's Cradle	2.2.0	Yes	No	No
NFC Reader	0.1	Yes	No	No
NFCard	1.2.120228	Yes	No	No
Nice Compass	1.3	Yes	No	No
nicoWnnG	2012.1125.1.1305	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Night Dock	1.1.0	Yes	No	No
No-frills CPU Control CLASSIC	1.27	Yes	No	No
Noiz2	1.4.0	No	Yes	Yes
NoNonsense Notes	4.4.1	Yes	No	No
Notepad	1.06	Yes	Yes	Yes
Notification Plus	1.1	Yes	Yes	Yes
Notify Lite	3.29.2	Yes	Yes	Yes
NSTools	1.16	Yes	Yes	Yes
NTPSync	1.3	No	No	No
NXT Remote Control	1.4	No	No	No
Nyan	1.8	No	No	No
NZ Tides	5	Yes	Yes	Yes
OASVN Pro	1.0.10	Yes	Yes	Yes
Obsqr	2.5	Yes	No	No
OCaml Toplevel on Android	1	Yes	No	No
OCR Test	0.5.12	Yes	Yes	Yes
OctoDroid	3.2	No	No	No
ODK Collect	1.2.2	Yes	Yes	Yes
Offline Calendar	1	Yes	No	No
OI About	1.1	No	No	No
OI File Manager	2.0.2	Yes	Yes	Yes
OI Flashlight	1.1	Yes	No	No
OI Notepad	1.4.0.7	Yes	No	No
OI Safe	1.4.1	Yes	No	No
OI Shopping list	1.6	Yes	No	No
Omnidroid	0.2.1	Yes	Yes	Yes
Open BART	0.5.4	Yes	Yes	Yes
Open Explorer Beta	0.212	Yes	Yes	Yes
Open GPS Tracker	1.3.2-osmupdate	Yes	No	No
Open Manager	2.1.8	Yes	Yes	Yes
Open Training	0.2.3	Yes	Yes	Yes
OpenDocument Reader	2.0.7	Yes	No	No
OpenGPX	1.1.0	Yes	Yes	Yes
OpenMensa	0.4	Yes	No	No
OpenSudoku	1.1.5	Yes	Yes	Yes
OpenTimer	2.1	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
OpenVPN Settings	0.4.14	Yes	Yes	Yes
OpenWatch Recorder	1.4	Yes	No	No
OpenWnn Legacy	1.3.3	Yes	No	No
OrbitalLiveWallpaper	2	No	No	No
Orbot	0.2.3.23-rc-1.0.11-RC5-test2	Yes	Yes	Yes
Orweb v2	0.4.4	No	No	No
OS Monitor	2.0.5	Yes	Yes	Yes
OSChina	1.7.4	Yes	No	No
OsciPrime Oscilloscope	Dagobert	Yes	No	No
OsmAnd~	0.8.2	Yes	Yes	Yes
Osmdroid	2.0.0	Yes	No	No
OSMTracker	0.6.3	Yes	Yes	Yes
OTPdroid	2.1	No	No	No
ownCloud	1.4.0	Yes	Yes	Yes
PactrackDroid	1.3.1	Yes	Yes	Yes
PageTurner	2.0.8	Yes	Yes	Yes
Password Hash	1.3.2	No	No	No
PasswordMaker Pro	1.1.7	Yes	Yes	Yes
Pedometer	1.4.1	Yes	Yes	Yes
Penroser	1.2	No	No	No
Peppy Flowers	3.2	No	No	No
PerApp	1.02	No	No	No
Periodical	0.1	Yes	Yes	Yes
Permission Friendly Apps	1.4.1	No	No	No
Permissions	1.2	No	No	No
Persian Calendar	3.06	Yes	Yes	Yes
Petronius	1.3	Yes	Yes	Yes
Pinboard	2.3.0	Yes	Yes	Yes
Pixelesque	1.2.01	Yes	Yes	Yes
PlusMinusTimesDivide	2013.02.03.13	Yes	Yes	Yes
Pocket Bandit	1.1	Yes	No	No
Pocket Talk	2.5	Yes	Yes	Yes
Podax	5.8	Yes	Yes	Yes
Polite Droid	1.4	Yes	Yes	Yes
Pomodoro Tasks	1.5	Yes	Yes	Yes
Pony Express	1.1	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Port Scandroid	1.0.0	Yes	No	No
Postcode	1.1	Yes	No	No
PPSSPP	0.7.6	Yes	No	No
Practice Hub	2	Yes	No	No
PrBoom For Android	1.3.1-beta	Yes	No	No
primitive ftpd	1.0.1	Yes	No	No
pttdroid	1.2	Yes	No	No
PurpleDock	1.0.1-rc.1	Yes	Yes	Yes
Pushup Buddy	1	Yes	Yes	Yes
pyLoad	0.3.3	Yes	Yes	Yes
Quake	1	Yes	No	No
Quake2	1.91	Yes	Yes	Yes
Quick Battery	0.8.2	No	No	No
Quick Settings	1.9.9.3	Yes	Yes	Yes
QuickDic	4.0.1	Yes	Yes	Yes
Quickdroid	4.3	Yes	Yes	Yes
Quill	10.3	Yes	No	No
QuiteSleep	3	Yes	Yes	Yes
radio reddit	0.6	No	No	No
Rage Maker	1.5.4	Yes	Yes	Yes
RateBeer Mobile	1.6.0	No	No	No
Recent Contacts Widget	0.4.4	No	No	No
reddit by brian	2.3	No	No	No
RedReader Beta	1.6.2	No	No	No
ReGalAndroid	1.0.1	No	No	No
Rehearsal Assistant	0.8.2	Yes	Yes	Yes
ReLaunch	1.3.8	Yes	No	No
Remembeer	1.3.0	Yes	Yes	Yes
Remote for VLC	0.5.5	No	No	No
Remote Notifier	0.2.8	No	Yes	Yes
Remuco	0.9.6	No	No	No
Replica Island	1.4	Yes	Yes	Yes
Rights alert	0.3a	Yes	Yes	Yes
Ringdroid	2.6	Yes	No	No
RingyDingyDingy	0.7.4	Yes	Yes	Yes
RMaps	0.9.1	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
robotfindskitten	1.0.701	Yes	Yes	Yes
Rokon Game Engine	1.1.1	No	No	No
Rotation Lock	1.6	No	No	No
RoundR	2.0.0.1	Yes	Yes	Yes
RPN	2.0.3	Yes	No	No
s It Christmas?	2	Yes	No	No
Sage	0.3	No	No	No
Salat Times	2.5	Yes	Yes	Yes
SandwichRoulette	1.5	Yes	Yes	Yes
Sanity	2.11	Yes	Yes	Yes
Santoral	2.0.3	No	No	No
SASAbus	0.2.8	Yes	Yes	Yes
sbautologin	2.1	No	No	No
Scanner For Zotero	1.0.1	Yes	No	No
Scid on the go	1.42	Yes	No	No
Scrambled Net	5.0.1	Yes	No	No
Screen Notifications	0.76	Yes	Yes	Yes
ScreenInfo	1.0.6	Yes	No	No
Scribbler	0.1.8	Yes	Yes	Yes
Scrobble Droid	1.0.6	Yes	Yes	Yes
Scrum Poker	1.0.0	Yes	No	No
Seafile	0.4.0	No	No	No
Search based launcher	1.13	Yes	Yes	Yes
Search Light	1.3	Yes	Yes	Yes
Seeks	1.2	No	No	No
Send Reduced	0.04	Yes	No	No
Send to Computer	1.5	No	No	No
Send to SD card	0.3.9	Yes	Yes	Yes
Send With FTP	1.1.0	Yes	Yes	Yes
Sensor Readout	1.2	No	No	No
Sequence Hunt	01.07.03	Yes	Yes	Yes
Serval Mesh	0.90.1	Yes	No	No
ServeStream	0.5.4	Yes	Yes	Yes
share my position	1.1.2	Yes	Yes	Yes
Share via HTTP	1.1	No	No	No
Shift	1.0.1	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Shortyz	3.2.1	Yes	Yes	Yes
Show Me Hills	0.3	Yes	Yes	Yes
SickStache	2.2.1	No	No	No
SidePanel	0.8.2(beta)	No	No	No
Simple C25K	0.2.4	Yes	Yes	Yes
Simple Chess Clock	1.2.0	Yes	Yes	Yes
Simple Coin Flip	5.3.2	Yes	No	No
Simple Deadlines	2.3.2	Yes	No	No
Simple Last.fm Scrobbler	1.4.4	Yes	Yes	Yes
Simple Loan Calculator	3.3.0	Yes	No	No
Simply Do	0.9.2	Yes	Yes	Yes
SIP Switch	1.0.4	Yes	No	No
Sipdroid	N/A	Yes	No	No
Sky Map	1.6.4	Yes	No	No
Slider	1.0.2	Yes	No	No
Slight backup	0.5.1	Yes	Yes	Yes
SLW Cpu Widget	1.5	No	No	No
SLW Traffic Meter Widget	1.1	No	No	No
Smoke Reducer	1	Yes	Yes	Yes
SMS Backup +	1.4.8	Yes	No	No
SMS Filter	1	Yes	Yes	Yes
SMS Popup	1.2.4	Yes	Yes	Yes
SMSdroid	1.4.4	Yes	Yes	Yes
SMSSync	2.0.2	Yes	Yes	Yes
Sokoban	1.11	Yes	Yes	Yes
Solitaire	1.12.2	Yes	Yes	Yes
Solitaire NG	1.12.1-ng3	Yes	No	No
Sonorox	1.0.1	Yes	Yes	Yes
Sound Manager	2.1.0	Yes	Yes	Yes
Sound Recorder	1	Yes	Yes	Yes
SparkleShare	1	Yes	Yes	Yes
Sparse rss	1.7	Yes	Yes	Yes
Speech Trainer	1.04	Yes	Yes	Yes
Speed of Sound	0.8.2	Yes	Yes	Yes
Speedo	1	Yes	Yes	Yes
Squeezer	0.9.1	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Std Atmosphere	1	Yes	Yes	Yes
Stickeroid	1.1.2	Yes	Yes	Yes
Subsonic	3.3	No	No	No
Sudoku Free	1.1	Yes	Yes	Yes
SuperGenPass	2.2.2	Yes	Yes	Yes
Superuser	1.0.1.7	Yes	No	No
SyncMyPix	0.15.2	Yes	No	No
TalkBack	3.2.1	Yes	No	No
TalkMyPhone	2.06-beta	Yes	No	No
Tallyphant	0.4	Yes	Yes	Yes
Taps Of Fire	1.0.5 (Droid)	Yes	Yes	Yes
Tea Timer	1.6	Yes	Yes	Yes
TeaCup	1.2	No	No	No
Ted	1.8.1	Yes	Yes	Yes
Terminal Emulator	1.0.52	No	No	No
Terminal IDE	2.02	Yes	No	No
Testnet3	3.01-test	No	No	No
Text Edit	1.5	No	No	No
TextWarrior	0.93	Yes	Yes	Yes
ThreeDLite	1	No	No	No
TicTacToe	1	Yes	Yes	Yes
Tilt Lander	2.1	Yes	No	No
TiltMazes	1.2	Yes	No	No
Timer	1.2	Yes	Yes	Yes
Timeriffic	1.09.05	Yes	Yes	Yes
Timesheet	1.4	Yes	Yes	Yes
Tint Browser	1.7	Yes	No	No
Tint Browser Adblock Addon	1.1	No	No	No
Tiny Open Source Violin	1.3	Yes	No	No
Tiny Tiny RSS	1.7	No	No	No
Tiny Tiny RSS Reader	1.46	No	No	No
Tippy Tipper	N/A	Yes	No	No
Toggle Headset 2	2.5	No	No	No
Tomdroid	0.7.2	Yes	Yes	Yes
Torch	0.2.3	Yes	No	No
TPT Helper	2.0.3	No	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Train Schedule	1.15	Yes	Yes	Yes
Tram Hunter	1.1	Yes	Yes	Yes
Transdroid Torrent Manager	1.1.13	No	No	No
Transdroid Torrent Search	1.1	No	No	No
Transports Bordeaux	2.6.8	Yes	No	No
Tricorder	5.11	No	No	No
Trolly	1.4	Yes	Yes	Yes
Trouve ton campus	1.0.1	Yes	Yes	Yes
Tryton	0.6	Yes	Yes	Yes
TTRSS-Reader	1.47	Yes	No	No
Tttris	0.9-beta	Yes	Yes	Yes
TUIOdroid	1.1	No	No	No
Tuner	1.02	Yes	No	No
TunesRemote+	2.5.3	No	No	No
TunesViewer	1.2.1	Yes	No	No
Tux Rider	1.0.9	Yes	No	No
TVHGuide	1.6.3	No	No	No
Twidere	0.2.1	No	No	No
Twidere Extension: TwitLonger	1.2	No	No	No
Twisty	0.82	Yes	No	No
Type and Speak	1.4.8	Yes	Yes	Yes
TZM	1.2.4	Yes	No	No
UberSync Facebook Contact Sync	1.2.1	No	No	No
Unicode Map	0.0.4	Yes	Yes	Yes
Units	1	Yes	Yes	Yes
Unix Epoch Converter	1.4	Yes	No	No
Urecord	1.4	Yes	Yes	Yes
URforms Database	1.14	Yes	No	No
USP - ZX Spectrum Emulator	0.0.46	Yes	No	No
Vanilla Music Player	0.1	Yes	No	No
Vanilla Player #1	0.9.10	Yes	Yes	Yes
Vanilla Player #2	0.9.18	Yes	No	No
Vector Pinball	1.3.1	Yes	No	No
Vespucci	0.8.3r419	Yes	Yes	Yes
Vi IMproved Touch	2.3	Yes	No	No
Vieilles Charrues 2010	1.12	Yes	No	No

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
VLC for Android	0.0.11-mips	Yes	No	No
Vlille Checker	2.2	Yes	Yes	Yes
Voice Notify	1.0.9.1	Yes	No	No
Voicesmith	2.3	Yes	Yes	Yes
Volume Control	1	Yes	No	No
Voodoo CarrierIQ Detector	2.0.4	Yes	No	No
Voodoo OTA RootKeeper	2.0.3	Yes	No	No
Voodoo Screen Test Patterns	3.3	No	No	No
VuDroid	1.4	Yes	Yes	Yes
VX ConnectBot	1.7.1-24	No	No	No
Wake On Lan	1.5	Yes	Yes	Yes
Weather notification	0.3.3	Yes	Yes	Yes
Weather Skin: Black	0.3.3	No	No	No
Weather Skin: White	0.3.3	No	No	No
WebSMS	4.5.4	Yes	Yes	Yes
WebSMS Connector: GMX	2	No	No	No
WebSMS Connector: smspilot.ru	1.4	No	No	No
weechat	0.8-dev-b5	No	No	No
Weight Chart	1	Yes	No	No
Whale shark and sardines	1.0.11	No	No	No
Wheelmap	0.8.3	Yes	Yes	Yes
WhereRing	1.99.0	Yes	No	No
Who Has My Stuff?	1.0.10	Yes	Yes	Yes
WiFi ACE	0.11	Yes	Yes	Yes
WiFi Barcode	1.2	Yes	No	No
Wifi Connecter Library	2.0.1	No	No	No
Wifi Fixer	0.9.5.6	Yes	Yes	Yes
WiFi Keyboard	2.3.3	Yes	No	No
WiFi Tether	3.2-beta2	Yes	No	No
Wi-Fi Widget	1	No	No	No
WiFiKeyboard	2.3.2	Yes	Yes	Yes
WiGLE Wifi Wardriving	1.55	Yes	Yes	Yes
WikiAndPad	0.2alpha03	Yes	Yes	Yes
Wikimedia Commons	1.0beta7	Yes	Yes	Yes
Wikipedia	1.3.4	No	No	No
Wikivoyage offline	1.3	Yes	Yes	Yes

Table 31(cont.)

Application	Version	Complex?	Source?	Can Run?
Wiktionary	1.0.1	No	No	No
Wireless Tether	2.0.7	Yes	No	No
WonderDroid	1.8c	No	No	No
Wordnik Dictionary	1.1	Yes	No	No
WordPress	2.3	No	No	No
World Clock	0.6	Yes	Yes	Yes
WorldMap	2.0.2	Yes	Yes	Yes
WWWJDIC for Android	2.2.5	Yes	No	No
X Server	1.14	Yes	Yes	Yes
Xabber	0.9.29a	No	Yes	Yes
XBMC Remote	1.0.9	No	No	No
XCSoar	6.4.5	Yes	No	No
xkcdViewer	4.1.0	Yes	Yes	Yes
Xmp for Android	3.5.0	Yes	Yes	No
YAAB	1.9.5	Yes	Yes	Yes
Yaaic	1	No	No	No
Yahtzee	1	Yes	Yes	Yes
yaxim	0.8.6b	No	No	No
Yin Yang	1.5	No	No	No
YouTube Downloader	1.9	Yes	No	No
YubiTOTP	0.0.2	No	No	No
YubNub Command Line	1.1.5	Yes	Yes	Yes
Zandy	1.3.6	Yes	Yes	Yes
Zirco Browser	0.4.4	Yes	Yes	Yes
ZooBorns	1.4.4	No	No	No
ZXing Test	1.2.1	No	No	No
四次元	0.483	No	No	No
注音倉頡輸入法	1	No	No	No
	749	**565**	**326**	**323**

Appendix B – Pre-test Results

Table 33. Pre-test Reliability Results

Application Package (APK)	OOM	NPE	BTE	OTH	TOT	Events	d/kE
aarddict.android_21.apk	0	0	0	0	0	20000	0.000
acr.browser.barebones_42.apk	0	0	0	0	0	20000	0.000
An.stop_10.apk	0	0	0	0	0	20000	0.000
android.androidVNC_13.apk	1	1	0	0	2	19870	0.101
apps.babycaretimer_5.apk	0	0	0	1	1	19912	0.050
apps.droidnotify_67.apk	0	0	0	0	0	20000	0.000
arity.calculator_27.apk	0	0	0	0	0	20000	0.000
au.com.darkside.XServer_14.apk	0	0	0	0	0	20000	0.000
aws.apps.androidDrawables_8.apk	2	0	0	12	14	18821	0.744
bander.notepad_12.apk	0	0	0	0	0	18805	0.000
be.ppareit.swiftp_free_27.apk	0	0	0	20	20	19345	1.034
biz.gyrus.yaab_19.apk	1	0	0	0	1	19949	0.050
budo.budoist_33.apk	1	0	0	0	1	19995	0.050
bughunter2.smsfilter_1.apk	0	0	0	0	0	17734	0.000
caldwell.ben.bites_4.apk	0	0	0	0	0	20000	0.000
caldwell.ben.trolly_6.apk	0	0	0	0	0	20000	0.000
cc.co.eurdev.urecorder_5.apk	1	0	0	0	1	19982	0.050
cgeo.geocaching_20130605.apk	28	0	0	1	29	15977	1.815
ch.fixme.cowsay_4.apk	0	0	0	0	0	20000	0.000
com.adam.aslfms_31.apk	1	0	0	0	1	17553	0.057
com.addi_44.apk	0	0	0	0	0	20000	0.000
com.alfray.asqare_103.apk	1	0	0	0	1	19958	0.050
com.alfray.timeriffic_10905.apk	0	0	0	0	0	20000	0.000
com.amphoras.tpthelper_24.apk	1	0	0	0	1	19949	0.050
com.android.keepass_125.apk	0	0	0	0	0	20000	0.000
com.androidemu.gbc_32.apk	0	0	0	0	0	20000	0.000
com.androidemu.nes_61.apk	0	0	0	0	0	20000	0.000
com.androzic_90.apk	20	0	0	0	20	18368	1.089
com.andybotting.tramhunter_1200.apk	0	0	0	0	0	20000	0.000
com.angrydoughnuts.android.alarmclock_8.apk	1	0	0	0	1	19974	0.050
com.anoshenko.android.mahjongg_14.apk	40	0	0	0	40	17649	2.266
com.app2go.sudokufree_3.apk	1	0	0	0	1	19989	0.050
com.appengine.paranoid_android.lost_12.apk	0	0	0	0	0	20000	0.000
com.aripuca.tracker_24.apk	18	0	0	0	18	18742	0.960
com.artifex.mupdfdemo_52.apk	0	0	0	0	0	20000	0.000
com.aselalee.trainschedule_116.apk	0	0	0	0	0	20000	0.000
com.axelby.podax_38.apk	1	0	0	0	1	18769	0.053
com.banasiak.coinflip_30.apk	21	0	0	0	21	18510	1.135
com.beem.project.beem_15.apk	0	0	0	0	0	20000	0.000
com.boardgamegeek_27.apk	0	0	0	0	0	20000	0.000
com.boombuler.games.shift_101.apk	0	1	0	0	1	19953	0.050
com.brocktice.JustSit_17.apk	0	0	0	0	0	20000	0.000
com.brosmike.airpushdetector_11.apk	0	0	0	0	0	18802	0.000

Table 32(cont.)

Application Package (APK)	OOM	NPE	BTE	OTH	TOT	Events	d/kE
com.bwx.bequick_201107260.apk	0	0	0	0	0	20000	0.000
com.byagowi.persiancalendar_38.apk	0	0	0	0	0	18605	0.000
com.cepmuvakkit.times_200.apk	0	0	0	0	0	20000	0.000
com.chessclock.android_8.apk	0	0	0	0	0	20000	0.000
com.ciarang.tallyphant_5.apk	0	0	0	0	0	20000	0.000
com.commonsware.android.arXiv_106.apk	1	0	0	1	2	19935	0.100
com.dozingcatsoftware.asciicam_5.apk	31	1	0	0	32	18041	1.774
com.dozingcatsoftware.dodge_8.apk	0	0	0	0	0	20000	0.000
com.dozuki.ifixit_14.apk	15	0	0	0	15	19266	0.779
com.dririan.RingyDingyDingy_705.apk	1	0	0	0	1	19973	0.050
com.drismo_17.apk	1	0	0	0	1	18685	0.054
com.eleybourn.bookcatalogue_151.apk	1	0	0	0	1	19959	0.050
com.eolwral.osmonitor_42.apk	0	0	0	1	1	19992	0.050
com.episode6.android.appalarm.pro_31.apk	0	0	0	0	0	19905	0.000
com.evancharlton.mileage_3110.apk	3	0	0	1	4	19723	0.203
com.everysoft.autoanswer_6.apk	1	0	0	0	1	19977	0.050
com.fivasim.antikythera_5.apk	4	0	0	0	4	17496	0.229
com.frostwire.android_85.apk	0	0	0	0	0	20000	0.000
com.gcstar.scanner_1.apk	2	0	0	4	6	19571	0.307
com.gcstar.viewer_10.apk	0	40	0	0	40	17788	2.249
com.ghostsq.commander_210.apk	4	0	0	0	4	17515	0.228
com.gimranov.zandy.app_1370.apk	0	72	0	1	73	17227	4.238
com.github.mobile_1300.apk	0	0	0	0	0	20000	0.000
com.gladis.tictactoe_1.apk	0	0	0	0	0	20000	0.000
com.glTron_4.apk	19	3	0	0	22	15739	1.398
com.gmail.altakey.effy_9.apk	0	0	0	0	0	20000	0.000
com.googamaphone.typeandspeak_36.apk	0	0	0	0	0	20000	0.000
com.google.android.apps.authenticator2_21.apk	0	0	0	0	0	20000	0.000
com.google.android.diskusage_3040.apk	10	0	0	0	10	19321	0.518
com.google.code.appsorganizer_167.apk	1	0	0	0	1	17430	0.057
com.googlecode.awsms_24.apk	1	0	0	0	1	18782	0.053
com.googlecode.gtalksms_66.apk	0	0	0	47	47	17555	2.677
com.googlecode.networklog_21800.apk	1	2	0	0	3	19773	0.152
com.gueei.applocker_3.apk	0	0	0	0	0	19000	0.000
com.hectorone.multismssender_13.apk	1	0	0	0	1	19921	0.050
com.hughes.android.dictionary_23.apk	28	0	0	0	28	17653	1.586
com.ideasfrombrain.search_based_launcher_v2_5.apk	0	0	0	22	22	16230	1.356
com.ihunda.android.binauralbeat_39.apk	0	0	0	0	0	20000	0.000
com.integralblue.callerid_7.apk	0	0	0	0	0	20000	0.000
com.irahul.worldclock_2.apk	0	0	0	0	0	17705	0.000
com.jadn.cc_143.apk	0	0	0	9	9	19502	0.461
com.java.SmokeReducer_1.apk	0	0	0	27	27	18605	1.451
com.jeyries.quake2_21.apk	0	0	0	0	0	20000	0.000

Table 32(cont.)

Application Package (APK)	OOM	NPE	BTE	OTH	TOT	Events	d/kE
com.jlyr_36.apk	0	1	0	0	1	16432	0.061
com.kai1973i_4.apk	0	0	0	0	0	20000	0.000
com.kkinder.charmap_101.apk	0	0	0	0	0	20000	0.000
com.kmagic.solitaire_450.apk	0	0	0	0	0	20000	0.000
com.kpz.pomodorotasks.activity_8.apk	1	0	0	0	1	19916	0.050
com.kvance.Nectroid_11.apk	0	0	0	0	0	15125	0.000
com.leafdigital.kanji.android_2.apk	11	0	0	0	11	19405	0.567
com.leinardi.kitchentimer_116.apk	0	0	0	0	0	20000	0.000
com.lukekorth.screennotifications_14.apk	1	0	0	0	1	15664	0.064
com.madgag.agit_130400912.apk	1	0	0	1	2	19937	0.100
com.manuelmaly.hn_11.apk	0	0	0	0	0	20000	0.000
com.matburt.mobileorg_98.apk	0	0	0	0	0	20000	0.000
com.maxfierke.sandwichroulette_2.apk	0	0	0	0	0	20000	0.000
com.mendhak.gpslogger_27.apk	0	0	0	0	0	20000	0.000
com.midisheetmusic_8.apk	43	0	0	0	43	18924	2.272
com.miracleas.bitcoin_spinner_37.apk	0	0	0	0	0	20000	0.000
com.mkf.droidsat_24.apk	1	0	0	0	1	17372	0.058
com.mobilepearls.sokoban_12.apk	0	0	0	0	0	16000	0.000
com.monead.games.android.sequence_21.apk	0	0	0	3	3	19791	0.152
com.morphoss.acal_60.apk	0	0	0	0	0	20000	0.000
com.myopicmobile.textwarrior.android_13.apk	0	0	0	0	0	20000	0.000
com.naholyr.android.horairessncf_301.apk	0	0	0	0	0	20000	0.000
com.namelessdev.mpdroid_33.apk	0	1	0	9	10	19278	0.519
com.nanoconverter.zlab_38.apk	0	2	0	0	2	19839	0.101
com.nauj27.android.colorpicker_20121130.apk	1	0	0	0	1	19928	0.050
com.netthreads.android.noiz2_12.apk	3	0	0	2	5	14821	0.337
com.newsblur_38.apk	1	1	0	0	2	19886	0.101
com.nexes.manager_218.apk	1	0	0	0	1	19945	0.050
com.nolanlawson.apptracker_10.apk	0	0	0	0	0	20000	0.000
com.nolanlawson.chordreader_8.apk	0	0	0	0	0	20000	0.000
com.nolanlawson.jnameconverter_1.apk	0	0	0	0	0	19100	0.000
com.nolanlawson.keepscore_20.apk	0	0	0	10	10	19093	0.524
com.opendoorstudios.ds4droid_40.apk	0	0	0	0	0	20000	0.000
com.owncloud.android_104004.apk	0	0	0	0	0	20000	0.000
com.palliser.nztides_5.apk	0	0	0	0	0	20000	0.000
com.pindroid_56.apk	1	0	0	0	1	19919	0.050
com.piwi.stickeroid_6.apk	0	0	0	0	0	20000	0.000
com.politedroid_5.apk	0	0	0	0	0	20000	0.000
com.purplefoto.pfdock_3.apk	0	0	0	0	0	20000	0.000
com.qubling.sidekick_16.apk	0	0	0	0	0	20000	0.000
com.replica.replicaisland_14.apk	0	0	0	0	0	20000	0.000
com.rhiannonweb.android.migrainetracker_2.apk	0	0	0	0	0	20000	0.000
com.rigid.birthdroid_2.apk	0	0	0	0	0	19000	0.000

Table 32(cont.)

Application Package (APK)	OOM	NPE	BTE	OTH	TOT	Events	d/kE
com.rj.pixelesque_7.apk	80	0	0	0	80	13480	5.935
com.robert.maps_8140.apk	0	0	0	0	0	20000	0.000
com.roozen.SoundManagerv2_19.apk	0	0	0	0	0	20000	0.000
com.scottmain.android.searchlight_4.apk	0	0	0	21	21	18281	1.149
com.seavenois.tetris_3.apk	0	0	0	20	20	12486	1.602
com.showmehills_30.apk	0	0	0	0	0	19202	0.000
com.sigseg.android.worldmap_5.apk	0	0	0	0	0	20000	0.000
com.smerty.ham_18.apk	0	0	0	43	43	16629	2.586
com.smorgasbork.hotdeath_8.apk	18	3	0	0	21	18904	1.111
com.sweetiepiggy.everylocale_9.apk	0	0	0	122	122	12296	9.922
com.tastycactus.timesheet_6.apk	0	0	0	0	0	20000	0.000
com.teamdc.stephendiniz.autoaway_23.apk	1	0	0	53	54	15652	3.450
com.teleca.jamendo_38.apk	0	0	0	0	0	16802	0.000
com.tmarki.comicmaker_33.apk	0	0	0	4	4	19739	0.203
com.totsp.bookworm_19.apk	1	0	0	0	1	19968	0.050
com.totsp.crossword.shortyz_30201.apk	0	0	0	0	0	20000	0.000
com.tum.yahtzee_1.apk	1	0	0	0	1	19981	0.050
com.valleytg.oasvn.android_11.apk	0	0	0	0	0	16500	0.000
com.veken0m.bitcoinium_33.apk	2	0	12	3	17	19180	0.886
com.vlille.checker_5.apk	0	0	0	0	0	20000	0.000
com.volosyukivan_30.apk	0	0	0	0	0	20000	0.000
com.wanghaus.remembeer_50.apk	0	0	0	0	0	20000	0.000
com.xabber.androiddev_78.apk	1	0	0	0	1	19989	0.050
com.xatik.app.droiddraw.client_4.apk	0	0	0	0	0	17870	0.000
com.zachrattner.pockettalk_7.apk	0	0	0	0	0	20000	0.000
com.zapta.apps.maniana_26015.apk	1	0	0	0	1	19974	0.050
com.zoffcc.applications.aagtl_31.apk	3	0	0	0	3	18389	0.163
cri.sanity_21100.apk	0	0	0	0	0	20000	0.000
cx.hell.android.pdfview_40000.apk	0	0	0	0	0	20000	0.000
cz.hejl.chesswalk_8.apk	0	2	0	9	11	19829	0.555
cz.romario.opensudoku_1105.apk	0	0	0	0	0	20000	0.000
de.antonfluegge.android.yubnubwidgetadfree_7.apk	19	14	0	0	33	16319	2.022
de.arnowelzel.android.periodical_13.apk	0	0	0	0	0	20000	0.000
de.blau.android_19.apk	2	0	0	0	2	19386	0.103
de.danoeh.antennapod_31.apk	0	0	0	0	0	20000	0.000
de.duenndns.gmdice_6.apk	0	0	0	0	0	20000	0.000
de.freewarepoint.whohasmystuff_11.apk	0	0	0	0	0	20000	0.000
de.jurihock.voicesmith_9.apk	0	0	0	0	0	20000	0.000
de.mangelow.network_3.apk	0	0	0	0	0	19000	0.000
de.mbutscher.wikiandpad.alphabeta_200300.apk	0	30	0	0	30	18910	1.586
de.schaeuffelhut.android.openvpn_39.apk	0	0	0	0	0	20000	0.000
de.shandschuh.slightbackup_18.apk	1	0	0	0	1	19924	0.050
de.shandschuh.sparserss_87.apk	0	0	0	0	0	20000	0.000

Table 32(cont.)

Application Package (APK)	OOM	NPE	BTE	OTH	TOT	Events	d/kE
de.skubware.opentraining_11.apk	9	2	0	0	11	18163	0.606
de.stefan_oltmann.falling_blocks_8.apk	0	0	0	0	0	20000	0.000
de.stefan_oltmann.kaesekaestchen_7.apk	0	0	0	0	0	15300	0.000
de.tui.itlogger_2.apk	0	0	0	35	35	17454	2.005
de.ub0r.android.smsdroid_7144000.apk	0	0	0	0	0	20000	0.000
de.ub0r.android.websms_7454000.apk	0	0	0	0	0	20000	0.000
de.wikilab.android.friendica01_9.apk	0	0	0	17	17	18824	0.903
dev.drsoran.moloko_94211.apk	0	0	0	0	0	20000	0.000
dk.andsen.asqlitemanager_17.apk	0	0	1	9	10	19847	0.504
edu.killerud.kitchentimer_4.apk	0	0	0	2	2	19920	0.100
edu.nyu.cs.omnidroid.app_6.apk	0	0	0	8	8	19796	0.404
edu.sfsu.cs.orange.ocr_31.apk	0	0	0	1	1	19105	0.052
es.cesar.quitesleep_13.apk	0	0	0	0	0	20000	0.000
eu.domob.angulo_10200.apk	0	0	0	0	0	20000	0.000
eu.domob.bjtrainer_100.apk	0	0	0	0	0	20000	0.000
eu.hydrologis.geopaparazzi_39.apk	19	0	0	0	19	17995	1.056
eu.lavarde.pmtd_2013050413.apk	3	0	0	0	3	19863	0.151
eu.prismsw.lampshade_117.apk	0	0	0	0	0	20000	0.000
fm.libre.droid_4.apk	0	0	0	0	0	20000	0.000
fr.bellev.stdatmosphere_1.apk	1	0	0	0	1	19913	0.050
fr.keuse.rightsalert_3.apk	0	1	0	0	1	19923	0.050
fr.strasweb.campus_4.apk	0	0	1	0	1	19942	0.050
fr.xgouchet.texteditor_19.apk	0	0	0	0	0	20000	0.000
goo.TeaTimer_9.apk	0	0	0	0	0	20000	0.000
hsware.HSTempo_17.apk	0	0	0	0	0	20000	0.000
hu.vsza.adsdroid_3.apk	0	43	0	0	43	18216	2.361
i4nc4mp.myLock_28.apk	0	0	0	0	0	20000	0.000
in.shick.lockpatterngenerator_7.apk	1	0	0	0	1	19957	0.050
info.staticfree.android.robotfindskitten_7.apk	2	0	0	0	2	19806	0.101
info.staticfree.android.twentyfourhour_8.apk	11	0	0	0	11	19060	0.577
info.staticfree.android.units_9.apk	0	0	0	0	0	20000	0.000
info.staticfree.SuperGenPass_20.apk	0	0	0	0	0	20000	0.000
it.iiizio.epubator_12.apk	1	0	0	0	1	19979	0.050
it.sasabz.android.sasabus_24.apk	0	3	6	20	29	18607	1.559
kaljurand_at_gmail_dot_com.diktofon_980.apk	0	0	0	0	0	20000	0.000
kdk.android.simplydo_2.apk	0	0	0	0	0	20000	0.000
me.guillaumin.android.osmtracker_28.apk	0	0	0	0	0	20000	0.000
mixedbit.speechtrainer_5.apk	0	0	0	0	0	20000	0.000
mobi.cyann.nstools_20.apk	0	0	0	0	0	20000	0.000
mohammad.adib.roundr_24.apk	0	0	0	0	0	20000	0.000
name.bagi.levente.pedometer_6.apk	0	0	0	0	0	20000	0.000
net.androgames.level_33.apk	0	0	0	0	0	20000	0.000
net.bible.android.activity_107.apk	0	0	0	0	0	20000	0.000

Table 32(cont.)

Application Package (APK)	OOM	NPE	BTE	OTH	TOT	Events	d/kE
net.bytten.xkcdviewer_32.apk	0	0	0	0	0	20000	0.000
net.cactii.mathdoku_281.apk	0	0	0	0	0	20000	0.000
net.codechunk.speedofsound_9.apk	0	0	0	69	69	15782	4.372
net.cyclestreets_11.apk	0	0	0	0	0	18800	0.000
net.everythingandroid.smspopup_124.apk	0	0	0	0	0	20000	0.000
net.fercanet.LNM_5.apk	0	0	0	0	0	20000	0.000
net.gorry.aicia_201212241.apk	0	0	0	0	0	20000	0.000
net.gorry.android.input.nicownng_201304251.apk	0	0	0	0	0	20000	0.000
net.healeys.lexic_41.apk	0	0	0	0	0	20000	0.000
net.jjc1138.android.scrobbler_7.apk	0	0	0	0	0	20000	0.000
net.kervala.comicsreader_17.apk	0	0	0	0	0	20000	0.000
net.logomancy.diedroid_9.apk	0	0	0	0	0	20000	0.000
net.mafro.android.wakeonlan_13.apk	0	0	0	0	0	20000	0.000
net.micode.compass_1.apk	0	0	0	0	0	20000	0.000
net.micode.soundrecorder_1.apk	0	0	0	0	0	20000	0.000
net.nightwhistler.pageturner_25.apk	0	0	0	6	6	19783	0.303
net.osmand.plus_145.apk	43	0	0	0	43	18658	2.305
net.pierrox.mcompass_10.apk	0	0	0	0	0	20000	0.000
net.pmarks.chromadoze_10.apk	0	0	0	0	0	20000	0.000
net.progval.android.andquote_7.apk	0	0	0	0	0	20000	0.000
net.sf.andbatdog.batterydog_11.apk	0	0	0	1	1	19973	0.050
net.sf.andhsli.hotspotlogin_20.apk	0	0	0	0	0	20000	0.000
net.sourceforge.andsys_30.apk	0	0	0	0	0	20000	0.000
net.sourceforge.servestream_72.apk	0	0	0	0	0	20000	0.000
net.sylvek.sharemyposition_24.apk	0	0	0	0	0	20000	0.000
net.szym.barnacle_39.apk	0	0	0	0	0	20000	0.000
net.vivekiyer.GAL_24.apk	0	0	0	0	0	20000	0.000
net.wigle.wigleandroid_55.apk	1	0	0	0	1	19877	0.050
nl.ttys0.simplec25k_8.apk	0	0	0	0	0	20000	0.000
nu.firetech.android.pactrack_1310.apk	0	0	0	7	7	18147	0.386
org.addhen.smssync_15.apk	0	0	0	0	0	20000	0.000
org.ametro_17.apk	0	0	0	0	0	20000	0.000
org.androidsoft.coloring_1.apk	0	0	0	0	0	20000	0.000
org.andstatus.app_77.apk	0	0	0	0	0	20000	0.000
org.balau.fakedawn_3.apk	0	0	0	0	0	20000	0.000
org.blockinger.game_13.apk	0	0	0	0	0	20000	0.000
org.brandroid.openmanager_212.apk	0	0	0	0	0	20000	0.000
org.coolreader_847.apk	0	0	0	0	0	20000	0.000
org.cry.otp_20.apk	0	0	0	11	11	19866	0.554
org.damazio.notifier_11.apk	0	0	0	0	0	20000	0.000
org.dnaq.dialer2_17.apk	0	0	0	0	0	20000	0.000
org.dpadgett.timer_2.apk	12	0	0	0	12	19058	0.630
org.droidseries_13.apk	0	0	0	0	0	20000	0.000

Table 32(cont.)

Application Package (APK)	OOM	NPE	BTE	OTH	TOT	Events	d/kE
org.dsandler.apps.markers_40.apk	1	0	0	0	1	19938	0.050
org.example.pushupbuddy_1.apk	0	0	0	0	0	20000	0.000
org.fastergps_7.apk	0	0	0	0	0	20000	0.000
org.fdroid.fdroid_47.apk	0	0	0	0	0	20000	0.000
org.gc.networktester_2.apk	0	0	0	0	0	20000	0.000
org.github.OxygenGuide_4.apk	0	0	0	0	0	20000	0.000
org.gmote.client.android_5.apk	0	0	0	3	3	19757	0.152
org.gnucash.android_10.apk	0	0	0	2	2	19909	0.100
org.herrlado.geofonts_41.apk	0	0	0	0	0	20000	0.000
org.jessies.dalvikexplorer_36.apk	0	4	0	0	4	19798	0.202
org.jessies.mathdroid_293.apk	0	0	0	0	0	20000	0.000
org.jmoyer.NotificationPlus_2.apk	0	0	0	0	0	20000	0.000
org.jtb.alogcat_43.apk	0	0	0	0	0	20000	0.000
org.kontalk_24.apk	1	0	0	0	1	18846	0.053
org.kreed.vanilla_910.apk	0	0	0	3	3	19751	0.152
org.ligi.fast_37.apk	0	0	0	0	0	20000	0.000
org.linphone_2120.apk	85	0	0	13	98	11811	8.297
org.madore.android.unicodeMap_4.apk	0	0	0	1	1	19902	0.050
org.marcus905.wifi.ace_20120115.apk	0	0	0	0	0	20000	0.000
org.ncrmnt.nettts_3.apk	0	0	0	36	36	17633	2.042
org.nerdcircus.android.klaxon_27.apk	0	0	0	0	0	20000	0.000
org.odk.collect.android_1023.apk	0	0	0	0	0	20000	0.000
org.opengpx_192.apk	0	0	0	0	0	20000	0.000
org.openintents.filemanager_26.apk	11	0	0	0	11	15760	0.698
org.passwordmaker.android_7.apk	0	0	0	0	0	20000	0.000
org.pocketworkstation.pckeyboard_1033.apk	0	0	0	0	0	20000	0.000
org.pulpdust.lesserpad_10.apk	0	0	0	0	0	20000	0.000
org.pyload.android.client_17.apk	2	0	0	0	2	16004	0.125
org.scoutant.blokish_13.apk	0	0	0	0	0	20000	0.000
org.sixgun.ponyexpress_12.apk	23	0	0	5	28	15332	1.826
org.sparkleshare.android_1.apk	0	0	0	7	7	14489	0.483
org.tof_17.apk	0	0	0	0	0	20000	0.000
org.tomdroid_11.apk	0	0	1	0	1	19994	0.050
org.torproject.android_51.apk	0	0	0	0	0	20000	0.000
org.totschnig.myexpenses_55.apk	0	0	0	0	0	20000	0.000
org.totschnig.sendwithftp_8.apk	0	0	0	0	0	20000	0.000
org.tryton.client_6.apk	6	0	0	0	6	18424	0.326
org.vono.narau_6.apk	0	29	0	0	29	18820	1.541
org.vudroid_5.apk	0	0	0	0	0	20000	0.000
org.wahtod.wififixer_983.apk	0	0	0	0	0	20000	0.000
org.wheelmap.android.online_16.apk	0	0	0	0	0	20000	0.000
org.wikimedia.commons_9.apk	0	0	0	0	0	20000	0.000
org.woltage.irssiconnectbot_393.apk	0	0	0	0	0	20000	0.000

Table 32(cont.)

Application Package (APK)	OOM	NPE	BTE	OTH	TOT	Events	d/kE
org.yuttadhammo.BodhiTimer_45.apk	0	0	0	0	0	20000	0.000
org.zakky.memopad_3.apk	11	0	0	0	11	18998	0.579
org.zeroxlab.zeroxbenchmark_9.apk	0	0	0	0	0	20000	0.000
org.zirco_18.apk	0	0	0	0	0	20000	0.000
pl.magot.vetch.ancal_32.apk	0	0	0	0	0	20000	0.000
pro.dbro.bart_11.apk	0	0	0	0	0	20000	0.000
pt.isec.tp.am_4.apk	22	6	0	0	28	9789	2.860
ru.gelin.android.sendtosd_36.apk	0	0	0	0	0	20000	0.000
ru.gelin.android.weather.notification_39.apk	0	0	0	0	0	20000	0.000
se.johanhil.duckduckgo_1.apk	0	0	0	0	0	20000	0.000
sk.madzik.android.logcatudp_5.apk	0	0	0	17	17	19253	0.883
sonoroxadc.garethmurfin.co.uk_4.apk	0	0	0	6	6	19755	0.304
SpeedoMeterApp.main_1.apk	0	0	0	0	0	20000	0.000
tkj.android.homecontrol.mythmote_2308.apk	0	85	0	0	85	16470	5.161
tritop.android.androsens_2.apk	0	0	0	0	0	20000	0.000
uk.ac.cam.cl.dtg.android.barcodebox_3.apk	0	0	0	58	58	16563	3.502
uk.org.ngo.squeezer_13.apk	0	0	0	2	2	19906	0.100
urbanstew.RehearsalAssistant_22.apk	0	0	0	0	0	19900	0.000
us.lindanrandy.cidrcalculator_118.apk	0	0	0	0	0	20000	0.000
vnd.bluerarat.kaleidoscope6_18.apk	0	0	0	0	0	20000	0.000
vu.de.urpool.quickdroid_49.apk	0	0	0	0	0	19100	0.000
zame.GloomyDungeons.opensource.game _1358421967.apk	0	0	0	56	56	17152	3.265
Totals	**724**	**347**	**21**	**842**	**1934**	**6232213**	**0.310**
Number of APKs	**77**	**23**	**5**	**51**	**129**	-	-

Appendix C – Data Collection Instruments

C1 – Example Inspection Journal (DOC)

1 PURPOSE

The purpose of this journal is to record the investigations related to the cgeo.geocaching_20130605.apk. This APK exhibited 28 OOM errors.

2 JOURNAL

Date	*Entry*
9/6	Scanning cgeo.java cold. Only knowing that 28 OOM errors occurred. Nothing jumps out. If I were to guess it was the rebuilding the views onCreate every time. Effort: Minimal.
9/12	Rescanning for patterns. No null checks on intent (cgeo.java line 180). Some null checks on resources (cgeo.java line 195) nothing else jumping out.
9/30	Inspecting line 93 of AbstractFileListActivity.java. No checking on values or key check. Inspecting line 380 of CGeoMap.java. No keycheck or on values. Inspecting line 139 of AbstractPopupActivity.java. No key check or value check. Inspecting line 214 of CacheDetailActivity.java. Minimal guarding. Inspecting line 426 of cgeocaches.java. No check on keys or values. Inspecting line 43 of SimpleDirChooser.java. No guarding. Inspecting line 70 of cgeonavigate.java. Some guarding on the bundle. Inspecting line 120 of EditWaypointActivity.java. Some guarding on the bundle object. Inspecting line 35 of ImagesActivity.java. Some guarding. Inspecting line 63 of ImageSelectActivity.java. Some guarding. Inspecting line 119 of LogTrackableActivity.java. Some guarding. Inspecting line 97 of StaticMapsActivity.java. Some guarding. Inspecting line 123 of TrackableActivity.java. Some guarding. ("'Some guarding' generally means null check on the object or on the values, but not both.") Inspecting line 283 of VisitCacheActivity.java. Some guarding. Inspecting line 30 of WaypointPopup.java. Some guarding.

C2 – NullPointerException Journal Excerpt (XLS)

	Package	File	Line No.	NPE Smell	COR Pattern	ENH Pattern	Comment
2	An.stop_10	)_src\trunk\src\An\stop\ShowTimesActivity.java	63	N	N		
3	android.androidVNC_13	src\android\androidVNC\VncCanvasActivity.java	585	Y	N		
4	apps.babycaretimer_5	ps\babycaretimer\receivers\AlarmReceiver.java	38	N	N		
5	apps.babycaretimer_5	er\services\AlarmBroadcastReceiverService.java	56	Y	N		
6	apps.babycaretimer_5	er\services\AlarmBroadcastReceiverService.java	79	N	N		
7	apps.babycaretimer_5	ycaretimer\services\AlarmReceiverService.java	121	N	N		
8	apps.babycaretimer_5	_src\src\apps\babycaretimer\AlarmActivity.java	97	Y	N		
9	apps.babycaretimer_5	c\src\apps\babycaretimer\SetAlarmActivity.java	74	Y	N		
10	apps.droidnotify_67	rc\src\apps\droidnotify\common\Common.java	1723	N	N		
11	apps.droidnotify_67	fy\preferences\UpgradePreferenceActivity.java	46	Y	N		
12	apps.droidnotify_67	ceivers\CalendarNotificationAlarmReceiver.java	54	N	N		
13	apps.droidnotify_67	tify\receivers\GenericNotificationReceiver.java	49	N	N		
14	apps.droidnotify_67	ps\droidnotify\receivers\K9AlarmReceiver.java	38	N	N		
15	apps.droidnotify_67	src\apps\droidnotify\receivers\K9Receiver.java	54	N	N		
16	apps.droidnotify_67	droidnotify\receivers\RescheduleReceiver.java	51	N	N		
17	apps.droidnotify_67	ceivers\ScreenManagementAlarmReceiver.java	40	N	N		
18	apps.droidnotify_67	rc\apps\droidnotify\receivers\SMSReceiver.java	54	N	N		
19	apps.droidnotify_67	NotificationAlarmBroadcastReceiverService.java	51	Y	N		
20	apps.droidnotify_67	NotificationAlarmBroadcastReceiverService.java	87	N	N		
21	apps.droidnotify_67	apps\droidnotify\services\CalendarService.java	39	N	N		
22	apps.droidnotify_67	lnotify\services\GenericNotificationService.java	57	N	N		
23	apps.droidnotify_67	services\K9AlarmBroadcastReceiverService.java	47	Y	N		
24	apps.droidnotify_67	services\K9AlarmBroadcastReceiverService.java	67	N	N		
25	apps.droidnotify_67	notify\services\K9BroadcastReceiverService.java	50	N	N		
26	apps.droidnotify_67	rc\src\apps\droidnotify\services\K9Service.java	42	N	N		
27	apps.droidnotify_67	ps\droidnotify\services\RescheduleService.java	48	Y	N		
28	apps.droidnotify_67	s\droidnotify\services\SMSReceiverService.java	52	Y	N		
29	apps.droidnotify_67	s\droidnotify\services\SMSReceiverService.java	72	N	N		
30	apps.droidnotify_67	\src\apps\droidnotify\services\SMSService.java	42	Y	N		
31	apps.droidnotify_67	rc\src\apps\droidnotify\NotificationActivity.java	869	Y	N		
32	apps.droidnotify_67	rc\src\apps\droidnotify\NotificationActivity.java	1069	Y	N		
33	aws.apps.androidDrawables	oidDrawables\services\ExportIntentService.java	82	N	N		

C3 – OutOfMemoryError Journal Excerpt (XLS)

	Package	File	Line	Smell	COR Pattern	ENH Pattern	Comment
2	aarddict.android_21	oid_21_src\src\aarddict\android\ArticleViewActivity.java	628	Y	N	N	
3	aarddict.android_21	oid_21_src\src\aarddict\android\ArticleViewActivity.java	671	Y	N	N	
4	aarddict.android_21	oid_21_src\src\aarddict\android\ArticleViewActivity.java	703	Y	N	N	
5	aarddict.android_21	oid_21_src\src\aarddict\android\ArticleViewActivity.java	736	Y	N	N	
6	aarddict.android_21	oid_21_src\src\aarddict\android\ArticleViewActivity.java	761	Y	N	N	
7	aarddict.android_21	oid_21_src\src\aarddict\android\ArticleViewActivity.java	776	Y	N	N	
8	aarddict.android_21	ndroid_21_src\src\aarddict\android\LookupActivity.java	85	Y	N	N	
9	aarddict.android_21	ndroid_21_src\src\aarddict\android\LookupActivity.java	109	Y	N	N	
10	aarddict.android_21	ndroid_21_src\src\aarddict\android\LookupActivity.java	209	Y	N	N	
11	aarddict.android_21	ndroid_21_src\src\aarddict\android\LookupActivity.java	212	Y	N	N	
12	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	314	Y	N	N	
13	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	809	Y	N	N	
14	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	1288	Y	N	N	@Suppress
15	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	1312	Y	N	N	
16	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	1540	Y	N	N	
17	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	1698	Y	N	N	
18	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	2144	Y	N	N	
19	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	2401	Y	N	N	
20	acr.browser.barebones_42	ones_42_src\src\acr\browser\barebones\Barebones.java	2433	Y	N	N	
21	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	64	Y	N	N	
22	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	82	Y	N	N	
23	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	275	Y	N	N	
24	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	610	Y	N	N	
25	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	771	Y	N	N	
26	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	793	Y	N	N	
27	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	796	Y	N	N	
28	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	1106	Y	N	N	
29	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	1117	Y	N	N	
30	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	1132	Y	N	N	
31	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	1264	Y	N	N	
32	android.androidVNC_13	oidVNC\src\android\androidVNC\VncCanvasActivity.java	1345	Y	N	N	

C4 – BadTokenException Journal Excerpt (XLS)

	Package	File	Line	NIF Smell	GAC Smell	COR Pattern	ENH Pattern	Comment
2	aarddict.android_21	c\aarddict\android\ArticleViewActivity.java	796	Y	-	-	-	CTC
3	aarddict.android_21	rc\src\aarddict\android\LookupActivity.java	312	Y	N	N	N	NIF
4	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	526	Y	Y	N	N	NIF & GAC
5	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	870	Y	Y	N	N	NIF & GAC
6	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	915	Y	Y	N	N	NIF & GAC
7	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	2064	Y	Y	N	N	NIF & GAC
8	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	2300	Y	-	N	N	CTC
9	acr.browser.barebones_42	rc\src\acr\browser\barebones\Settings.java	363	Y	N	N	N	NIF
10	android.androidVNC_13	ndroid\androidVNC\VncCanvasActivity.java	1037	Y	-	N	N	CTC
11	android.androidVNC_13	ndroid\androidVNC\VncCanvasActivity.java	1110	Y	-	N	N	CTC
12	apps.babycaretimer_5	er\preferences\MainPreferenceActivity.java	283	Y	N	N	N	NIF
13	apps.babycaretimer_5	er\preferences\MainPreferenceActivity.java	479	Y	Y	N	N	NIF & GAC
14	apps.babycaretimer_5	\src\apps\babycaretimer\TimerActivity.java	1754	Y	N	N	N	NIF
15	apps.droidnotify_67	y\calendar\CalendarPreferenceActivity.java	202	Y	Y	N	N	NIF & GAC
16	apps.droidnotify_67	eferences\AdvancedPreferenceActivity.java	104	Y	-	N	N	CTC
17	apps.droidnotify_67	eferences\AdvancedPreferenceActivity.java	146	Y	-	N	N	CTC
18	apps.droidnotify_67	eferences\AdvancedPreferenceActivity.java	181	Y	N	N	N	NIF
19	apps.droidnotify_67	eferences\AdvancedPreferenceActivity.java	224	Y	N	N	N	NIF
20	apps.droidnotify_67	ferences\QuietTimePreferenceActivity.java	129	Y	-	N	N	CTC
21	apps.droidnotify_67	c\apps\droidnotify\NotificationActivity.java	286	Y	-	N	N	CTC
22	apps.droidnotify_67	c\apps\droidnotify\NotificationActivity.java	338	Y	-	N	N	CTC
23	apps.droidnotify_67	c\apps\droidnotify\NotificationActivity.java	1591	Y	N	N	N	NIF
24	aws.apps.androidDrawables_8	pps\androidDrawables\activities\Main.java	340	Y	-	N	N	CTC
25	bander.notepad_12	Notepad\src\bander\notepad\NoteEdit.java	248	Y	-	N	N	CTC
26	bander.notepad_12	Notepad\src\bander\notepad\NoteList.java	213	Y	-	N	N	CTC
27	be.ppareit.swiftp_free_28	it\swiftp\gui\ServerPreferenceActivity.java	225	Y	Y	N	N	NIF & GAC
28	be.ppareit.swiftp_free_28	it\swiftp\gui\ServerPreferenceActivity.java	240	Y	N	N	N	NIF
29	budo.budoist_33	src\src\budo\budoist\views\FileDialog.java	247	Y	-	N	N	CTC
30	budo.budoist_33	\src\budo\budoist\views\ItemListView.java	400	Y	Y	N	N	NIF & GAC
31	budo.budoist_33	\src\budo\budoist\views\ItemListView.java	500	Y	Y	N	N	NIF & GAC
32	budo.budoist_33	\src\budo\budoist\views\ItemListView.java	524	Y	Y	N	N	NIF & GAC

C4 – BadTokenException Journal Excerpt (XLS)

	Package	File	Line	NIF Smell	GAC Smell	COR Pattern	ENH Pattern	Comment
2	aarddict.android_21	c\aarddict\android\ArticleViewActivity.java	796	Y	-	-	-	CTC
3	aarddict.android_21	rc\src\aarddict\android\LookupActivity.java	312	Y	N	N	N	NIF
4	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	526	Y	Y	N	N	NIF & GAC
5	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	870	Y	Y	N	N	NIF & GAC
6	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	915	Y	Y	N	N	NIF & GAC
7	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	2064	Y	Y	N	N	NIF & GAC
8	acr.browser.barebones_42	src\acr\browser\barebones\Barebones.java	2300	Y	-	N	N	CTC
9	acr.browser.barebones_42	rc\src\acr\browser\barebones\Settings.java	363	Y	N	N	N	NIF
10	android.androidVNC_13	ndroid\androidVNC\VncCanvasActivity.java	1037	Y	-	N	N	CTC
11	android.androidVNC_13	ndroid\androidVNC\VncCanvasActivity.java	1110	Y	-	N	N	CTC
12	apps.babycaretimer_5	er\preferences\MainPreferenceActivity.java	283	Y	N	N	N	NIF
13	apps.babycaretimer_5	er\preferences\MainPreferenceActivity.java	479	Y	Y	N	N	NIF & GAC
14	apps.babycaretimer_5	\src\apps\babycaretimer\TimerActivity.java	1754	Y	N	N	N	NIF
15	apps.droidnotify_67	y\calendar\CalendarPreferenceActivity.java	202	Y	Y	N	N	NIF & GAC
16	apps.droidnotify_67	eferences\AdvancedPreferenceActivity.java	104	Y	-	N	N	CTC
17	apps.droidnotify_67	eferences\AdvancedPreferenceActivity.java	146	Y	-	N	N	CTC
18	apps.droidnotify_67	eferences\AdvancedPreferenceActivity.java	181	Y	N	N	N	NIF
19	apps.droidnotify_67	eferences\AdvancedPreferenceActivity.java	224	Y	N	N	N	NIF
20	apps.droidnotify_67	ferences\QuietTimePreferenceActivity.java	129	Y	-	N	N	CTC
21	apps.droidnotify_67	c\apps\droidnotify\NotificationActivity.java	286	Y	-	N	N	CTC
22	apps.droidnotify_67	c\apps\droidnotify\NotificationActivity.java	338	Y	-	N	N	CTC
23	apps.droidnotify_67	c\apps\droidnotify\NotificationActivity.java	1591	Y	N	N	N	NIF
24	aws.apps.androidDrawables_8	pps\androidDrawables\activities\Main.java	340	Y	-	N	N	CTC
25	bander.notepad_12	Notepad\src\bander\notepad\NoteEdit.java	248	Y	-	N	N	CTC
26	bander.notepad_12	Notepad\src\bander\notepad\NoteList.java	213	Y	-	N	N	CTC
27	be.ppareit.swiftp_free_28	it\swiftp\gui\ServerPreferenceActivity.java	225	Y	Y	N	N	NIF & GAC
28	be.ppareit.swiftp_free_28	it\swiftp\gui\ServerPreferenceActivity.java	240	Y	N	N	N	NIF
29	budo.budoist_33	src\src\budo\budoist\views\FileDialog.java	247	Y	-	N	N	CTC
30	budo.budoist_33	\src\budo\budoist\views\ItemListView.java	400	Y	Y	N	N	NIF & GAC
31	budo.budoist_33	\src\budo\budoist\views\ItemListView.java	500	Y	Y	N	N	NIF & GAC
32	budo.budoist_33	\src\budo\budoist\views\ItemListView.java	524	Y	Y	N	N	NIF & GAC

C4 – Code Inspection Snippet Sample (JAVA)

```
C:\Experiment\All_Sources\aarddict.android_21_src\src\aarddict\android\Articl
eViewActivity.java

Inspecting line 797 of ArticleViewActivity.java.

Enclosing class is:

69: public class ArticleViewActivity extends BaseDictionaryActivity {


Enclosing method is:

787:                                 new OnClickListener() {


787:                         .setNeutralButton(R.string.btnDismiss,

788:                                 new OnClickListener() {

789:                                     public void onClick(DialogInterface
dialog,

790:                                             int which) {

791:                                         dialog.dismiss();

792:                                         if (backItems.isEmpty()) {

793:                                             finish();

794:                                         }

795:                                     }

796:                                 });

797:                 dialogBuilder.show();

798:             }

799:         });

800:     }

801:

802:     private void setTitle(CharSequence articleTitle, CharSequence
dictTitle) {

803:         setTitle(getString(R.string.titleArticleViewActivity,
articleTitle,

804:                 dictTitle));

805:     }

806:
```

C5 – Recorded Error Log Sample (CSV)

```
apk,timestamp,errors

android.androidVNC_13.apk,monkey,2014-02-06-
22:53:05,java.lang.NullPointerException,2

apps.babycaretimer_5.apk,monkey,2014-02-06-
22:54:57,android.content.ActivityNotFoundException,3java.lang.OutOfMemoryErro
r,1android.util.AndroidRuntimeException,1

budo.budoist_33.apk,monkey,2014-02-07-
00:41:36,android.content.ActivityNotFoundException,10java.lang.NullPointerExc
eption,10

cc.co.eurdev.urecorder_5.apk,monkey,2014-02-07-
02:34:09,java.lang.NullPointerException,10

apps.droidnotify_67.apk,monkey,2014-02-07-
07:29:46,java.lang.OutOfMemoryError,1android.content.ActivityNotFoundExceptio
n,14

com.androidemu.nes_61.apk,monkey,2014-02-07-
08:45:32,java.lang.UnsatisfiedLinkError,6

com.aripuca.tracker_24.apk,monkey,2014-02-07-
10:45:17,java.lang.OutOfMemoryError,9

com.app2go.sudokufree_3.apk,monkey,2014-02-07-
11:34:30,java.lang.OutOfMemoryError,7

com.aselalee.trainschedule_116.apk,monkey,2014-02-07-13:38:52,

com.artifex.mupdfdemo_52.apk,monkey,2014-02-07-13:53:18,

com.bwx.bequick_201107260.apk,monkey,2014-02-07-
15:40:24,java.lang.OutOfMemoryError,5

com.beem.project.beem_15.apk,monkey,2014-02-07-
17:32:50,java.lang.NullPointerException,5

com.commonsware.android.arXiv_106.apk,monkey,2014-02-07-
17:52:06,java.lang.OutOfMemoryError,2

com.eleybourn.bookcatalogue_151.apk,monkey,2014-02-07-
21:04:00,java.lang.OutOfMemoryError,3

com.episode6.android.appalarm.pro_31.apk,monkey,2014-02-07-
23:18:00,android.content.ActivityNotFoundException,93java.lang.OutOfMemoryErr
or,4

com.evancharlton.mileage_3110.apk,monkey,2014-02-08-
01:12:16,java.lang.OutOfMemoryError,1

com.googlecode.awsms_24.apk,monkey,2014-02-08-03:10:46,

com.googlecode.networklog_21800.apk,monkey,2014-02-08-05:01:02,

com.hectorone.multismssender_13.apk,monkey,2014-02-08-06:45:05,

com.leinardi.kitchentimer_116.apk,monkey,2014-02-08-
09:27:19,java.lang.OutOfMemoryError,19
```

C5 – Recorded Test Summary Log Sample (CSV)

```
timestamp,port,strategy,apk,errors,aborts,crashes,restarts,emulator,elapsed,e
vents

2014-02-06-
22:53:05,5554,monkey,android.androidVNC_13.apk,2,4,2,3,2,3283975,39644

2014-02-06-
22:54:57,5556,monkey,apps.babycaretimer_5.apk,5,5,5,59,0,3092767,39814

2014-02-07-
00:41:36,5556,monkey,budo.budoist_33.apk,20,21,20,2,1,2775801,38108

2014-02-07-
02:34:09,5556,monkey,cc.co.eurdev.urecorder_5.apk,10,10,10,0,0,2895639,39570

2014-02-07-
07:29:46,5554,monkey,apps.droidnotify_67.apk,15,15,15,3,0,3028687,39713

2014-02-07-
08:45:32,5556,monkey,com.androidemu.nes_61.apk,6,10,6,52,4,2965191,38950

2014-02-07-
10:45:17,5556,monkey,com.aripuca.tracker_24.apk,9,9,9,8,0,3078703,39749

2014-02-07-
11:34:30,5554,monkey,com.app2go.sudokufree_3.apk,7,8,7,8,1,2970815,39440

2014-02-07-
13:38:52,5554,monkey,com.aselalee.trainschedule_116.apk,0,1,0,1,1,3063839,398
52

2014-02-07-
13:53:18,5556,monkey,com.artifex.mupdfdemo_52.apk,0,5,0,13,3,3035913,39225

2014-02-07-
15:40:24,5556,monkey,com.bwx.bequick_201107260.apk,5,3,5,1,0,3088111,39671

2014-02-07-
17:32:50,5554,monkey,com.beem.project.beem_15.apk,5,5,5,6,2,900912,11705

2014-02-07-
17:52:06,5556,monkey,com.commonsware.android.arXiv_106.apk,2,3,2,9,6,3117489,
39058

2014-02-07-
21:04:00,5556,monkey,com.eleybourn.bookcatalogue_151.apk,3,6,3,7,5,4311033,39
443

2014-02-07-
23:18:00,5556,monkey,com.episode6.android.appalarm.pro_31.apk,97,102,97,12,8,
2999267,33048

2014-02-08-
01:12:16,5556,monkey,com.evancharlton.mileage_3110.apk,1,3,1,6,2,3086323,3966
4

2014-02-08-
03:10:46,5556,monkey,com.googlecode.awsms_24.apk,0,2,0,13,2,2969830,39660
```

C5 – Recorded Test Event Log Sample (TXT)

```
20:46:55[1502580:1502464] - INFO - harness.android.swarms.execute - Starting
monkey swarm on 5554 against
C:\Experiment\Experiment\android.androidVNC_13.apk

 20:46:55[1503644:1503656] - INFO - harness.android.swarms.execute - Starting
monkey swarm on 5556 against
C:\Experiment\Experiment\apps.babycaretimer_5.apk

 20:48:05[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 20:50:52[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 20:51:35[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 20:52:32[1503644:1327888] - INFO - harness.android.observers.notify -
ShortMsg Observer: android.util.AndroidRuntimeException

 20:52:38[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 20:52:46[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 20:56:35[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 20:58:08[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 20:58:28[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 21:01:44[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 21:04:32[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 21:09:30[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 21:10:14[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 21:10:36[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 21:13:48[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 21:16:54[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer

 21:21:52[1503644:1503656] - INFO - harness.android.swarms.recover -
Relaunching package apps.babycaretimer
```