A Survey of Android App Quality Using Third Party Markets

by

Eric Shaw

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 13, 2014

Keywords: Android, Third-Party Markets, Quality, Reverse Engineering, APK files, Google Play

Approved by

David Umphress, Chair, Professor of Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Kai Chang, Professor and Chair of Computer Science and Software Engineering

Abstract

Since its inception, the smart phone has quickly become one of the most ubiquitous technological artifacts in today’s society. In the first quarter of 2014, 281.5 million units were shipped. This was a 28.6 percent increase from the first quarter of 2013. Sales continue to increase year after year, as the phones become even more accessible and intertwined in day-to-day life. Such phones have achieved success due in part to easily available third-party applications (apps). Apps have major implications for both end users and software developers, especially in terms of software quality.

Google’s Android and Apple’s iOS operating systems dominated the market in 2013, accounting for just over 90 percent of the total market, with Android at 79.3 percent. Whereas Apple takes a very restrictive and carefully monitored approach to its app market, Google allows apps to be distributed from any source. Though Google does host an official market, the Google Play store, many third party markets have seen success due to their relaxed policy.

While many Android app markets have a rating system, no mechanism exists by which to assess the quality of an app based on its implementation. In addition, little data exists about the quality of these markets as a whole. In this work we present a process for gathering applications from several third-party Android markets, decompiling them into readable form, and assessing their quality based on their implementation. We use a mixture of traditional and custom metrics to determine the quality of the code. We then present our findings, comparing the performance of apps and markets. We also leverage the app rating system to compare quality from the perspective of the developer and the end user in order to gain new insight into the concept of software quality.

Acknowledgments

I would like to thank my advisor Dr. David Umphress for his contributions of time and expertise which were critical to performing this work. Your guidance has allowed me to conduct research that I find both interesting and valuable. Next I would like to thank my committee members, Dr. James Cross and Dr. Kai Chang for their time and effort.

I would also like to thank my family for their continued support throughout my academic career. It is these contributions that have allowed me to achieve my goals, culminating in this research.

Table of Contents

Abstract ii

Acknowledgments.. iii

List of Tables vi

List of Illustrations vii

List of Abbreviations ix

Chapter 1 – Introduction 1

Chapter 2 – Previous Work 11

Chapter 3 – Solution 26

Methodology 26

Chapter 4 – Solution Validation 51

Results 51

Chapter 5 – Conclusions and Future Efforts 64

Conclusions 64

Future Work 65

References 66

Appendix 1 – Source Code 71

Slide Me Targeted Crawler 71

Apps APK Targeted Crawler...73

F-Droid Targeted Crawler..75

Decompiler...77

Driver 78

Manifest Data Extraction 83

Size Metrics 85

Object Oriented Metrics...87

Class...91

Method 93

View Metrics..94

Android Views 96

Unchecked Bundles 98

Android Objects 100

Bad Smell Method Calls 102

Component Launch Metrics...105

Battery Metrics 108

Black Hole Exception Handling 110

ANR Metrics..112

Network Timeout Metrics..115

Slide Me Rating and Metadata Collection 117

Appendix 2 – Additional Data 119

Size Metrics 119

Object Oriented Metrics...121

View Metrics..123

BadTokenException...124

Fragments...124

Bad Smell Method Calls 125

Component Launch Metrics...129

50 Most Frequently Occurring Android Objects 131

Demographic Metrics...133

List of Tables

Table 1 – David Garvin’s Five Views of Quality 19

Table 2 – Dalvik Byte-Code Registers 38

Table 3 – Conditional Statements in Dalvik Byte-Code 41

Table 4 – Top 10 Methods Causing Crashes 45

Table 5 – Component Launch Metrics...47

Table 6 – Results for Size Metrics..52

Table 7 – Results for Object Oriented Metrics 52

Table 8 – Results for View Metrics 53

Table 9 – Results for Unchecked Bundles, BadTokenExceptions, and Fragments 53

Table 10 – Results for Bad Smell Method Calls 54

Table 11 – Results for Battery Metrics 54

Table 12 – Results for Component Launches 54

Table 13 – Results for ANR Metrics 55

Table 14 – Results for Black Hole Exception Handling 56

Table 15 – Results for Demographic Metrics 56

List of Illustrations

Illustration 1 – Smart Phone Market Share by Operating System, Second Quarter 2013 2

Illustration 2 – App Revenue by Market, 2013..4

Illustration 3 – Sample Method Declaration in Objective C 7

Illustration 4 – Rating information for a random app on Google Play 9

Illustration 5 – A warning provided by the lint program. 21

Illustration 6 – A random user comment for a random app 27

Illustration 7 – Some of the Metadata provided by Google Play for a random app 27

Illustration 8 – Market rating for a random Slide Me app 31

Illustration 9 – Comment for a random Slide Me app 31

Illustration 10 – Apps Considered by Market 32

Illustration 11 – .apk Directory Structure 34

Illustration 12 – AndroidManifest.xml File Structure..35

Illustration 13 – Hello World Smali Program 37

Illustration 14 – McCabe’s Cyclomatic Complexity 40

Illustration 15 – Public Field in Smali Byte-Code 42

Illustration 16 – Accessing a Field in Smali Byte-Code 42

Illustration 17 – Slide Me Ratings Sorted Linearly 51

Illustration 18 – Metrics with Highest Difference Index 57

Illustration 19 – Market Comparison by Metric 58

Illustration 20 – Taxonomy of Bad Smells 59

List of Equations

Equation 1 – Difference Index 52

List of Abbreviations

ADB	Android Debug Bridge
API	Application Programming Interface
APK	Android Package File
APP	Android Application
CAPTCHA	Completely Automated Public Turing Test to Tell Computers and Humans Apart
DEX	Dalvik Executable
DVM	Dalvik Virtual Machine
GB	Gigabyte
HTML	Hyper Text Markup Language
JAR	Java Archive
JVM	Java Virtual Machine
OS	Operating System
PC	Personal Computer
SDK	Software Development Kit
XML	Extensible Markup Language

CHAPTER 1 – Problem Description

Smart phones have been absorbed into everyday life at an astounding rate, and continue to become more and more widely used. Data from the International Data Commission shows a 28.6 percent increase in smart phone production between the first quarter of 2013 and 2014, with 281.5 million units shipped [IDC 14]. This number has continued to grow since the mass adoption of the smart phone. Mobile subscription rates reached an all-time high in 2013, reaching 96 percent of the global population [ITU 13]. While this does include many individuals with multiple subscriptions, it demonstrates the truly profound effect that the smartphone has had on global communication. Such acute penetration has revolutionized the way humans interact and has the potential to do so at an even deeper level.

The origin of the smart phone cannot be clearly defined, but can be traced at least as far back as the Personal Data Assistants (PDA) popular in the 1990s. Such devices used a stylus for interaction, and ran operating systems (OS) such as Palm OS and Blackberry OS [REED 10]. PDAs were used primarily for business purposes and not appropriated for personal use on a large scale. Since the release and widespread adoption of Apple's iPhone in 2007, touch-based interfaces have become the norm for smart phones, and the general use of the phones has expanded far beyond the business realm [APPLE 07]. Today the most popular devices, those running Apple's iOS and Google's Android OS, follow this paradigm. Illustration 1 shows a breakdown of the market share claimed by the most popular smart phone operating systems, as of the second quarter of 2013. iOS and Android are by far the most widely used, combining to make up over 90 percent of the market. Android has outpaced iOS in recent years, trending more and more toward market dominance [IDC 14]. Microsoft has seen limited success with its Windows Phone, despite its increasing commitment to the mobile market [IDC 13]. The overwhelming success of Android and iOS is due in large part to the extreme success of their application markets.

Much of the success of the modern smart phone paradigm can be attributed to the presence of third party applications (apps). Users may pick from a large collection of software, in domains ranging from games to productivity. Each platform makes the task of installing and removing apps very simple, further inciting users to try new software. In 2013, both Apple's App Store and Google's Play Store boasted their respective 50 billionth downloads. Google currently hosts over 1 million apps in its market, while Apple claims 900,000. Such an overwhelming number of apps can make distinguishing among them a daunting task. [WELCH 13] [APPLE 13]

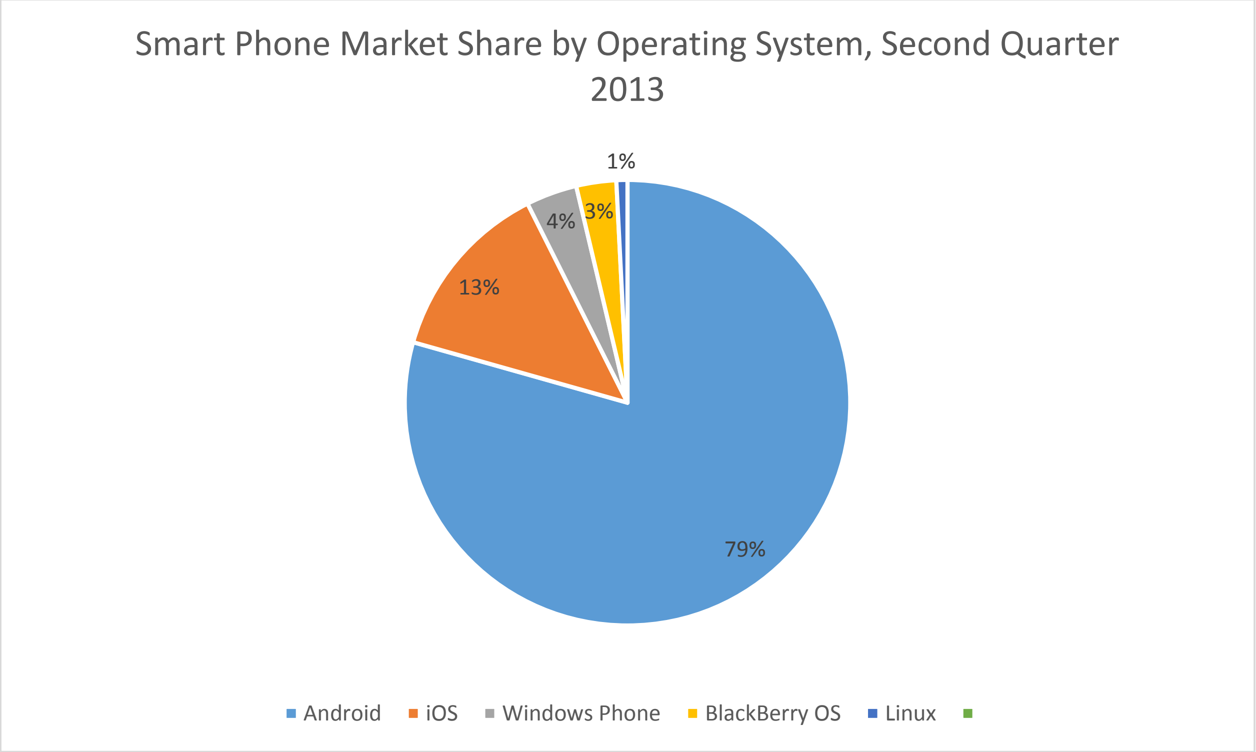


Illustration 1: Smart Phone Market Share by Operating System [ITC 13]

One of the reasons for such a vast number of published apps is the large potential for profit. King Digital Entertainment, developers of the popular "Candy Crush Saga" mobile game (and several others), reported $606.7 million in revenue in the first three months of 2014 alone [KING 14]. Such immense profits have incentivized developers to publish apps, saturating the markets with apps that are often indistinguishable. There are four main ways to monetize mobile applications. First, a client may pay the developer to create an app with the desired specifications. Anything from personal interest to business promotion can fall into this category. A second way to monetize apps is through in-app advertisement. Apple and Google both provide advertisement frameworks through which a developer can earn revenue based on the success of his app. Third, the app creator may make use of in-app purchases. These are commonly used in games, allowing the player to purchase new materials such as additional levels or power-ups for a nominal fee. Finally, the developer may choose to charge a fee for the download of his app, ranging from $.99 to a market-specific maximum. Often a developer will publish a corresponding free version with a paid app, with a reduced set of features. He hopes to entice the user to purchase the full app through the free preview. The use of any of the latter three of these monetization schemes results in the market owner taking a small cut. It is important to note that these methods are not mutually exclusive – all could be used in conjunction. Additionally, this list is not exhaustive – there are many ways to profit with mobile apps. Illustration 2 shows a comparison of app revenue across the proprietary markets of each of the three most popular platforms in 2013. Interestingly, Apple boasts higher per-app and per-developer profits despite its lower market share.

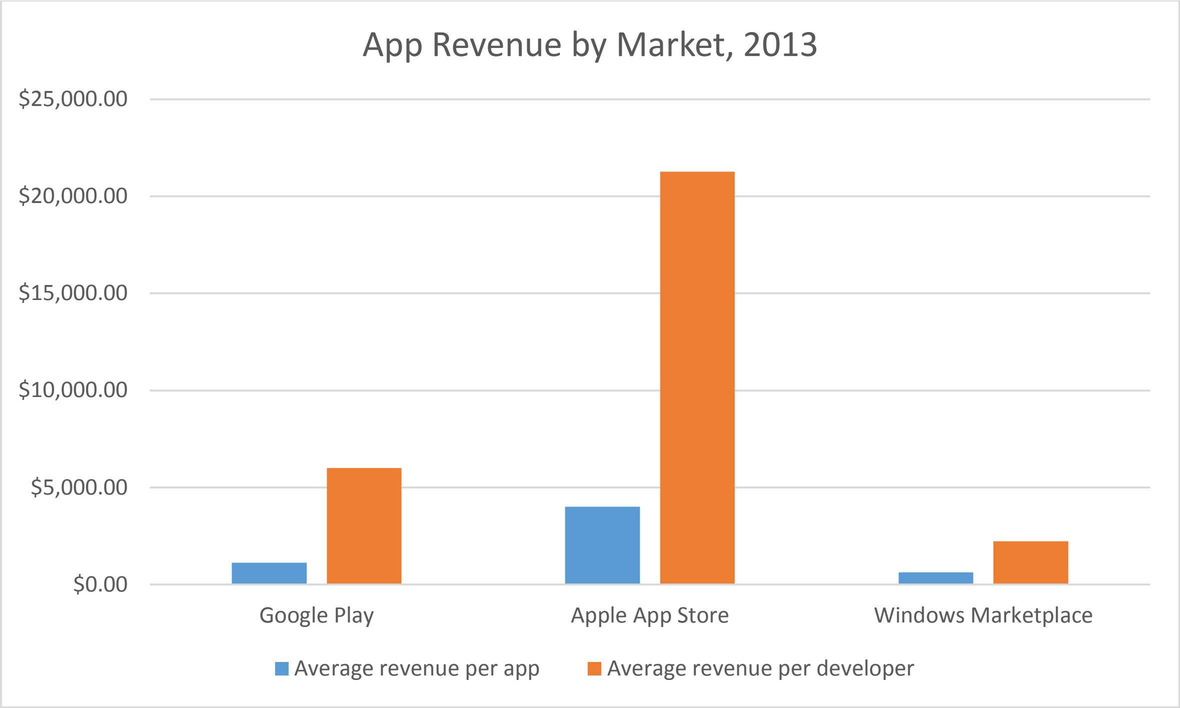


Illustration 2: App Revenue Statistics as of August 2013 [LOUIS 13]

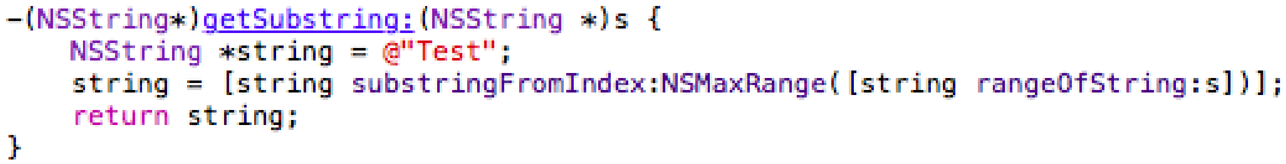
Apple and Google take starkly differing approaches toward the distribution of apps for their respective platforms. Apple takes a restrictive "walled garden" approach, in which it assumes all responsibility for verifying that an application meets certain standards [BARRERA 10]. In order to distribute an app in the App Store, a developer must submit it to Apple for official review. While Apple is secretive about the details of this vetting process, they do state that the review "ensures that apps on the App store are reliable, perform as expected, and are free of explicit material" [APPLE 14a]. No assessment mechanisms are published, but registered developers may access an extensive list of guidelines. To the frustration of many developers, Apple does not provide any guarantees on the amount of time that a review will take. An entire site, http://appreviewtimes.com/, has arisen around the desire to measure the expected turnaround for a new app to enter the market. At the time of this writing, the site listed 4 days as the average wait time [SHINY 14]. Apple does provide some visibility into the review process, but it is very course-grained, allowing the developer to only see whether his app is waiting for review, in review, or post review, and any decisions that have been made. In extenuating circumstances a developer may request an expedited review, greatly reducing the time to market.

Google's approach is much more open, reflecting the philosophy of the company. Mobile device users may download applications from the official Google Play store (formerly the Android Market) or from one of a number of third party markets. Apps on the Play Store are not reviewed in any way, and can be downloaded very shortly after submission. However, Google does screen apps for certain undesirable qualities after they have been uploaded, and will remove an app if it is found to be in contrast with the company's standards. Generally applications are removed when found to be malicious or contain inappropriate content. This unrestricted model is often called the "wild west" [HIGA 08]. In this paradigm, more responsibility is placed on the user to screen an app and determine whether it is safe and whether it meets his needs. This can be problematic as the app may be new, and the user likely has little insight into how the app works. Even an experienced developer will have no way to accurately assess an app without downloading it. Google attempts to protect the user through the use of permissions [GOOGLE 14a]. In order to perform certain sensitive tasks, such as accessing the user's contacts, an app must request permission at the time of download. If the user grants this permission, it is granted as long as the app is installed on the phone. Thus responsibility of screening an app shifts from the market owner to the application user.

In addition to its own Google Play store, Google allows the distribution of Android apps through third party marketplaces. Many of these have seen success. Most popular is the Amazon App Store, which currently hosts over 50,000 free and paid apps [AMAZON 14]. The company launched its market along with its Kindle Fire, an Android-enable electronic reading device. The Fire comes with the Amazon marketplace set up and ready for download, but the Google Play store must be downloaded separately. By doing this, Amazon attempts to incite users to use its market rather than Google's. In fact, some devices are not compatible with the Android market at all, and rely completely upon third party app distribution [GOOGLE 14b]. Such third party markets distribute binaries directly, which can be installed directly from a phone or from a PC using a USB connection. Third party markets take on a variety of forms, but often closely follow the model presented by the Google Play Store, requiring developers to register in order to distribute free or paid apps. Others take, for example, an open source approach in which no account is required and source code is freely available in addition the binaries. In its most primitive form, a market may simply be a collection of packaged apps.

Android binaries come in the form of .apk files, standing for "Android Package." These files contain all executable code and resources necessary to run an app on a device. A developer can easily create one directly from source code using command line tools or a graphical user interface provided by Google's Android development toolkit. Smart phone users may place .apk files directly on their devices, whether from a market or directly from a computer. This means that apps can be distributed directly among peers, without the need of any overseeing third party. Such flexibility has several useful applications such as acceptance testing before software is released. However, distributing apps through channels not monitored by Google brings up many security concerns. By default, a user may not install an app from any source other than Google Play – he must explicitly allow this capability. Therefore, the user assumes responsibilities for vetting any apps that he plans to install. In Apple's walled garden model, the user need be much less wary of the product he is downloading.

Further widening the gap between the two markets is the barrier to entry for developers. In order to register to submit apps to Apple's app store, the developer must pay a fee of $99. This fee is assessed upfront, and recurs each year [APPLE 14b]. Google requires only a one-time registration fee of $25 for its Play store, citing a desire to prevent useless apps and encourage higher quality products as motivation [GOOGLE 14c]. Third party markets often allow developers to distribute apps without registration, bringing up the question of how this affects app quality. Also increasing the barrier of entry into the App Store is the requirement to learn a custom programming language, Objective C. This is an object-oriented language built on top of C, but its syntax differs greatly. Illustration 3 demonstrates a sample of the syntax. Apple recently unveiled a new custom programming language, Swift, which can be used to implement iPhone apps [LENDINO 14]. This may address the steep learning curve, as it is designed specifically around usability [APPLE 14c]. Android applications are implemented in Java, a widely known and used programming language that will generally require no additional training. Development for an iOS application must be done on a Mac operating system, whereas Android development is cross-platform, allowing programmers to write apps using Windows, Mac, and Linux operating systems. Apple requires the use of a custom Integrated Development Environment (IDE), Xcode, requiring new users to become familiar with the tool. Google allows development in the Eclipse IDE, one of the most widely used, through a plug-in that is easy to download.



```
-(NSString*)getSubstring:(NSString *)s {
    NSString *string = @"Test";
    string = [string substringFromIndex:NSMaxRange([string rangeOfString:s])];
    return string;
}
```

Illustration 3: A method definition in Objective C

Many of the factors discussed bring up the question of what quality attributes are present in mobile apps. Software quality is a term that is used broadly within software engineering, and takes on various meanings. Often, practitioners disagree over what exactly quality entails [PETRASCH 99]. A common definition is that quality is the extent to which a software system meets the specifications, in other words the extent to which the product performs the expected task. Others define quality as conformance to a set of attributes, such as maintainability, readability, performance, security, usability, and testability. Neither definition fully encompasses an entire software artifact, nor is either easily or accurately measured. Quality is thought of as measuring how good a product is, which is generally subjective. Some may perceive a particular piece of software as high quality, while others denounce it as quite useless. For example, one user of an email service might consider it of high quality because it reliably delivers his mail, while another may complain that the service is too slow and thus of low quality. The desire to build the best software possible along with the inherent subjectivity of software quality have led it to become a subject of much research.

The intersection of software quality and mobile application research is relatively new. No standard mechanisms exist by which to assess app or market quality. Many marketplaces, including the App Store and Google Play, have a rating system that allows the end user to assign a score out of five to each app. Illustration 4 gives an example of an app rating on the Google Play Store. No criteria is given as to how these apps should be scored, so the decision is left completely to the user. Users may also leave comments regarding the app. Many of these indicate the user's feelings toward the quality of the app, often advising others on whether the app is worth downloading. Outside of the official marketplace, Android markets may or may not provide a similar rating system. They may also choose to perform their own app reviews, similar to those performed by Apple. In this manner they can attempt to control the quality of the applications, though the definition of quality is from their own perspectives.

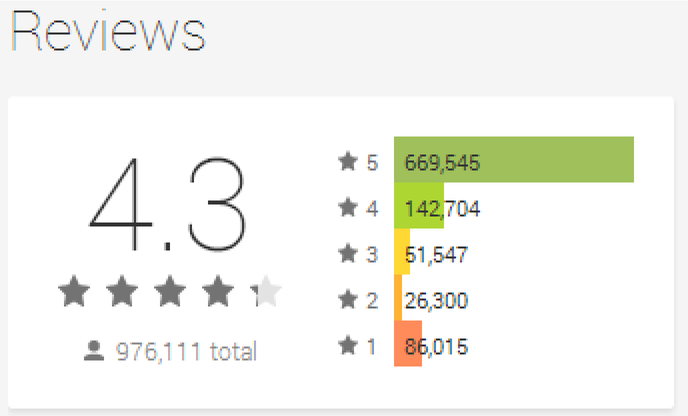


Illustration 4: Rating information for a random app on Google Play

Though some markets offer a system for users to share their opinions on app quality, no such mechanism exists for users to compare the quality of different markets. This stems in part from the fragmentation of the markets themselves – there is no centralized repository of markets. However, a method of assessment would give app users the utility of determining which markets should be used and which should be avoided.

With such a large presence of apps and markets, and their ever-increasing popularity, the question of how to determine the quality of an app and its marketplace is one that warrants consideration. Despite the overwhelming number of developers and users, no mechanisms exists to compare mobile applications and marketplaces based on their quality. It was this perceived gap that drove us to explore the topic. The collection of data could be useful in many ways. Smartphone owners could use the data to determine what apps to download, and from which marketplaces. Market maintainers could compare their performance to that of other marketplaces, and use the data to improve their performance. Our research might demonstrate disadvantages of current application rating systems, and point toward a new assessment mechanism. App developers could gain insight into what particular implementation practices and patterns lead to apps of various qualities. This would help them better meet the expectations of the end user. With this motivation, we selected the assessment of mobile application and marketplace quality as the subject of this master's thesis.

The Android platform was the obvious choice in performing our research. Most importantly, there are many marketplaces that we could choose from, giving us the ability to compare the quality of each. Though it would be possible to compare markets across platforms, such analysis would be difficult. Additionally, Android apps are distributed in a way that we could download and analyze them. Since each market is simply a repository of .apk files which are distributed in an unrestricted manner, it was easy to gather a collection from several markets. Apple's close guard on binary files would not suit our needs. The open nature of the platform, large market share and presence of applications, and large developer base all pointed us in the direction of the Android OS.

In performing our research we had three main goals. First we wanted to identify a way to assess the quality of an Android application based on the contents of its .apk file. Second, we wanted to use this data to build an assessment mechanism for the quality of a market as a whole. Finally, we wanted to use this data to evaluate the rating systems currently in certain mobile application markets.

CHAPTER 2 – Previous Work

Our work in performing this study builds on that presented in William Symon's master's thesis [SYMON 12]. Symon presented a demographic overview of several third party Android markets, examining information such as permissions used, SDK (software development kit) level, external libraries, and advertising schemes. Such findings pointed to development practices among Android programmers, some of which could be improved. The author then used the information to compile a demographic overview of the marketplaces considered. In his thesis, he presents a process for downloading a large body of apps, reverse engineering them, and gathering demographic information.

Though he chose to focus on demographic information for his study, Symon originally identified the lack of quality assessment mechanisms for mobile apps and marketplaces. His problem identification stemmed from the goal of adapting a personal software process for mobile development. This led to the idea to examine a body of Android apps to determine how they were being developed. In many ways our work closely resembles his, and as such we were able to leverage some of the techniques presented. We wanted to download and reverse engineer a large collection of apps from third party Android marketplaces. Although we targeted different markets, the description of the process followed, tools used, and provided source code were crucial for our work. In order to collect the applications needed for his study, the author developed several web crawlers using the Python programming language. These used regular expressions to identify the URL of .apk files in the HTML source of the selected markets. Symon's analysis focused on artifacts that could be gathered directly from an .apk file. He presented several tools for decompiling a packaged Android application, eventually landing on apktool [WISNIEWSKI 12] because of its ability to convert the AndroidManifest.xml file into human readable form and to produce resources in nearly their original forms. In addition, the tool can produce Dalvik byte-code (a special form a byte-code specific to Android) corresponding to each source code file, and can be used to repackage applications in an .apk file.

This means that the app could be modified and repackaged, potentially allowing developers to fix errors or add features to apps after distribution.

In order to determine if we could measure android application quality from an .apk file, we needed to look into the artifacts produced in the disassembly process. Since this varies based on the tool used, we needed to determine if apktool was indeed suitable for our purposes. Though we had not yet developed our quality metrics, we knew that we would focus primarily on the static analysis of code and resource files. By code we did not necessarily mean the original Java source code, but we did need a human readable artifact. Since apktool can produce resources in their original form, we knew it would be an acceptable tool in this regard. Thus we needed to examine Davlik byte-code to determine whether it was readable and corresponded to the original source code. After reading Symon's thesis and some experimentation with apktool, we found that Dalvik byte-code is in fact readable, and that it correlates very closely with the original Java source. For these reasons, we decided to use apktool for disassembly.

After determining the focus of our study, we determined three areas in which to explore previous work. First, we needed to examine mobile computing and the Android platform, in order to gain historical context and a knowledge of what artifacts we could analyze for our quality assessment. Second, we needed to explore work on the topic of software quality in order to understand how it might be evaluated. This includes work on both software quality in general and mobile applications specifically. Finally, we needed to examine work on large bodies of Android apps in order to determine what process we should follow to do this and to identify any trends of which we should be aware.

Sharon Hall and Eric Anderson's "Operating System for Mobile Computing" provides a concise look into the history of the smart phone up to the time of publication [HALL 09]. PDAs such as the Palm 1000 gained some traction in the business market in the mid-1990s. Functionality was limited, but the devices could perform simple computing operations such storing contacts and keeping track of scheduled appointments, which could be synced with a computer. Society stood on the precipice of the mass adoption of the cellular phone at the same time, with more and more subscribers each year. In order to encourage subscribers, cellular service providers began to add new features to their devices. Research in Motion (RIM) released the Blackberry in 1999, a device which soon added the ability to send email using a full QWERTY keyboard. Microsoft also pushed itself into the mobile market with its Pocket PC 2000. These devices featured Internet connectivity and many features standard to desktop computers. Eventually the market for these powerful handheld devices converged with that of cell phones. Phones not only had the power to make calls and send text messages, but could also send emails, surf the web, and run a wide range of software applications. Five hardware manufacturers - Nokia, Ericsson, Sony, Panasonic, and Samsung - joined forces to create a mobile operating system, known as the Symbian OS, in an effort to combat hardware fragmentation. This OS held a large market share until Apple released the iPhone in 2007. It was this event that sparked the mass adoption of the smartphone. At the time of publication, iOS held the largest share of the mobile market. The authors discuss the beginnings of the Android OS, which emerged from a company acquired by Google in 2005. After noticing a large increase in the number of mobile devices accessing their site, Google identified the smart phone market as a potential boon. Thus they formed the Open Handset Alliance with several mobile companies and developed the Linux-based Android OS. The authors foreshadow the platform’s rise to market dominance, citing its open nature and ease of application development as factors that might lead to its success.

After examining the history of mobile computing, we turned our attention to understanding the Android platform itself. In “Mobile Devices – An Introduction to the Android Operating Environment: Design, Architecture, and Performance Implication,” Dominique Heger presents the underlying design of the Android OS [HEGER 12]. Android is built on a layered architecture, with a modified Linux 2.6 kernel at its core. Changes to the kernel have been made in order to optimize mobile performance. Several libraries critical to development are present in the next layer up. Also in this layer is the Android runtime, consisting of core libraries and the Dalvik Virtual Machine (DVM), a virtual machine optimized for the mobile environment. The DVM is used to run applications, written in Java. Traditionally a Java program is converted to Java byte-code and executed on a Java Virtual Machine (JVM). The DVM does not execute Java byte-code, but rather a special form called Dalvik byte-code. Because the apps run in a virtual machine, they are isolated from other apps. Only with special permission can applications share data. When running an Android application, the compiler converts the Java source code into Java byte-code, which is then transformed to Dalvik byte-code, or a Dalvik executable (dex). For this reason, Dalvik byte-code is also known as dex byte-code. Understanding this process proved crucial to our study, as most of our work relied on the analysis of dex byte-code. Heger continues his discussion of the Android OS architecture with the application framework layer. This layer contains the application API interface, and as such holds several constructs with which app developers should be familiar. Included in these are the activity manager, which controls the flow of execution through various lifecycle methods for each view, and the resource managers, which facilitates the use of resources such as layout files and graphics. At the highest layer are the applications themselves, including custom third party applications and those that come packaged with each device.

Heger's paper prompted us to explore in more depth the Android application lifecycle, as it represents a concern unique to mobile applications. Google presents the topic in its Android Developers Guide [GOOGLE 14d]. Each screen that a user views in an Android app has a corresponding source code file known as an activity. These must extend the Activity class provided by Google (android.app.Activity). When an activity is loaded it is pushed onto the program stack. Launching another activity will simply add the new one to the stack, while navigating to the previous screen via the "back" hardware button will remove the activity from the stack. Once an activity is loaded, it calls several methods, each representing a different phase of the lifecycle. Developers implement these callbacks as needed. When a program is first loaded, the application calls the onCreate() method. Here the app developer should create or load a view, and restore the activity state based on passed data if necessary. Data is passed in a Bundle, a custom Java object that can be used to encapsulate any kind of data. Immediately before the view is shown to the user, the onStart() method is called. Whereas the onCreate() method is only called when the Activity is loaded, onStart() is called each time it comes into view. A corresponding onStop() method is called when the Activity is removed from visibility. An onDestroy() method is called when an instance of an Activity is being destroyed. This could be done to save memory or by an explicit call.

Considering the app lifecycle brings up several concerns for developers. In order to successful create an application, the programmer must understand the lifecycle. This is complex and does not directly correlate to non-mobile applications, which generally have a smaller subset of execution states. Failure to properly use the android navigation mechanisms could result an overflow of the program stack. Passing data between views and persisting that data when the application is paused, stopped, or otherwise changing states requires careful attention to serialization techniques and the use of custom Android objects to wrap and transport the data. We later discovered that one of the most common errors in Android development was the failure to validate the Bundle object, used to pass data between activities on specific lifecycle events, and ultimately included this in our study.

Also published by Google on the Android Developers Guide is a set of best practices for app development [GOOGLE 14e]. We identified these as potentially useful in an assessment of quality. The flagship article is on supporting multiple screen sizes, reflecting the hardware fragmentation of the Android platform. Devices of all sizes are capable of running the same apps (unless explicitly disallowed), and as such, interfaces must conform to each. Screen sizes are divided into four categories: small, normal, large, and xlarge, based on a diagonal measure. Small and normal sizes generally belong to phones, while large and xlarge represent tablets. Localizations can also be used to support screen orientation, namely "land" for landscape. Symon's study looked into the use of each of these, finding that less than 25 percent of apps used them. By far the most popular localization was land at around 13 percent of all apps, with xlarge in second at around 4 percent [SYMON 12]. Also in the best practices section of Google's guide is an article entitled "Improving App Quality". The article did provide some general tips, but did not present any metrics that could be measured from an app's implementation.

Next we turned our attention to work on software quality. In his classic paper, "No Silver Bullet: Essence and Accidents of Software Engineering," Frederick Brooks addresses indirectly the topic of software quality [BROOKS 87]. The paper details the author's argument that no single technique promises any degree of improvement in productivity, reliability, or simplicity. Brooks discusses several developments that were perceived as potential breakthroughs, and why they are not the "silver bullet" for software development. He argues instead for a disciplined, consistent approach to engineering. Throughout the paper, suggestions of the concept of quality can be seen. Brooks points out the difficulty of conceptually modeling software, leading some designs to be better than others. The result is a perceived separation among end products, which can be interpreted as quality. From the beginning it was important to understand that we could not expect one single metric to be a “silver bullet”. While we might find interesting and significant correlations between program implementation and our definition of quality, the complex nature of software implies that this makes no guarantees.

In their 1968 paper, “Quantitative Measurement of Program Quality,” Raymond J. Rubey and R. Dean Hartwick propose a method of assessing quality based on a set of metrics [RUBEY 68]. They state that quality is generally assessed by determining to what degree the specifications are met, but this approach is not sufficient to fully describe how good a program really is. No method of quality assessment beyond correct program functionality had been widely adopted. The authors identified seven categories of attributes that programs should satisfy: (1) Mathematical calculations are correctly performed, (2) The program is logically correct, (3) There is no interference between program entities, (4) Computation time and memory usage are optimized, (5) The program is intelligible, (6) The program is easy to modify, (7) The program is easy to learn and use. From these general attributes, the authors derived a more specific set of metrics that could be directly measured from a program, many in an automated fashion. Using these metrics, the authors performed a survey of space-industry software for the Air Force Space and Missile Systems Organization. Although the authors developed their assessment mechanism with this domain in mind, they argue that their metrics are applicable to all types of software. They then normalized metric scores across apps to determine relative quality. It is important to note that these metrics do not measure quality directly, nor are they intended to do so. Metrics measure surrogates of software quality – that is, they serve as an appropriate substitute. Due to its abstract nature and its malleable definition, software quality cannot be measured directly, nor is it likely that a mechanism to do so will ever exist. However, metrics give us some insight into the quality of a software artifacts. The use of software metrics, which could be directly measured in an automated fashion, was key to our study.

Building on the work presented by Rubey and Hartwick, Barry Boehm *et al.* presented their work on the creation and evaluation of several software metrics in “Quantitative Evaluation of Software Quality” [BOEHM 76] The paper focuses on three topics: how to characterize software quality in a measurable way, how to measure quality through the use of specific metrics based on this model, and how these metrics can be used in the software lifecycle to improve code quality. In their coverage of the first of these topics, the authors discuss the findings of their previous study, “Characteristics of Software Quality” [BOEHM 73]. In this study, two separate programs were written based on the same specification, given instructions to focus on a specific nonfunctional requirement. The authors developed metrics to assess the quality, and found them useful in distinguishing the programs. After discussion of the previous work, the paper’s attention turns to characterization of high quality code. A list of nonfunctional requirements are given in the form of a hierarchy, with general utility as the root. Among the code characteristics listed are portability, reliability, testability, efficiency, understandability, and modifiability. Many of these are further broken down into even more primitive characteristics, such as conciseness and legibility for understandability. The authors assert that their set of quality characteristics are non-overlapping and reasonably exhaustive. From these characteristics, they derive a set of 151 metrics to directly measure program quality. First, they analyze each metric to determine how well it correlated with the associated quality characteristic. Then they determine how important it is for a software artifact to have a high score for that metric – in other words, how beneficial the metric might be. They also resolve the ease with which the metric could be gathered from a program in an automated fashion, and the completeness of the automated evaluation. Applying this to a set of programs, the authors found that the automated collection of their metrics lead to improvement in error detection and detection. Finally, they discuss how their findings can be used to improve the software life-cycle, using their metrics to detect errors early. This study is of particular importance to ours, because it demonstrates the use of metrics to analyze program quality.

What Boehm described in his classic paper is generally referred to as a framework or model for software quality. These provide guiding principles for developers but allow change to accommodate the various needs of consumers. Most modern models of quality are based in some part on Boehm’s and on that presented by Jim McCall *et al.* for the U.S. Air Force in 1977 [MCCALL 77]. In their model, three types of quality characteristics (often known as the "Triangle of Quality") are defined: product operation, product revision, and product transitions. Product operations refer simply to how well the product performs. This includes correctness, or conformance to specifications, as well as nonfunctional requirements such as efficiency and reliability. Product revision refers to the ease with which the system can be modified, and is defined by the characteristics maintainability, flexibility, and testability. Finally, product transition describes the ease with which a system can be transferred from one environment to another. This could mean different hardware, a different location, or even a different programming language. Portability, reusability and interoperability are included in this category. McCall's model partitions the system into nonfunctional requirements, which can be used to derive directly measurable quality metrics.

A commonly used model in practice is the ISO 902, a specification for which can be purchased from the International Organization for Standardization (ISO) [ISO 11]. This document describes a set of attributes that can be used to describe software quality, along with instructions for measurement and evaluation. Included in this set are functionality, portability, maintainability, reliability, usability, and efficiency. Despite the utility provided in terms of program measurement, the ever increasing complexity and eclectic nature of software mean that the model will rarely be suitable for use out-of-the-box. Some degree of tailoring is necessary in order to reflect the attributes important for a particular product. Still, the ISO model demonstrates a disciplined approach to the definition and measurement of quality in a wide variety of software applications, laying the groundwork for much future research.

Several studies ([JAMWAL 10] ,[AL-BARDEEN 11],[DUBEY 12]) have been performed with the goal of comparing software quality models. These usually present a description of which nonfunctional attributes are considered by each. One model does not emerge as universally more fit for software than any others. The conclusion is that the developer must select a model that most closely fits his purposes. Additionally, he will generally have to modify the model in some way in order to meet his needs.

Boehm's work, along with that of Rubey and Hartwick, have led to the widespread use of software metrics to describe frameworks for software quality in many domains. Among these are [SCHOLTZ 04], which suggests a general-to-specific framework for metric development, [HAIGH 10], which describes the importance of each of several quality attributes in an IT-business setting, and [ALVERO 10], which offers a framework to assess the quality of software components in the context of Component Based Software Engineering. Despite all the work done on the assessment of software quality, no truly universal framework has emerged. This is due in large part to the abstract and subjective nature of quality.

In "Software Quality: The Elusive Target," Barbara Kitchenham and Shari Lawrence Pfleeger take an in-depth look at software quality, with focus on its inherent subjectivity [KITCHENHAM 96]. Both the definition and measurement of software quality come under scrutiny in in this paper. In defining software quality, the authors discuss five views of the concept, as defined by David Garvin. Table 1 summarizes these views.

View	Description of Quality
Transcendental view	Quality can be recognized, but not defined
User view	Quality is the degree to which the product is fit for the user
Manufacturing view	Quality is the degree to which the product conforms to its specifications
Product view	Quality is defined by nonfunctional product attributes
Value-based view	Quality is determined by the amount a customer is willing to pay for the product

Table 1: David Garvin's Five Views of Quality [GARVIN 84]

The authors conducted a survey in which they asked software developers to define their own view of quality and compare it to those presented by Garvin. Most commonly reflected were the user, manufacturing, and product views. Nearly all respondents indicated that the disagreements on the nature of quality was problematic.

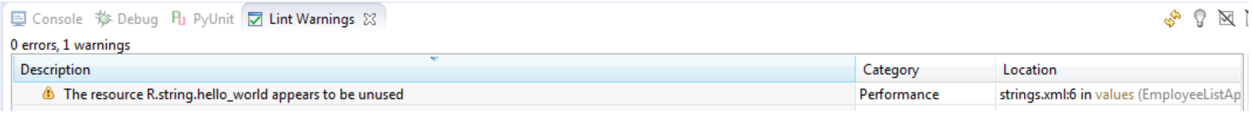
Kitchenham and Pfleeger then turn their attention to measuring software quality. They identify three factors that influence the user experience of a software product: conformance to specifications, nonfunctional qualities, and the constraints that determine whether the customer will use the product.

Whereas the first two are common definitions of quality, the third adds a new dimension to the analysis. For user and manufacturer views of quality defined by Garvin, the authors suggest methods to measure the quality of a software artifact. Measuring the user's view relies on defining the attributes important to the user, while measuring the manufacturer's view focuses on defect counts and rework counts. However, in either case the evaluator must clearly specifiy his own definition of quality. The authors conclude that the definition of quality should be derived from business goals, and thus will vary among projects.

With an understanding of work done in the realm of software quality, we turned our focus to work on mobile application quality. An organization known as the App Quality Alliance (AQuA) has arisen around the desire to encourage the best applications possible. AQuA published in 2013 the most recent version of a document detailing best practices for mobile applications [AQUA 13]. While those listed are very general, they do represent many attributes often associated with quality. Examples include requesting the minimum number of necessary permissions, keeping launch time short, and ensuring that all network activities are performed in the background rather than on the user-interface (UI) thread. Several other organizations have similar best practice documents, including the Groupe Speciale Mobile Association (GSMA) [GSMA 12]. AQuA claims that the organizations work closely with one-another to ensure consistency among their documents. Whereas this resource did not present any directly measurable quality assessment mechanisms, it did point to several general concepts that might be used in establishing a standardized framework for mobile app quality.

In February 2014, *IEEE Computer* published an issue on software testing, an important mechanism for quality assurance. A featured article from Jerry Gao *et al.* focused on software testing for mobile applications [GAO 14]. We examined this piece with the motivation that testing criteria may indicate potential surrogates for app quality. Traditional software testing is well-established and widely researched, with many tools and frameworks. Mobile apps bring new considerations, not limited to hardware fragmentation, inconsistent network connectivity, and complex lifecycles. Such factors indicate that prior testing frameworks are insufficient to fully exploit this new form of software. The authors identify several goals of application testing, analogous to non-functional requirements including performance, reliability, scalability, interoperability, usability, security, and privacy. After dividing these into subcategories, the authors evaluated the testability of each with one of four approaches: emulation-based testing, device-based testing, cloud testing, and crowd based testing. Emulation-based testing refers to running the application on a virtualized device, usually one provided with the SDK for the particular platform. Often functionality is limited on these devices – for example, testing gesture recognition is difficult or impossible on emulators. Device-based testing follows similar procedures, but with an actual device. It is generally recommended that all apps are tested on an actual device before submission to a marketplace. Cloud testing relies on the use of cloud service providers for a web-based framework for large scale testing, and crowd-based testing refers to the use of testing engineers or a community of end users to evaluate the product. Analysis of the suggested test criteria served as a primer for developing mobile application quality metrics.

One of the most helpful sources we found was the set of checks provide by lint, Google's static analysis tool for Android apps [GOOGLE 14f]. When developing apps in Eclipse using the Android SDK plug-in, lint is run periodically on the developer's code. Additionally, the developer may choose to execute lint at any time and receive a console output detailing each problem found, as shown in Illustration 5. Any problems found are also highlighted in source code as warnings or errors. Most of the checks defined in the lint program manifest themselves as warnings, which may be either fixed or ignored. We hypothesized that running lint on each of our apps might bring forth interesting results, but found that doing so would not be possible as it would require the Java source code. However, we could implement some of the checks ourselves in order to use them in our study. This led us to the adoption of several of the metrics used in our work.



0 errors, 1 warnings

Description	Category	Location
⚠ The resource R.string.hello_world appears to be unused	Performance	strings.xml:6 in values (EmployeeListAp

Illustration 5: A warning provided by the lint program.

Research on software quality for mobile apps proved to be scarce. Most of the work done in this realm focuses on best practices, as discussed previously. However, we did find a paper from German researchers that suggests a model for mobile software quality [FRANKE 12]. Dominik Franke *et al*. discuss the models of Boehm and ISO, citing them as too general to be completely effective for mobile applications. The task of isolating the characteristics desirable for a mobile app from an existing model can be time consuming, and thus detrimental to business goals. From previous models coupled with concerns specifically related to mobile applications, the authors identify seven quality attributes as part of their framework: usability, data persistence, efficiency, flexibility, adaptability, portability, and extensibility. Many of these are also present in previous models, but have new implications in mobile systems. For example, efficiency is concerning due to battery consumption, which is not a factor in desktop systems. The "data persistence" attribute is new to the quality model. A specific rationale discussed is the increased complexity of the lifecycle of mobile applications, with a desire to persist data between states. The authors also performed a small case study in which they evaluated two note-writing Android applications: one with a high market rating, and one with a low rating. As expected, the highly rated app performed much better in terms of the model attributes. The authors give no exact methodology for the measurement of these attributes, preferring to stick to the abstract. Much like the traditional models of software quality, this one suggests general attributes, but nothing measurable. However, it does guide the evaluator in the development of metrics.

There is no shortage of research on the Android platform. Studies cover topics ranging from an evaluation of the platform against other mobile operating systems [KAUR 14] to malware detection techniques [WEI 12] to the presentation of specific applications [KHAN 14]. We knew early that we wanted to perform a study on a large body of Android apps, much like in Symon's study. Although we were unable to find any that focused specifically on quality, we did identify several works that considered a body of apps. Most of these evaluated security, examining the permission system.

In 2012, Pern Hui Chia *et al.* performed a study to determine if any risk signals for malicious apps could be found [CHIA 12]. Their study was based around the principle of least authority, which states that an app should be granted only as much privilege as is absolutely necessary. Collecting 20,500 new and 650 popular Android apps, the researchers proceeded to analyze correlation between number of permissions requested and a set of information collected from the marketplace. Included in this information is number of downloads, number of installations, price, and market rating. Although they did not find any reliable signal of risk, they did come to several important and interesting conclusions. First, users are trained to accept permission requests from apps, in many cases without even reading them. Without any reliable risk indicators, it is difficult for a user to assess whether an application really needs the permission to perform its task. Some familiarity with the permission system can mitigate, but not solve this problem. The study also found that, in general, popular apps request more permissions. A number of reasons may explain this, including richer feature sets, but still the possibility of over-permissioning is worrisome. Another conclusion of the study is that free apps request more permissions than paid apps. This implies that developers may be using malicious means to monetize, especially since paid apps generally have more functionality. Finally, they found that "look-alike" apps request more permissions than is typical. A "look-alike" app is a knock-off designed to look and perform like a more popular one in the hopes of tricking users into downloading them instead, or encouraging download due to a lower price. Sometimes these apps will vary only slightly in name, and look and feel nearly identical. However, the increased permission level with the same functionality implies over-permissioning. Gathering demographic information from the marketplace was a technique that we knew we wanted to include in our study. From the HTML source of the app page, we planned to gather the market rating, number of downloads, and number of installations of the app. While the authors did not present their technique or source code for this, they did demonstrate to us that us was a feasible task. Additionally we learned that there is no statistically significant correlation between number of permissions requested and market rating in the body of apps examined. However, this did not mean that permissions are not important in determining the quality of an app, as there are many other ways to analyze permissions, and market rating is not a direct measure of quality.

Chia *et al*. state that a major limitation of their study was a lack of source and binary code analysis. A prior study based around such analysis was performed by Adrienne Felt *et al*. in 2011 with the goal of determining the effectiveness of Android permissions [FELT 11]. In this study the authors collected a body of 856 free and 100 paid apps from the official Android Market. They analyzed the frequency of dangerous permission use, finding that they were requested in 93 percent of free and 82 percent of paid apps. A dangerous permission is defined as those that allow potentially harmful API calls, such as accessing a network. By far the most requested permission was the INTERNET permission, found in 86.6 percent of free and 65 percent of paid apps. After collecting data on the permissions used, the authors turned to a source code analysis in an attempt to find over-privileged apps. For this analysis, they selected 18 free and 18 paid applications – one from each category in the Android Market. Four of the 36 were found to request permissions unnecessarily, with INTERNET being the most common unused permission. In order to address the problem, the authors suggest a tool that can detect unused permissions and warn the developer as well as the end user. It could be used simply as a warning, or integrated into the application packaging process such that binaries could not be created until the requested permissions matched those used. The authors also suggest improving permission granularity, such that each corresponds to only one action rather than a subset.

Since we wanted to perform an analysis across several markets, we looked for work that also did so. We found one such study from 2012, also focusing on security and permissions. Yajin Zhou *et.al* collected 204,040 apps from the Android Market and four third party markets [ZHOU 12]. They designed a tool called DroidRanger to perform analysis on the security of these apps. First, they looked for known patterns of malware using both the permissions requested and the dex byte-code. DroidRanger also had the ability to detect previously unknown malware through the use of heuristic-based filtering and dynamic execution monitoring. The first heuristic was helpful in furthering our understanding of the Android platform. An .apk file usually contains a classes.dex directory which holds all the byte-code for the app. However, an application can load code from another source such as a .jar or .apk file using the DexClassRunner class. In their study, the authors considered any code that made use of this class suspicious. The second heuristic is the use of native code, which directly exposes system calls to the underlying OS. Next, the researchers applied their detection techniques to each of the applications, finding that apps from third party markets had a higher rate of infection – out of 211 infected apps, 179 came from the external markets. Making this even more significant, only 25.02 percent of apps in the study were downloaded from third-party markets. This implies that unofficial markets are generally more dangerous to the end user. Zhou's study was similar to ours in that we both wanted to assess the health of several Android markets. It demonstrates that there is indeed, as expected, academic value in such a study. However, we wanted to focus on quality to make this assessment, rather than malware.

The lack of work on quality assessment for a large body of Android applications confirmed our belief that such a study would be valuable to the academic community. However, the work done on related topics was instrumental. Armed with the knowledge presented by past researchers, we were ready to begin our study.

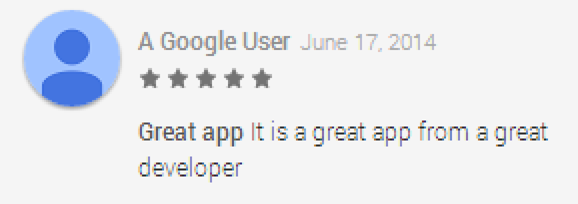
CHAPTER 3 – Solution

Methodology

After determining our focus of examining .apk files in order to assess the quality of a large body of applications as well as the marketplaces from which they came, we began working toward this goal. A first step was to determine which markets to use in our study. Ideally we would be able include Google's official Play Store alongside several third party markets, in order to make the task of comparison more meaningful. However, gaining access to applications from Google's store proved to be infeasible.

We identified Google Play as a prime target for our study for several reasons. This is by far the largest and most used source for Android applications, with over a million published [WELCH 13]. These apps represent a wide array of categories, including business, education, lifestyle, sports, and games. Apps can also be sorted by cost, average user rating, and time of publication. Additionally, the majority of devices sold are compatible and come with the market preinstalled, making it the most accessible. In fact, using other markets requires the explicit permission of the user.

Google Play also provides for each app some metadata which we identified as potentially useful in the assessment of quality. In addition to the ratings, as shown previously in Illustration 1, the market also shows the total number of ratings that the app has received. Google integrates with its social network, Plus, by showing the number of people who have recommended the app using the platform. All of these could potentially be used as surrogates for quality, with the assumption that a higher number of downloads implies a better app. Alongside the market ratings are individual reviews, in which users, identified by their Google accounts, can write a description of their experience with the app. Also displayed in the review is the rating given to the app by that particular user – the user must assign a rating in order to leave a comment. Illustration 6 demonstrates a comment on an app's page. Others can mark comments as helpful, unhelpful, or inappropriate, with the idea that the most helpful rise to the top and inappropriate comments are removed. Other information includes the date of the last update, number of installs, and content rating. Illustration 7 shows this "additional information".

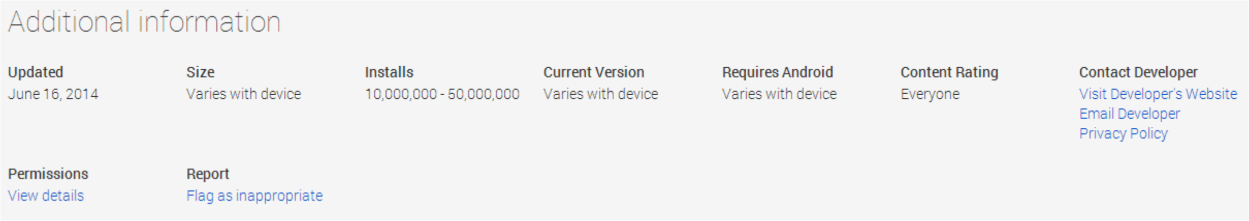


A Google User June 17, 2014

★★★★★

Great app It is a great app from a great developer

Illustration 6: A random user comment for a random app



Additional information

Updated	Size	Installs	Current Version	Requires Android	Content Rating	Contact Developer
June 16, 2014	Varies with device	10,000,000 - 50,000,000	Varies with device	Varies with device	Everyone	Visit Developer's Website Email Developer Privacy Policy
Permissions	**Report**					
View details	Flag as inappropriate					

Illustration 7: Some of the Metadata provided by Google Play for a random app

Much of the data mentioned can be found directly in the HTML code. This means that the collection of data could be easily automated through the use of a web scraper, looking for specific tags using string matching, a custom parser, or regular expressions. Given the name of an app, we could find its market entry, collect relevant data, and store it in a database, all in an automated fashion. In addition, there is a third party Android Market API [THIEL 11] that has the ability to obtain the ratings for an application based on the package name. This software provides a convenient interface to interact with the Google Play store using Java (though it can be adapted to other languages), and would eliminate the need to write custom web scrapers. Metadata accessible includes all of the previously discussed items.

Despite the attractiveness of using the Google Play Store in our study, we found several major barriers in the way of doing so. Symon ran into similar issues in his study, ultimately landing on the use of third party markets [SYMON 12]. First, there is no easy way by which to download an .apk file directly to a PC. Most likely out of a desire to prevent such action, the market is designed such that the .apk files are difficult to access directly. A possible solution is to download the app to a device first, using the Android debug bridge (adb), a command line tool that allows communication with a connected Android device or emulator, to transfer the file to the PC. This presents a problem not in the transfer, but in the downloading of the files. It is very difficult to automate the downloading of apps, which we needed to do due to the large quantity of apps needed for our study.

In addition to its difficulty, accessing a Google service in an automated fashion is also against the company's terms of service [GOOGLE 14g]. One option was to attempt to contact Google directly, but we did not pursue this path. Instead we applied for a Google Faculty Research Award [GOOGLE 14h] that requested not only funding for our work, but also access to a body of .apk files from the Play Store. Google declined to support the research.

Because of the restrictions on the use of Google's official market, we decided to focus our work on third party markets. Though none of these have achieved a user base as large as Play, many have seen success. Our next task was to identify a number of markets to analyze. We established several criteria for doing so. First, and most importantly, we needed markets that distributed .apk files directly, such that they could easily be downloaded to a PC. Also beneficial was the organization of the files such that automation was relatively easy. URL schemes and

HTML code made this easy to analyze. An ideal market would allow us to navigate among pages of applications through a simple change in the URL, and to download the file using a link found in the HTML source.

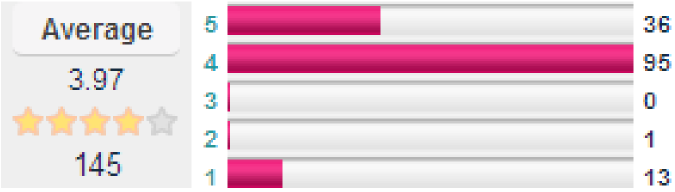
We identified several other constructs that might be restrictive to easily downloading an .apk file to a PC. Some markets require the creation of a user account in order to download apps. Using these would add a level of complexity to the collection of apps, requiring us to authenticate our automation techniques. Some markets used captchas or other methods of deterring automated downloads, which would prevent us from easily downloading a large body of apps. Others explicitly forbade downloading an app directly to a PC in their terms of service, whether from the market or from a device using adb.

Our second criteria for an ideal market was that it should contain apps from a variety of categories. Several markets exist that distribute apps only in a particular domain. For instance, 9game (www.9game.com) is devoted specifically to games. We wanted to get a general overview across all apps, so we determined that using a market that focused on only one category would be restrictive. In a similar vein, certain markets contained only or mostly paid apps. We did not wish to acquire paid apps without proper payment, and as we had no budget to obtain them, we chose to examine only free apps. This meant that we needed markets with large collections of free apps.

Our last criteria in market selection was that we wanted at least one market with quality-related metadata similar to that found in Google Play. Most important of these was an app rating system that allowed users to assign a score to the app. We needed a market that displayed the average rating, preferably along with the total number of ratings received. Number of downloads was also a desirable attribute. In addition, we needed a market in which the rating system was actually used meaningfully. We encountered several markets that had very few ratings submitted for each app. Other data such as comments, version information, and content rating, though potentially useful, was not imperative to our study.

With our criteria in mind we began the search for third party markets. First we examined those used by Symon in his study. Since our works builds on his, and he identified very similar criteria for market selection, it would be ideal to use the same markets. In addition to being able to reuse his web crawlers, we could also recollect the demographic data from his study and see how the markets have changed over time. Symon selected the following markets: Apps For Adam (http://appsforadam.tk), And App Online (https://www.andapponline.com), App Town (www.apptown.com), and Slide Me (http://slideme.org). Out of these, we found only Slide Me was still a viable candidate. Apps For Adam was blocked by the University malware detection software, and as such we decided to avoid it. And App Online no longer distributed .apk files directly, and as such it would be difficult to collect apps in an automated fashion. Finally, App Town no longer existed as a market, though there did exist a Dutch app development company with the same name.

Slide Me proved to be an ideal market for our study for several reasons. First, the market consists of a collection of .apk files that can be downloaded directly, with an easily navigable URL scheme. We were able to collect apps using Symon’s web crawler with only very minor modifications. Another strength of the market is that is has a rating similar very similar to that of Google Play, as demonstrated in Illustration 8. Not only is the rating system present, but the market has a large enough user base that it is actually populated with meaningful data. A comment system similar to that of Play is also present, though we found that in general only the most popular apps had received any feedback. Illustration 9 shows an example. From the HTML source we could gather the average rating, number of ratings, and number of installations, among other data. The data was placed in very specific tags, meaning that the collection process could easily be automated. Finally, Slide Me contained a large variety of apps and did not restrict accessing the site in an automated fashion.



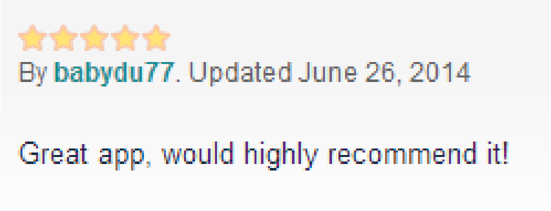
Average

3.97

145

5	36
4	95
3	0
2	1
1	13

Illustration 8: Market Rating for a Random Slide Me App



By babydu77. Updated June 26, 2014

Great app, would highly recommend it!

Illustration 9: Comment for a Random Slide Me App

In addition to Slide Me, we found two markets that met our needs: F-Droid (https://f-droid.org/), a catalog of free and open source (FOSS) apps, and Apps Apk (http://www.appsapk.com/), a market that distributed Android apps as freeware. Both of these were mostly collections of .apk files without the metadata found in Google Play and Slide Me. However, since our criteria was that we needed only one market with such data, we chose to use these markets in our study. Both made the automated crawling and downloading of .apk files fairly straightforward.

With our markets selected, we began the process of collecting our bodies of free apps. We wanted to gather as many free apps as possible from each site. In order to accomplish this goal we wrote three targeted web crawlers, one for each market. Each crawler would begin on a specified web page and parse it for any links leading to apk files. A regular expression was used to detect any links ending in ".apk" in the context of particular HTML tags, which differed by market. Links that would not lead to any download were ignored. When a link was found to an apk file, the program would download the file to the hard drive of the PC. After the entire HTML source for a page had been considered, the crawler moved on to the next page, based on a specified pattern that we observed and programmed. For instance, in Apps Apk the page number is part of the URL, so http://www.appsapk.com/android/all-apps/page/10/ gives the tenth page of apps. We started with 1 and incremented this number, continuing until the page was not found. The source code for these and all other programs written for this research were written in Python and can be found at the end of this document in Appendix 1.

Our crawler produced just over 18,000 apps for examination. In order to perform our work we removed those that had non-unicode characters in the app name, as they would be difficult to process in our programs. The later metric collection pruned away some apps that were missing resources. In the end we examined 17,867 apps in our study. Illustration 10 shows a breakdown by market.

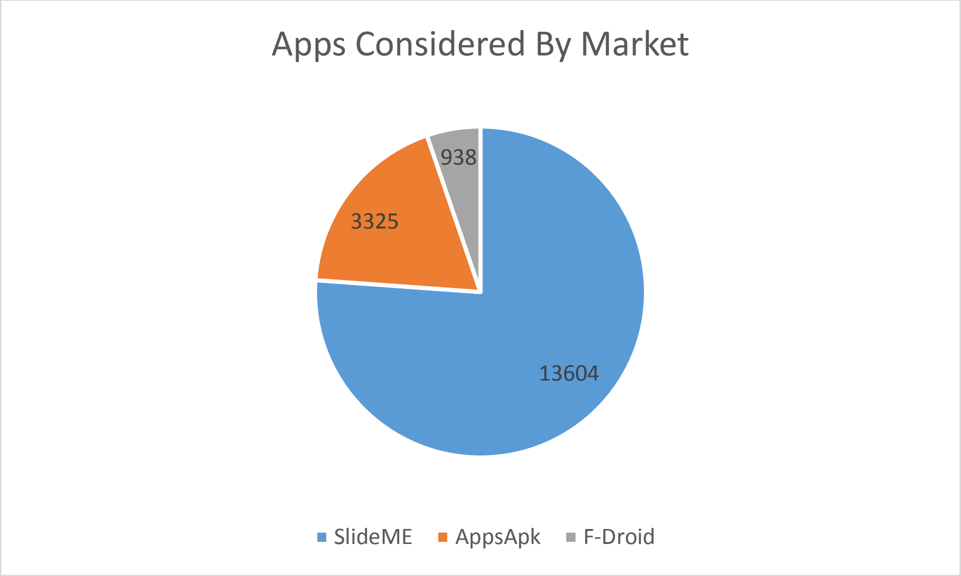


Illustration 10: Apps Considered in Our Study, by Market

Next we began the process of reverse engineering the apps in order to produce artifacts to be used in assessing quality. We wrote a Python script to consider each .apk file that we downloaded, and reverse engineer it using apktool from the command line. Program code can be found in Appendix 1.

With our body of apps in place and disassembled, we were ready to begin collecting data. In order to do this, we needed to identify the quality metrics we planned to use. A key concern was that we planned to analyze artifacts that could be gathered from the .apk file, meaning that source code analysis would not be possible. However, Dex byte-code, as well as the AndroidManifest.xml file and application resources were available for analysis.

An apk file holds five main components: the META-INF and res directories, and the classes.dex, AndroidManifest.xml, and resources.arsc files. Illustration 11 demonstrates this breakdown. Other directories may be present, but are not required in every app.

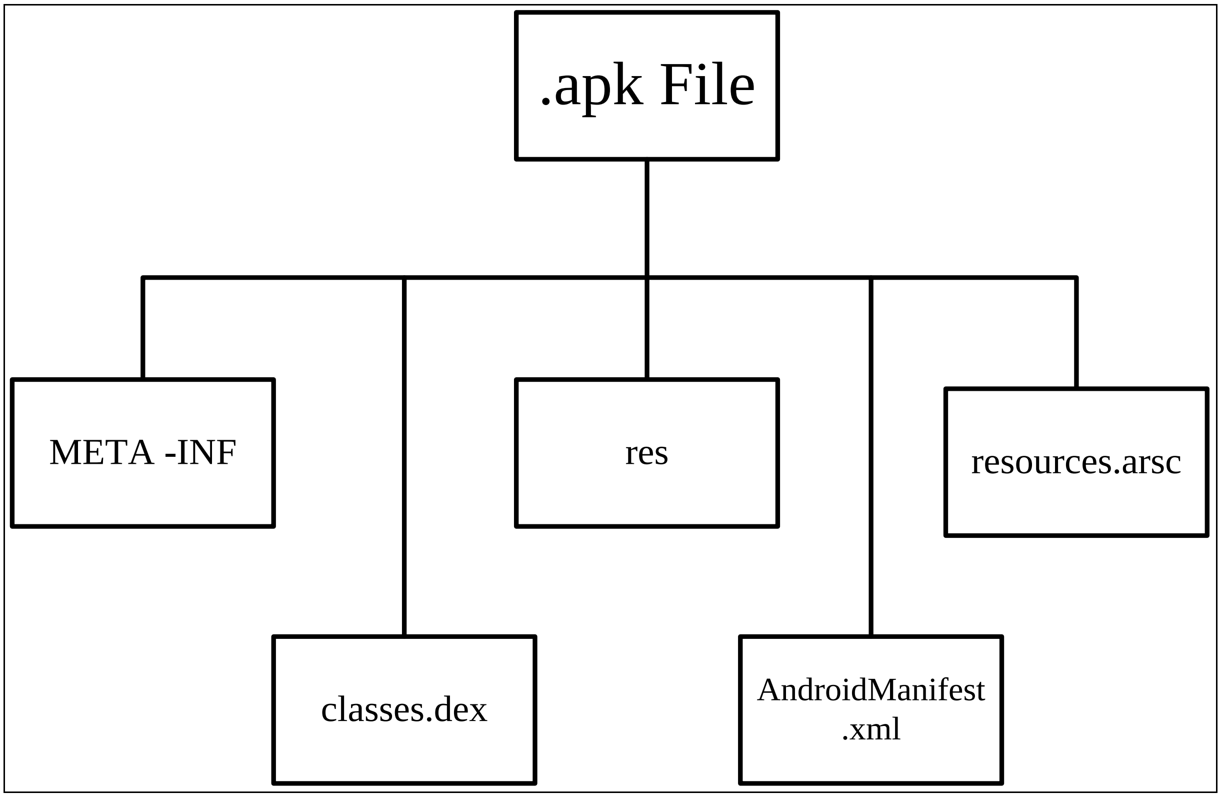


Illustration 11: apk File Structure [Symon 12]

META-INF contains information about the author. Within the META-INF directory are three files: MANIFEST.MF, CERT.SF, and CERT.RSA. The MANIFEST.MF and CERT.SF files establish a mapping between names and SHA1-Digests, and the CERT.RSA file holds the certificate information required to package and distribute an app. This is a binary file, but the others are human-readable. However, none of them contain information that we deemed useful in determining the quality of an app.

The res folder holds app resources that are not compiled into resources.arsc. A variety of different resources can be used, and the resource directory contains a set of directories corresponding to each. Examples include the drawable, layout, and values folders. Drawable holds image or XML files that represent a particular shape to be drawn on a device's screen. The layout folder contains XML files, generally one per view, that describe how the objects on the screen are to be arranged. Values can hold several different types of resources in the form of XML files. For example, a strings.xml file, which declares string literals with assigned names, is found in the values directory. Using this file decouples the Strings from the implementation, centralizes the location of String literals, and allows for localizations. Resources from the res directory are compiled into the R class of an app, and thus are access through this class rather than directly through the filename. The resources.arsc file contains information about these resources.

AndroidManifest.xml is a required file that contains key information including the app's name and version number. The file must be present at the root of the app and properly formed in order to run the app. Many tags exist to declare different attributes, but there are several that are required of all apps. These include name, version number, minimum SDK level, activities used, configurations used, compatible screens, and permissions used [GOOGLE 14i]. Illustration 12 shows the required format of the file. The manifest file has the quality that it must be well formed. This means that it can be easily processed using an XML parser. In the apk file, the manifest is stored in a binary format; however, apktool has the ability to reproduce the well-formed human-readable artifact.

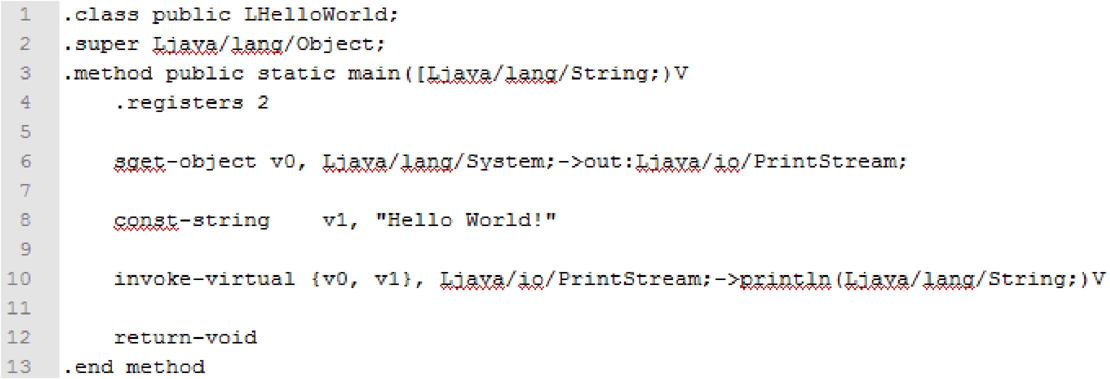
```
<?xml version="1.0" encoding="utf-8"?>
<manifest>

    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application>

        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
```

```
      </intent-filter>
      <meta-data />
    </activity>

    <activity-alias>
      <intent-filter> . . . </intent-filter>
      <meta-data />
    </activity-alias>

    <service>
      <intent-filter> . . . </intent-filter>
      <meta-data/>
    </service>

    <receiver>
      <intent-filter> . . . </intent-filter>
      <meta-data />
    </receiver>

    <provider>
      <grant-uri-permission />
      <meta-data />
    </provider>

    <uses-library />

  </application>
</manifest>
```

Illustration 12: The Required Structure of the AndroidManifest.xml File [Google 14i]

The final component in the apk file is the classes.dex file. Within this compressed directory is all of the Dalvik byte-code for the application, organized by package. Each file corresponds directly to a Java source file, with additional files automatically generated. When an application is loaded, the instructions from these files are run on the Dalvik Virtual machine. Reverse engineering this folder produces a “smali” directory that lists each file in package structure. Files are given the .smali extension, from the disassembler tool used [SMALI 14a]. Since we wanted to focus mainly on code in our quality analysis, this was the most important part of the apk file. Our next task was to understand the syntax of the byte-code in order to get an idea of what metrics we could collect.

Illustration 13 demonstrates a simple program in smali format. The program is a single class with a single method that prints “Hello World” to standard output. In smali, all classes are declared with the `.class` directive. Only lines declaring a class will begin with these characters. Likewise, the `.method` directive declares a method. In the example, the class is named “HelloWorld” and the method present is “main”, taking one String as a parameter. The appended ‘L’ denotes that a class path is to follow, a pattern that is followed consistent throughout [CRACKING 14]. In line 4, the `.registers` directive declares that two registers are used in the program. These are `v0`, and `v1`, which are local registers. There are registers beyond these, some of which are parameter registers (which can also be used as further local registers). Table 2 shows each of these, as specified in [SMALI 14b]. Line 6 assigns the `PrintStream` object from the java.io package to the `v0` register. The following line assigns the String literal, “Hello World” to register `v1`. Line 10 then calls the `println()` method of the `PrintStream` class using the `invoke-virtual` directive. All methods have a return statement, even if the method does not return anything, as indicated by line 12. Finally, all methods are closed with the `.end` directive. We consider byte-code further in our discussion of metrics.



```
1   .class public LHelloWorld;
2   .super Ljava/lang/Object;
3   .method public static main([Ljava/lang/String;)V
4       .registers 2
5
6       sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
7
8       const-string    v1, "Hello World!"
9
10      invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V
11
12      return-void
13  .end method
```

Illustration 13: A hello world program in smali byte-code

Register		Description
v0		the first local register
v1		the second local register
v2	p0	the first parameter register
v3	p1	the second parameter register
v4	p2	the third parameter register

Table 2: The first five registers used in Dalvik byte-code [SMALI 14b]

With our body of apps in place and decompiled, and an understanding of the artifacts produced, our next step was to determine the quality metrics that we would collect. Our only constraint was that they must be gatherable from the reverse engineered apk files. We had no source files to examine – only byte-code. Overall, this did not turn out to be especially limiting. The Dalvik byte-code correlates very closely to the Java source, and is presented in a manner that is sufficiently readable. An example of a limiting factor is the lack of comments in byte-code – these are removed in the compilation process. We examined several sources and found metrics that we thought might be useful in determining quality. Originally we were going to focus only on traditional white-box metrics, such as those related to program size. However, our results did little to differentiate apps based on quality and as such we turned to new metrics. These were Android-specific metrics designed to measure attributes related to user perception of the apps. In source code analysis we did not consider packages that were part of the SDK and always included. The following sections present each of the metrics we used.

Size Metrics

Size metrics measure the program in term of code size and modularity. These can be traced back to the 1960s, when the Lines of Code (LOC) metric was commonly used to measure productivity [FENTON 00]. As a code base grows, its readability, maintainability, and testability are diminished. Following are the size metrics we considered.

Number of Byte-code Instructions. A count of the executable instructions. These can be identified in byte-code as lines that do not begin with '#' (generated comments) or '.' (method, class, or field declarations). For each app, we counted the total number of instructions across all classes.

Number of Classes. A count of the number of classes in the developer's package. Classes can be identified in byte-code using the `.class` directive.

Number of Methods. A count of the number of methods across all classes. Methods can be identified in byte-code by the `.method` directive.

Methods per Class (MPC). The quotient of number of methods divided by the number of classes.

Instructions per Method (IPM). The Quotient of number of byte-code instructions divided by the number of methods.

Complexity Metrics

This metric measures program code in terms of its complexity. A more complex program has more possible paths of execution, so it is difficult to test fully. More complex programs are also more difficult to understand and maintain.

Cyclomatic Complexity. Thomas McCabe, in [MCCABE 76] introduced a metric to measure program complexity based on possible paths of execution. His strategy is to construct a program graph based on the flow of information in the program. Once built, the number of regions in the graph, including the area outside the graph, is the cyclomatic complexity. Illustration 14 gives an example.

```
1 method:
2 int x = 0
3 while(x != 0):
4    x = userInput()
5    if(x < 0):
6         print x * -1
7    else:
8         print x
```

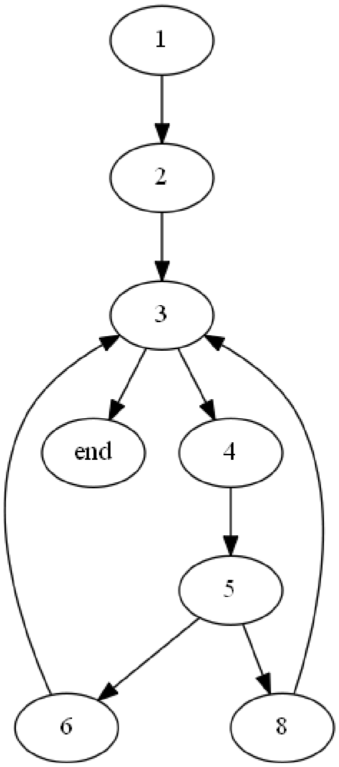


Illustration 14: A sample program and graph. The cyclomatic complexity is 3.

Another method by which to determine the cyclomatic complexity is to count the number of conditional statements in the code considered and add one. Conditional statements are any lines of code that could lead to more than one next path, such as if, for, and while in C-based languages. We used this approach in our analysis, because Dalvik bytecode has a finite set of conditionals. Table 3 shows each of the conditional operator and its meaning. In addition, a ‘z’ can be added to each, indicating ‘zero’. We calculated the cyclomatic complexity of each method and found the average for each app.

Conditional	**Meaning**
if-eq	if equal
if-en	if not equal
if-le	if less than or equal
if-lt	if less than
if-ge	if greater than or equal
if-gt	if greater than

Table 3: Conditional Statements in Dalvik Byte-Code [TITELBAUM 12]

Object Oriented Metrics

The following metrics are used in order to assess conformance to object oriented design principles, such as modularity, low coupling, high cohesion and encapsulation. Unless otherwise noted, each of these comes from Chidamber and Kimerer's "A Metric Suite for Object Oriented Design" [CHIDAMBER 94].

Number of Children (NOC). A count of the number of methods children for a given class. In byte-code the parent of a class is identified by the `.super` directive. We used this to build a tree data structure in memory representing the class hierarchy. In order to reduce the data to a single value per app, we considered the maximum number of children for any class in the app.

Depth of Inheritance Tree (DIT). We considered the depth each inheritance tree, found using the `.super` directive as discussed in the previous metric. In order to reduce the data to a single value per app, we considered the deepest tree for any class in the app.

Lack of Cohesion of Methods (LCOM). Cohesion refers to how related methods and fields of a class are. Ideally cohesion is high, because a well-designed class will have a very specific task that all parts work to achieve. In order to measure this, we used the following technique from NASA [NASA 95]: Calculate the number of access to each data field in a class, and find the average. Subtract from 100 to obtain a percent lack of cohesion. Lower percentages indicate greater cohesion of data and methods, so a lower score is desirable. Fields are denoted in byte-code by the `.field` directive, which is followed by its visibility, as demonstrated in Illustration 15. They are accessed using the fully qualified class name and the `->` operator, as shown in Illustration 16. We calculated the average score among each class for this metric.

```
.field public hours:I
```

Illustration 15: The declaration of a public integer field named hours.

```
iget v0, p0, Lcom/example/employeelistapp/HourlyEmployee;->hours:I
```

Illustration 16: Accessing the hours field uses the -> operator.

Coupling Between Objects (CBO). Coupling is the degree to which one class depends upon another. It is generally desirable to avoid coupling. We counted the number of calls that invoked methods outside of the calling class's inheritance tree.

Percent Public Instance Variables (PPIV). Thomas McCabe suggests this metric in [MCCABE 84]. It is simply the percentage of public and protected fields in a class. We calculated the average among all classes.

Access to Public Data (APD). Also introduced in [MCCABE 84], this metric is the number of accesses to public and protected fields. It is simply the percentage of public and protected fields in a class. We calculated the average number of accesses per class.

View Metrics

These metrics assess the use android layout files and view objects. A layout for a particular screen may be defined in one of two ways. It may be declared in an XML file in the layout directory, or defined in a Java class. View objects are those that are seen in a particular view, such as a button or a text box.

Separation of View and Controller (SVC). We developed this custom metric to assess the conformance of each app to the model-view-controller (MVC) architectural pattern. In this pattern, commonly used for user-interface software, there are three components. The model contains the data for the application. A view presents something to the user, generally based on the model, and allows interaction. Finally, the controller implements the program logic, mediating the interaction between the view and the model. [BASS 13] Android development is based around this pattern. Activities are the controllers, plain old Java objects (POJO) comprise the model, and the previously discussed mechanisms (XML or Java) comprise the view. A fundamental tenet of the pattern is that views should not be defined in controllers. We compiled a list of view objects from the Android.widget package [GOOGLE 14j], and used it to determine where view objects were defined. We measured the percentage of views not defined in controllers.

Average Number of Views per XML Layout File. Lint [GOOGLE 14f], Google's static analysis tool, checks the number of views in each layout file. The reasoning is that a large number (over 80) will result in poor app performance. We found the average number of views across all layout files in each app.

Maximum Number of Views in an XML Layout File. For similar reasoning as in the previous metric, we determined the maximum number of views in an XML file in each app.

Android-Specific Metrics

Originally we intended only to include the previously presented metrics in our study. However, upon data collection we found that these did little to differentiate apps based on market rating. We shifted our focus away from the white box metrics and toward those that directly influence the user's experience. The following sections detail each of these metrics.

Unchecked Bundles. On any given day, one in five Android users will experience a crash, and up to half will uninstall the offending app [BUGSENSE 14]. This certainly could incite the user to assign a low score. Research has shown that the most common reason for app crashes is the occurrence of a NullPointerException [KECHAGIA 14]. Many of these are left uncaught and thus cause crashes. The NullPointerException is often manifested in the implementation of the application lifecycle. In order to pass data among views, the developer can use a custom object, the Intent, to wrap data. The developer must call the `putExtras()` method of the Intent class, which creates a Bundle, another custom object, to wrap the data. From the receiving end, there is a class-level Intent. A Bundle can accessed from the Intent using the `getExtras()` method. In some cases this method might return null [GOOGLE 14k], meaning that if left unchecked it may lead to a crash. Thus, the developer should handle the possibility of the exception being thrown when using Bundles from Intents to prevent the app from crashing. This can be accomplished using a try-catch block or an if statement. We counted the number of unchecked Bundles in each application.

Bad Token Exceptions. In Android development, the Context object is used to store information regarding, as expected, the current context of the application [GOOGLE 14l]. This is necessary to perform certain actions, such as showing a notification. In order to show a dialog, the context of an Activity must be used. Using any other context, such as that of a service, will throw a WindowManager.BadTokenException [SMITH 14]. We counted the number of dialogs shown from classes other than activities, which might lead to crashes.

Number of Fragments. The Fragment was added to the Android SDK level 11. It represents a behavior or a portion of user-interface in an Activity. This can be combined to create a fragmented interface. [GOOGLE 14m] Since these are new to the platform, we determined that they might be useful in determining quality, as developers may have trouble adjusting to new techniques. We counted the number of Fragments in each application.

Bad Smell Method Calls. In [KECHAGIA 14], this authors identify 10 methods in Android development that commonly throw exceptions, and therefore to crashes. Using these methods without a try-catch block is a bad smell – it does not necessarily mean that there will be a failure, but it indicates that one could occur. We counted the number of unchecked calls to each in each app. Table 4 shows each method and the exceptions they might throw.

Method	Exceptions
Dismiss	IllegalArgumentException, NullPointerException
Show	WindowsManager.BadTokenException, IllegalStateException, InflateException
setContentView	InflateException
createScaledBitmap	IllegalArgumentException, NullPointerException
onKeyDown	IllegalStateException
isPlaying	IllegalStateException
unregisterReceiver	IllegalArgumentException
onBackPressed	IllegalStateException
showDialog	WindowManager.BadTokenException
Create	Resources.NotFoundException

Table 4: Top 10 methods causing crashes [KECHAGIA 14]

Battery Metrics

One problem unique to mobile devices is battery life. Certain actions tend to drain battery faster than others. A Google IO talk from 2008 [GOOGLE 09] let us to develop several battery-related metrics, which we detail in the following sections.

WakeLocks with no timeout. A WakeLock is used to keep the screen from turning off automatically after a given period of time. These require particular consideration because they must be released manually, and remain if the application crashes. However, the developer may set a timeout so that the WakeLock is released after a specified period. [GOOGLE 14n] We counted the number of WakeLocks with no timeout.

Number of Location Listeners. An Activity may extend the LocationListener and implement a set of methods that keep track of the user's location. Using these services on a device increases battery consumption. [GOOGLE 09] We counted the number of LocationListener activities in each app.

Number of GPS Uses. There are three ways to get the user's location: using the cellular service, WIFI, or the built-in GPS. Of these, the GPS uses the most battery power. [GOOGLE 08] We counted the number of GPS uses in each app.

XML Parsers. In parsing XML, a developer has several choices. Some parsers are tree based, meaning they bring the entire document into memory in a tree data structure. These are known as Document Object Model (DOM) parsers. Others are event based, meaning each tag fires a callback, and the entire document is not saved in memory. Event based parsers are faster, and use less battery power. [GOOGLE 09] We counted the uses of a tree based parser (DOMParser) and two event based parsers (XMLPullParser and SAXParser).

Network Timeouts. When performing a networking operation, the developer may set timeouts to prevent hanging. There are two types of timeouts: socket timeouts and connection timeouts. Connection timeouts set a limit on the amount of time the app can take to establish a TCP connection; Socket timeouts set a limit on the amount of time an app can keep the connection open without receiving any data. We counted the number of apps using each. [APACHE 14]

Component Launches

An activity may launch a new component, such as another activity or a service. There are several types of components, as well as several ways in which they might be launched. For example, the `startActivity()` method simply launches a new view. The `startActivityForResult()` does the same, with the expectation that the developer will send a value back. We counted the use of each of the component launch methods, to get an idea of what components were used. Our reasoning was that apps using components beyond activities were more complex. Table 5 lists each method (without parameters).

Method Name
startActivities
startActivity
startInstrumentation
startIntentSender
startService
startActionMode
startActivityForResult
startActivityFromChild
startActivityFromFragment
startActivityIfNeeded
startIntendSenderFromResult
startIntentSenderFromChild
startNextMatchingActivity
startSearch

Table 5: Component Launch Methods

ANR Metrics

If an app becomes unresponsive to user input, the system displays an “Application Not Responding” (ANR) message. The user is given a choice to wait for the system to become responsive or force close the app. An ANR message is shown if there is no response to a key press within 5 seconds, or a `BroadcastReceiver` hasn't finished executing within 10 seconds. [GOOGLE 14q] In [YANG 13], the authors identify four common causes of ANR messages, all involving long running tasks on the main (user interface) thread. We used these to develop our metrics.

Networking. Network operations tend to introduce latency. When on the main thread, these can lead to ANR. An API change in level 11 added the `NetworkOnMainThreadException`, which causes any networking operating on the UI thread to crash [GOOGLE 14r]. However, we found that many apps still perform such operations on the main thread. We found the percentage of all network operations that were performed in the main thread for each app.

File IO. Android uses flash memory for its file system, which makes accesses expensive [YANG 13]. As such, accesses should be performed in a background thread. We found the percentage of all file IO operations that were performed in the main thread for each app.

SQL Lite. Android ships with a toolkit for SQL Lite, a database system similar with SQL syntax [GOOGLE 14s]. The data is stored on device, and tends to be expensive to use in terms of latency. We found the percentage of all SQL lite operations that were performed in the main thread for each app.

Bitmaps. Decoding bitmaps is a computationally expensive task on the Android platform [YANG 13]. We found the percentage of all calls to `BitmapFactory.decode()` that were performed in the main thread for each app.

Black Hole Exception Handling

We use the term “black hole exception handling” to refer to exception handling with no corrective action. The developer places code in a try-catch block, but leaves the catch block empty, or only logs the error. We calculated the percentage of catch blocks that took no corrective action. Since it would not be possible to determine for sure if an action was corrective, we assumed any code other than logging instructions to be corrective.

Demographic Metrics

In addition to the quality metrics, we also decided to examine several items of demographic nature. Our rationale was that these might be useful in differentiating apps based on quality. Many of the metrics used come from [SYMON 12].

Use of Android Objects. We decided to compare the usage of Android SDK objects across apps, to see if any distinction could be made in terms of quality. In byte-code, these objects are preceded by `Landroid/` and thus easy to find with a regular expression. We compiled a list of objects used by each app.

SDK Level. The SDK level determines what features and programming constructs are available. We collected the minSDKLevel and targetSDKLevel from the manifest file. minSDKLevel is the lowest API level that can run the app, and targetSDKLevel indicates the highest level that the app has been tested on. Any device with an API level higher than the targetSDKLevel (if declared) will not use compatibility features when running the app. The minSDKLevel tag is required, while the targetSDKLevel tag is not. [GOOGLE 14o]

APK File Size. The size of the .apk file has an effect on download time, and takes more memory on a device. The authors of [KIM 13] found that download time increases approximately linearly with file size. Users with slower connections will see even higher times, potentially leading to frustration. Google limits APK file size to 50MB (4GB with expansion files) [GOOGLE 14p].

Number of String Resources. Rather than hard-coding strings, the developer may declare each by name in a strings.xml resource file. This has the advantage of decoupling the Strings from the implementation as well as allowing for localizations.

Localizations. Localizations come in two forms: layout localizations and string localizations. Layout localizations allow the developer to define views for different screen sizes and orientations. String localizations allow the developer to support different languages. We counted the number of string and layout localizations in each app.

CHAPTER 4 – Validation
Results

Our aim in performing our research was to determine if we could measure app quality from the implementation, specifically an .apk file, and use this to measure market quality as well. In order to validate that we did indeed find a way to measure quality, we needed to determine which metrics influenced quality and which did not. To do so we created two groups of apps from the SlideMe market: the highest rated 1,000 and the lowest rated 1,000. We compared the average metric values among the groups, looking for discrepancies. From this, we determined which metrics were most closely correlated with user ratings. We used these to compare the markets. We present our findings in the following sections. Additional data can be found in Appendix 2.

Illustration 17 shows the breakdown of market ratings from the 10,740 Slide Me apps. In the lower 1,000 apps, ratings ranged from 0.5 to 2.0, whereas the highest rated apps were all scored at 5.0. More than 1,000 had a rating of 5.0 – we simply selected 1,000. We also collected metadata such as number of downloads, which we do not include here.

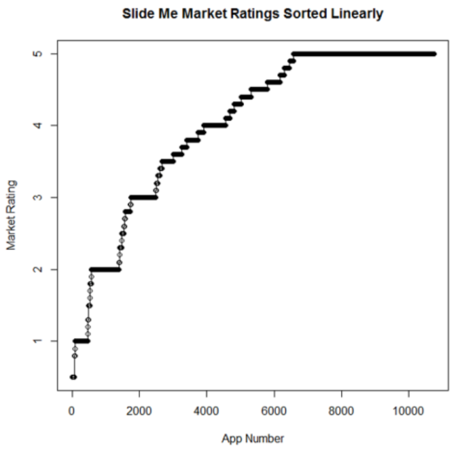


Illustration 17: Slide Me Ratings Sorted Linearly

In order to assess the relationship between market ratings and metric scores, we need a numerical way to describe the correlation. We define a Difference Index (DI) for each metric *m*, as follows:

$$DI_m = \left| \frac{AVG_{Top}}{AVG_{Bottom}} - 1 \right| * 100$$

Equation 1: Difference Index (DI)

A DI of 0 indicates that there is no difference between the score in the high-rated group and that of the low-rated group for a given metric. A high DI means a large difference between scores, indicating a correlation between that metric and the app rating. In the following sections we present the values and the Difference Index for each metric.

Size Metrics. Table 6 shows the average value in each group for the size metrics, along with each Difference Index.

Metric	Top 1,000	Bottom 1,000	Difference Index
Num Byte-Code Instructions	9093.44	8969.58	1.38
Num Methods	340.55	352.88	3.5
Num Classes	64.45	65.00	0.84
MPC	4.84	5.25	7.89
IPM	28.67	28.75	0.25
Cyclomatic Complexity	6.69	6.36	5.17
WMC	10.52	10.61	0.81

Table 6: Results for Size Metrics

Object Oriented Metrics. Table 7 shows the results for object oriented metrics.

Metric	Top 1,000	Bottom 1,000	Difference Index
NOC	1.05	0.79	32.23
DIT	1.40	1.37	2.15
LCOM	46.79	47.37	1.22
CBO	3.25	3.36	3.10
PPIV	17.40	19.07	8.75
APD	6.02	6.24	3.54

Table 7: Results for Object Oriented Metrics

The traditional software quality metrics did little to differentiate the groups. Most produced a low DI, with the highest coming from Number of Children. This was the only metric with a DI above 10. As mentioned previously, this led us to pursue other metrics.

View Metrics.

Metric	Top 1,000	Bottom 1,000	Difference Index
SVC	93.2	90.54	2.94
Avg Num XML Views	7.31	6.37	14.75
Max Num XML Views	20.94	16.67	25.6

Table 8: View Metrics

View metrics, as shown in Table 8, also did not prove to be strongly correlated with user ratings. Almost all apps followed the MVC pattern very closely, and few used a large enough number of views in an XML file to affect performance.

Unchecked Bundles, Bad Token Exceptions, and Fragments.

Metric	Top 1,000	Bottom 1,000	Difference Index
Unchecked Bundles	1.16	2.10	44.76
Potential Bad Token	.0209	.0235	11.06
Number of Fragments	.0546	.0323	69.04

Table 9: Unchecked Bundles, BadTokenExceptions, and Fragments

Two of the metrics shown in Table 9 produced high DI values. Unchecked Bundles were more present in the lower rated apps, suggesting that application crashes lead to poor ratings. Number of fragments also showed a high correlation, suggesting that staying up-to-date with SDK features leads to higher ratings.

Bad Smell Method Calls.

Method	Top 1,000	Bottom 1,000	Difference Index
Dismiss	1.85	1.65	12.12
Show	9.14	10.11	9.59
setContentView	0.62	2.97	79.24
createScaledBitmap	0	0	0
onKeyDown	0.001	0.03	96.25
isPlaying	0.49	0.55	10.9
unregisterReceiver	0.05	0.03	66.67
onBackPressed	0.001	0.01	81.13
showDialog	0.02	0.05	61.91
Create	1.88	1.93	2.59

Table 10: Bad Smell Method Calls

Several of the bad smell method calls, shown in Table 10, demonstrated large DI values. Of the ten methods, five produced a DI larger than 60. This further exemplified a relationship between application crashes and poor ratings.

Battery Metrics.

Metric	Top 1,000	Bottom 1,000	Difference Index
No Timeout WakeLocks	0	0	0
Num Location Listeners	.06	.09	11.06
DOM Parsers	0.23	0.07	69.04
SAXParser	0.07	0.06	16.67
XMLPullParser	0.01	0.04	75
No Network Timeouts	132	139	5.04

Table 11: Battery Metrics

Table 11 shows a comparison of battery metrics. These show that DOM Parsers are used more frequently in the higher rated group, while the XMLPullParser is used more in the lower rated group. While these metrics demonstrate high DI values, they do not suggest that battery consumption is correlated with user rating. XMLPullParser is faster and more battery efficient [GOOGLE 08]. However, many applications warrant the use of DOM parsers in order to keep the document in memory. We conclude that battery metrics are not useful in determining quality, but that DOM Parsers are indicative of high quality apps. We hypothesize that this is for performance reasons. DOM Parsers have more upfront cost and use more memory, but allow fast access once the tree has been built.

Component Launches.

Method	Top 1,000	Bottom 1,000	Difference Index
startActivities	0	0	0
startActivity	5.60	4.99	12.3
startInstrumentation	0	0	0
startIntentSender	0	0	0
startService	0.17	0.08	110.71
startActionMode	0.008	0.005	54.72
startActivityForResult	1.09	2.35	53.70
startActivityFromChild	0	0	0
startActivityFromFragment	0.0041	0.0021	95.24
startActivityIfNeeded	0	0.001	NA
startInentSenderForResult	0	0.002	NA
startNextMatchingActivity	0.01	0.005	92.45

Table 12: Component Launch Method Calls

As shown in Table 12, several components are launched less frequently in the higher rated group than in the lower rated group. However, several had high DI values because they are called more frequently in the higher rated groups. This implies that some components are associated with high-rated apps, while others are associated with low-rated apps.

ANR Metrics.

Metric	Top 1,000	Bottom 1,000	Difference Index
Networking	0.09	0.05	80
File IO	0.09	1.0	11.11
SQL Lite	0.16	0.17	6.25
Bitmaps	0.22	0.15	46.7

Table 13: ANR Metrics

Of the four ANR metrics, shown in Table 13, two proved to be significant in terms of DI. Networking produced the highest value. However, the network accesses occurred on the main thread more frequently in the top rated group. This goes against our hypothesis that networking on the main thread would cause lower ratings due to ANR dialogs. Since networking on the UI thread in no way indicates quality, we looked further into the issue. Our analysis of the use of

Android objects in each app revealed that the WebView was more common in the lower group. 34,854 WebView uses occurred in the bottom group, versus 5,890 in the top group. Using the WebView in place of performing HTTP requests to populate custom views handles threading internally. As such, it stands to reason that the higher-rated group showed more networking on the main thread simply because there was more networking.

Bitmap use on the main thread did demonstrate a notable DI. However, similarly with networking, more apps performed Bitmap operations on the main thread in the top group. Our data demonstrates no effect of ANR dialogs on app ratings.

Black Hole Exception Handling

Metric	Top 1,000	Bottom 1,000	Difference Index
Percent Black Hole	0.69	0.73	5.47

Table 14: Black Hole Exception Handling

As shown in Table 14, black hole exception handling was not correlated with user ratings. However, a large percentage of catch blocks took no corrective action. While these try-catch statements do prevent crashes, the lack of corrective action demonstrates poor engineering practice.

Demographic Metrics

Metric	Top 1,000	Bottom 1,000	Difference Index
.apk File Size	8.85	15.94	74.62
Number of Strings	66.03	46.48	42.08
Layout Localizations	280	217	29.03
Language Localizations	288	334	15.97
targetSDKLevel	382	676	82.7

Table 15: Demographic Metrics

Two of the demographic metrics, shown in Table 15, proved to be useful in our analysis. The top group of apps had on average smaller .apk files and more string resources. A large file size means the app takes longer to download and run [KIM 12]. This could lead to user frustration. The string resources allow easier development through decoupling and allow for the support of many languages.

From the above data we determined the 15 metrics with the highest DI. These are the metrics that we determined are most closely correlated with user ratings. Since user ratings are intended in part to measure app quality, we conclude that these metrics are correlated with Android app quality. However, we make this conclusion with an understanding that the user rating system is not perfect, and that many other factors reflect quality, regardless of ratings. Illustration 18 shows the metrics with the highest DI.

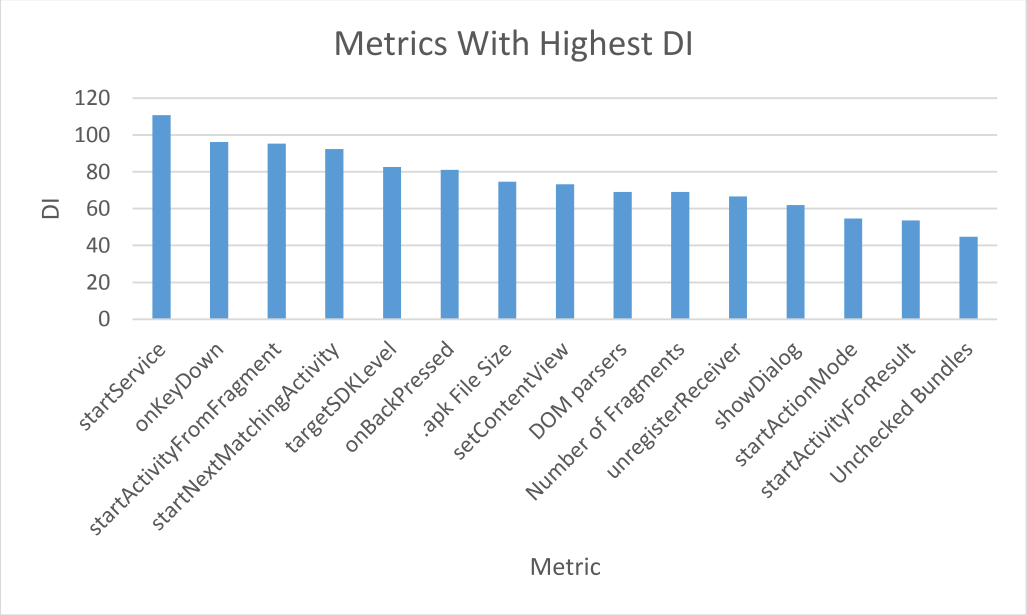


Illustration 18: Metrics with Highest Difference Index

Comparison of Markets

With the above metrics, we have determine a method by which to measure the quality of a mobile application. Our next goal was to determine if we could measure the quality of an application market. To do so, we compared the apps within each using the top 15 metrics to determine any differences.

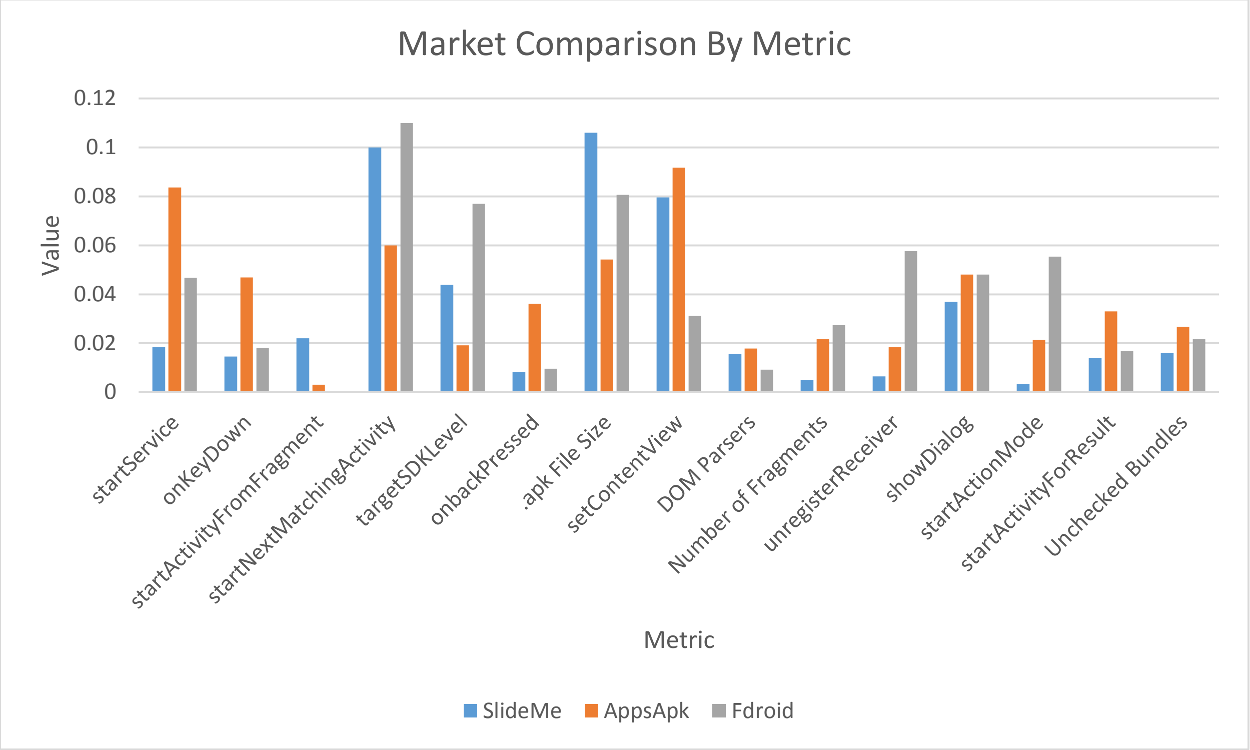


Illustration 19: Metric comparison by market. Values for certain metrics have been scaled by a power of 10 in order to use a consistent scale. We aim to show relative values in order to compare markets.

Illustration 19 presents a comparison of the average metric values for each market. Some of these values were modified to fit the scale, however, the comparison remains the same. In order to determine which market was of the highest quality, we determined which market had the most desirable value for the highest number of metrics (in some cases, a higher value was desirable, and in others, a lower value). F-Droid had the most desirable value for 1 metric, Apps Apk for 6, and Slide Me for the remaining 8. From this we conclude that Slide Me is the market of highest quality. This matched our expectations for several reasons. Of the three, Slide Me is the most widely used, with the largest selection of apps. It is the only one with a market rating system, and submitted apps go through a review process. Therefore it follows that the other markets are of lower quality.

The difference between F-Droid and Apps Apk brings up the topic of closed-source versus open-source quality. Both are similar in that they are just sites that distribute .apk files without the marketplace and community features that many others have. However, F-Droid, which ranks as lower quality in our study, contains only open-source apps. This may suggest that open-source apps are of lower quality. However there are many other factors that could explain this difference.

Taxonomy of Bad Smells

In order to further demonstrate the results of our study, we developed a taxonomy of bad smells for Android apps. Here we use bad smells specifically in terms of market ratings. That is, the smells indicate patterns that could lead to low ratings. Illustration 20 graphically represents our taxonomy. Most strongly correlated with user ratings are app crashes, followed by complexity of components used, and poor performance. While these were not the only significant factors in our study, they were the most prominent in the 15 metrics with the highest DI values. We suggest our taxonomy as a guide to developers who wish to improve app ratings. We offer further suggestions in the following discussion.

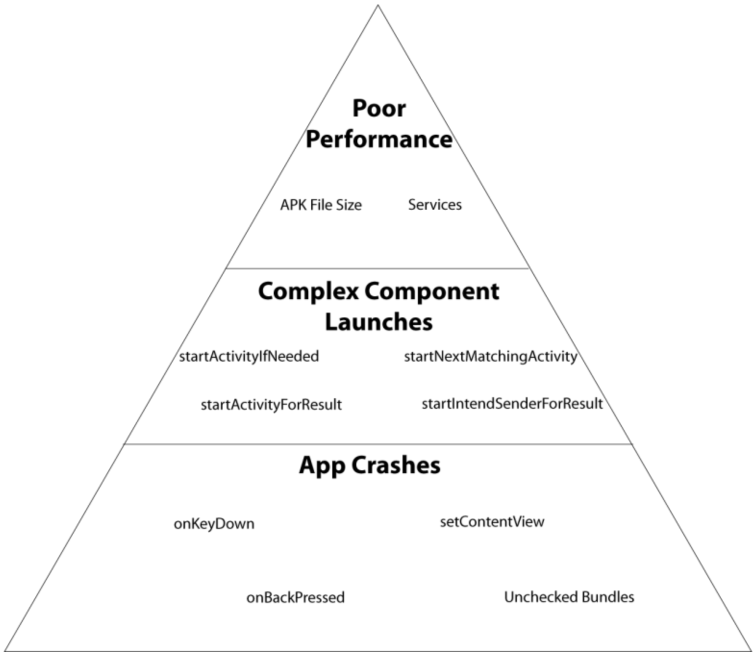


Illustration 20: A taxonomy of bad smells for Android Apps.

Discussion of Results

We aimed to gather useful information from our results. We first pursued the following research question: *Which coding practices should be followed, and which should be avoided, in order to create a high quality app?* By examining the results of our study, we created the following guidelines for Android developers:

1. Avoid app crashes. Of the 15 metrics with, the highest DI, 7 were related to crashes. Developers should be aware of unchecked exceptions and handle them through conditional logic or try-catch statements.
2. Use services to move computation to the background thread. The metric with the highest DI was the use of the startService method, which was called more frequently in apps with higher ratings. Services are used for long running computations, keeping the workload off of the main thread [GOOGLE 14t]. Apps that keep these long-running tasks on the UI thread may receive lower ratings due to delay.
3. Select application components and launch methods strategically. 5 of the 15 highest DI values came from component launch metrics. Some methods, such as startService, were correlated with high user ratings, while others, such as startNextMatchingActivity were correlated with low ratings.
4. Make use of fragments. Both the number of fragments and the startActivityFromFragment metrics showed high DI values. Fragments allow for more flexible and maintainable user interfaces that support many screen sizes [GOOGLE 14m].
5. Test thoroughly when using target SDK level. Apps declaring a targetSDKLevel in the manifest tended to be rated lower. Since a target SDK level allows features from a higher SDK level than the minimum, it is necessary for the developer to ensure backward compatibility [GOOGLE 14o].

6. Always check Bundles from Intents to ensure they are not null. Null values could lead to NullPointerExceptions. Research has shown that this is one of the most common exceptions thrown by Android apps.
7. Avoid producing excessively large apps. Larger .apk files are correlated with lower ratings, likely due to the increased time to download, install, and run the apps.

Our results show that white box traditional quality metrics, including those to measure size and compliance to object-oriented principles, showed little correlation with application ratings. However, research has shown that these metrics are still indicative of software quality [BASILI 96]. Users generally have no way to perceive these factors, and focus more on those that affect their experiences. The traditional metrics are often more useful for developers, indicating factors such as maintainability and readability. Since users assign market ratings, it follows that they are more closely related to the metrics observable in the finished product.

The lack of correlation between traditional quality metrics and app ratings represents a fundamental disconnect in the community rating systems. In order to answer the question, *How can the app rating system be improved such that an app's rating is more directly related to its quality?*, we suggest the following.

1. Combine user ratings with an additional rating from a third party organization, similar to Consumer Reports[1]. This would help reduce user bias and improve consistency as the organization would have rigid guidelines regarding assessing the quality of an app.
2. Require download before a user can rate an app. Without this constraint, a user could intentionally skew the app's score by rating it without ever using it. Some markets implement this policy, while others do not.

[1] http://www.consumerreports.org/cro/index.htm

3. Require the user to have the app installed for a specific amount of time before rating it. This ensures that the user has had sufficient time to evaluate the quality of the app. Alternately, the user could be required to use the app for a specific amount of time. This would become problematic, however, for apps that crash on launch.
4. Require a certain number of ratings before the rating becomes visible to the public. This ensures an app's rating is not determined by a small group of users.
5. Tie accounts to organizations and individuals. This will force users to be more conscientious about their rating habits due to loss of anonymity. Some markets do this already, such as Google Play.
6. Run static analysis on submitted apps to check for indicators of low quality. Unchecked Bundles and unchecked calls to bad smell method calls, for example, could easily be detected by such an analysis. These checks could also be incorporated into lint, and the developer could then be made aware and required to fix the problem before the app is published.
7. Provide number of downloads, number of ratings, and number of active downloads to all users. Some markets do this whereas others do not.

Our collection of metrics has shown several ways by which quality can be measured from a reverse-engineered .apk file using only static analysis. Many metrics showed a strong correlation with application ratings, which are also intended to measure quality. While they are not a perfect measurement, the correlation is still significant, as higher ratings will influence more downloads.

Limitations and Threats to Validity

Here we discuss the limitations of our study as well as the factors that pose a threat to the validity of the study. The most obvious limitation of the study is the fact that the rating system is designed to demonstrate user-perceived quality of apps. That is, traditional quality metrics are superseded by user experience. We agree with this notion, but we contend that the lack of correlation between traditional metrics and actual ratings represents a fundamental disconnect between the user’s perception of quality and true app quality. However, there is no way to differentiate apps using traditional metrics without defining arbitrary values deemed good and bad. As such we used market ratings, which reflect upon user perception – one facet of quality, to differentiate apps. Since software quality is abstract and subjective, the use of market ratings as a measurement is appropriate.

Another threat to the validity of our study is that we did not examine the primary market for Android apps (Google Play) due to accessibility limitations. However, we analyzed apps from several markets including a large, well known market with 10,740 apps available. Due to the large size and number of users, we feel that our results can be generalized to other Android app markets, including Google Play.

CHAPTER 5 – Conclusions and Future Efforts

Conclusions

We had three main goals in performing this research. First, we wanted to determine a method by which to measure the quality of an Android app based on the contents of its .apk file. Second, we sought to develop a method by which to measure the quality of a marketplace as a whole. Finally, we wished to use this mechanism to compare several app marketplaces. We collected 17,687 apps from three different third-party markets, and reverse engineered them using custom software that made use of an open source tool. We were successful in our endeavors through the use of several quality metrics that we collected on our body of apps. In order to determine which metrics were most indicative of quality, we developed a method by which to quantify the difference in the average values for two groups of apps: The highest rated 1,000 apps and the lowest rated 1,000 apps from the Slide Me market. From this we were able to come to several conclusions about what metrics affect user ratings, and thus quality, most directly. The metrics most related to app quality were those involving app crashes. Included in these are unchecked Bundle Intents and unchecked calls to specific bad smell methods. Metrics related to components and component launch methods also showed a strong correlation to app quality. Another prominent category were those related to performance, such as the size of .apk files. Other factors that showed strong correlation with app ratings were the use of WebViews, Fragments, and the declaration of a target SDK level.

We used the metrics most strongly correlated with user rating in order to measure the quality of the three markets from which we collected apps. To do this we compared the average value in the market for each of the 15 metrics that showed the strongest correlation with ratings. Our results show that Slide Me is the highest quality marketplace, Apps APK the next highest, and F-Droid is the market of lowest quality.

Future Efforts

As we performed our research, several ways by which it might be extended became apparent. Most obvious is the inclusion of other metrics. Our list of metrics is in no way complete. Many others may accurately measure the quality of an Android app. Over time, this work could be used to collect a large group of metrics known to influence app quality, which could be used as a guide for developers. Our work could also be used to measure the quality of an app or market over time. Changes in development techniques and user patterns may lead to significant changes. Watching the quality of an app or market over time could allow the maintainer to become aware of any problems. On a related note, our metrics could be used by the owner of an app market to prune away low quality apps.

Another possible research direction for the future is the automated detection and correction of bad smells. We have identified several patterns in code that show a correlation with low user ratings. These patterns could be automatically detected by a tool, similar to lint, and corrected. A specific example that lends itself well to this approach is unchecked calls to bad smell methods. These could easily be wrapped in a try-catch block by a custom tool. Alternatively, checks could be added to existing tools.

Finally, we focused our research on third party markets due to accessibility limitations. Performing our study on the official Google Play market would allow us to see how it compares to other markets in terms of app quality. This would bring up an interesting comparison of official versus third party markets.

References

[AL-BADAREEN 11] Al-Badareen, A. B.; Selemat, M. H.; Jabar, M. A.; Din, J.; Turaev;, "Software Quality Models: A Comparative Study", *Communications in Computer and Information Science*, vol. 179, pp. 46-55, 2011.

[ALVERO 10] Alvero, A.; Santana de Almeida, E.; Meira, S. R.;, "A Software Component Quality Framework," *ACM SIGSOFT Software Engineering Notes*, vol. 35, iss. 1, pp. 1-18, Dec. 2010.

[AMAZON 14] Amazon App Store http://www.amazon.com/mobile-apps/b?node=2350149011

[APACHE 14] Apache API – HttpClient, https://hc.apache.org/httpclient-3.x/apidocs/org/apache/commons/httpclient/HttpClient.html

[APPLE 07] Apple Reinvents the Phone with iPhone https://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html

[APPLE 13] Apple's App Store Marks Historic 50 Billionth Download http://www.apple.com/pr/library/2013/05/16Apples-App-Store-Marks-Historic-50-Billionth-Download.html

[APPLE 14a] App Review https://developer.apple.com/support/appstore/app-review/

[APPLE 14b] iOS Developer Program https://developer.apple.com/programs/ios/

[APPLE 14c] Introducing Swift https://developer.apple.com/swift/

[AQUA 13] App Quality Alliance. "Best Practice Guidelines." http://www.appqualityalliance.org/files/AQuA_best_practices_doc_v2%202_FINAL_5_feb_2013.pdf

[BARRERA 10] Barrerra, D.; van Oorschot, P.C.;, "Secure Software Installation on Smartphones" http://people.scs.carleton.ca/~paulv/davidb.pdf

[BASILI 96] Basili, V.R.; Briand, L.C.; Melo, W.L.:, "A validation of object-oriented design metrics as quality indicators.", *Software Engineering, IEEE Transactions on* , vol.22, no.10, pp.751-761. 1996.

[BASS 13] Bass, L.; Clements P.; Kazman, R.;, *Software Architecture in Practice* (3rd ed). Addison-Wesley, Upper Saddle River, NJ. 2013.

[BOEHM 73] Boehm, B. W.; Brown, J. W.; Lipow, M.;, "Characteristics of Software Quality", TWR Software Series TTW-SS-73-09. Dec. 1973.

[BOEHM 76] Boehm, B. W.; Brown, J. W.; Lipow, M.;, "Quantitative Evaluation of Software Quality", *Proceedings of the Second International Conference on Software Engineering*. Pp 592-605. 1976.

[BROOKS 87] Brooks Jr, Frederick P. "No Silver Bullet: Essence and Accidents of Software Engineering." *IEEE Computer*, vol. 20, no. 4, pp 10-19. Apr. 1987.

[BUGSENSE 14] Bugsense. Android Errors in Real Time, http://visual.ly/android-errors-real-time?utm_source=visually_embed

[CHIA 12] Chia, P.; Yamaoto, Y.; Asokan, N.;, "Is This App Safe?: A Large Scale Study on Application Permissions and Risk Signals", *Proceedings of the 21st International Conference on World Wide Web,* pp 311 -320, 2012.

[CHIDAMBER 94] Chidamber, S. R.; and Kemerer, C. F.;, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20, 1994.

[CRACKING 14] Android Cracking, http://androidcracking.blogspot.com/2010/09/examplesmali.html

[DUBEY 12] Dubey, S. K.; Ghosh, S.; Rana, A.;, "Comparison of Software Quality Models: an Analytical Approach", *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, iss. 2, Feb. 2012.

[FELT 11] Felt, A.; Greenwood, K.; Wagner, D.;, "The Effectiveness of Application Permissions", *Proceedings of the 2nd USENIX Conference on Web Application Development*, pp. 75 – 86, June 2011.

[FENTON 00] Fenton, N. E.; Neil, M.;, "Software metrics: roadmap", *Proceedings of the ACM Conference on The Future of Software Engineering (ICSE)*, pp. 357-370, 2000.

[FRANKE 12] Franke, D.; Kowalewski, S.; Weise, C.;, "A Mobile Software Quality Model", *2012 12th International Conference on Software Quality*, pp. 154-157, 2012.

[GAO 14] Gao, G.; Bai, X.; Tsai, W.; Uehara T.;, "Mobile Application Testing: A Tutorial", *IEEE Computer* vol. 74, iss. 2, Feb. 2014.

[GARVIN 84] D. Garvin, "What Does 'Product Quality' Really Mean?" Sloan Management Review, Fall 1984, pp. 24-45.

[GOOGLE 09] Sharkey, J.;, "Coding for Life – Battery Life that is", Google IO, 2009.

[GOOGLE 14a] Android Developer's Guide – System Permissions http://developer.android.com/guide/topics/security/permissions.html

[GOOGLE 14b] Google Play – Supported Devices https://support.google.com/googleplay/answer/1727131?hl=en

[GOOGLE 14c] Android Developer's Guide – Developer Registration https://support.google.com/googleplay/android-developer/answer/113468?hl=en

[GOOGLE 14d] Android Developer's Guide – Activities http://developer.android.com/guide/components/activities.html

[GOOGLE 14e] Android Developer's Guide – Best Practices http://developer.android.com/guide/practices/index.html

[GOOGLE 14f] Android Tools Project Site – Android Lint Checks http://tools.android.com/tips/lint-checks

[GOOGLE 14g] Google Terms of Service http://www.google.com/intl/en/policies/terms/

[GOOGLE 14h] Google Faculty Research Awards http://research.google.com/university/relations/research_awards.html

[GOOGLE 14i] The Android Manifest File http://developer.android.com/guide/topics/manifest/manifest-intro.html

[GOOGLE 14j] Android Developer's Guide – android.widget, http://developer.android.com/reference/android/widget/package-summary.html

[GOOGLE 14k] Android Developer's Guide – Intent, http://developer.android.com/reference/android/content/Intent.html

[GOOGLE 14l] Android Developer's Guide – Context, http://developer.android.com/reference/android/content/Context.html

[GOOGLE 14m] Android Developer's Guide – Fragments, http://developer.android.com/guide/components/fragments.html

[GOOGLE 14n] Android Developer's Guide – PowerManager.WakeLock, http://developer.android.com/reference/android/os/PowerManager.WakeLock.html

[GOOGLE 14o] Android Developer's Guide - <uses-sdk>, http://developer.android.com/guide/topics/manifest/uses-sdk-element.html

[GOOGLE 14p] Android Developer's Guide – APK Expansion Files, http://developer.android.com/google/play/expansion-files.html

[GOOGLE 14q] Android Developer's Guide – Keeping Your App Responsive, http://developer.android.com/training/articles/perf-anr.html

[GOOGLE 14r] Android Developer's Guide – android.os.NetworkOnMainThreadException, http://developer.android.com/reference/android/os/NetworkOnMainThreadException.html

[GOOGLE 14s] Android Developer's Guide – android.database.sqlite, http://developer.android.com/reference/android/database/sqlite/package-summary.html

[GOOGLE 14t] Android Developer's Guide – Services, http://developer.android.com/guide/components/services.html

[GSMA 12] Groupe Speciale Mobile Association. "Smarter Apps for Smarter Phones." http://www.gsma.com/technicalprojects/wp-content/uploads/2012/04/gsmasmarterappsforsmarterphones0112v.0.14.pdf

[HAIGH 10] Haigh, M.; Software quality, non-functional software requirements and IT-business alignment, *Software Quality Control,* v.18 n.3, p.361-385, Sep. 2010

[HALL 09] Hall, S.; Anderson, E.;, "Operating Systems for Mobile Computing," *Journal of Computing Sciences in Colleges,* vol. 25, no. 2, pp 64 -71 Dec. 2009

[HEGER 12] Heger, Dominique A.;, "Mobile Devices-An Introduction to the Android Operating Environment Design, Architecture, and Performance Implications", *DHTechnologies (DHT),* pp 43-49, 2012

[HIGA 08] Higa, D.; , "Walled Gardens versus the Wild West," *Computer* , vol.41, no.10, pp.102-105, Oct. 2008 doi: 10.1109/MC.2008.439
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4640677&isnumber=4640644

[IDC 13] Apple Cedes Market Share in Smartphone Operating System Market as Android Surges and Windows Phone Gains, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS24257413

[IDC 14] Worldwide Smartphone Market Grows 28.6 Year Over Year in the First Quarter of 2014, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS24823414

[ITU 14] The World in 2013: ICT Facts and Figures http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013-e.pdf

[ISO 11] International Organization for Standardization; Information Technology - Product Quality - Part1: Quality Model, International Organization for Standardization Standard 9126, 2001.

[JAMWAL 10] Jamwal, D.;, "Analysis of Software Quality Models for Organizations", *International Journal of Latest Trends in Computing*, vol. 1, iss. 2, pp. 19-23, Dec. 2010.

[KAUR 14] Kaur, P.; Sharma, S.;, "Google Android a Mobile Platform: A Review", *2014 Recent Adavnces in Engineering and Computational Sciences*, pp. 1-5, Mar. 2014.

[KECHAGIA 14] Kechagia, M.; Spinellis. D.;,"Undocumented and unchecked: exceptions that spell trouble", *Proceedings of the 11th ACM Working Conference on Mining Software Repositories*, pp. 312-315. ACM, New York, 2014.

[KHAN 14] Khan, A. U. S.; Qureshi, M.N.; Qadeer, M.A.;, "Anti-theft Application for Android Based Devices". *IEEE International Advance Computing Conference(IACC)*, pp. 365-369, 2014.

[KING 14] King Reports First Quarter 2014 Results
http://investor.king.com/files/doc_news/King%20Reports%20First%20Quarter%20Results.pdf

[KIM 12] Kim, S. R.; Kim, J. H.; Kim, H. S.;, "A hybrid design of online execution class and encryption-based copyright protection for Android apps", *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, pp. 342-343, 2012.

[KITCHENHAM 96] Kitchenham, B.; Pfleeger, S. L.;, "Software Quality: The Elusive Target" *IEEE Software*, vol. 14, no. 1, pp. 12-21, 1996.

[LOUIS 14] Louis, T.;, "How Much Do Average Apps Make?" Forbes.
http://www.forbes.com/sites/tristanlouis/2013/08/10/how-much-do-average-apps-make/

[LENDINO 14] Lendino, J. "Apple Debuts Swift Programming Language at WWDC" *PC Magazine,* http://www.pcmag.com/article2/0,2817,2458851,00.asp. June 2014.

[MCCABE 76] McCabe, T. J.; "A Complexity Measure", *IEEE Trans. Softw. Engr.*, SE-2, pp. 308-320, 1976.

[MCCABE 84] McCabe and Associates, "McCabe Object Oriented Tool User's Instructions", 1994.

[MCCALL 77] McCall, J.A.; Richards, P.K.; Walters, G.F.;, "Factors in Software Quality," Griffiths Air Force Base, N.Y. Rome Air Development Center Air Force Systems Command. 1977.

[NASA 95] NASA. “Software Quality Metrics for Object Oriented Environments”, Software Assurance Technology Center (SATC), http://www.literateprogramming.com/ootech.pdf, 1995.

[PETRASCH 99] Petrasch, Roland. "The definition of software quality: a practical approach."*Proceedings of the 10th International Symposium on Software Reliability Engineering*. 1999.

[REED 10] Reed, B.;, “A brief history of smartphones” http://www.networkworld.com/slideshows/2010/061510-smartphone-history.html

[RUBEY 68] Rubey, R. J.; Hartwick, D.;, “Quantitative Measurement of Program Quality,” *Proceedings of the ACM National Conference*, pp. 671-677, 1968.

[SCHOLTZ 04] Scholtz, J.; Steves, M. P.;, “A Framework for Real-World Software Systems Evaluation,” *Proceedings of the 2004 ACM Conference on Computer Supported Cooperated Work”*, pp. 600-603, 2004.

[SHINY 14] Shiny Development. Average App Store Review Times http://appreviewtimes.com/

[SMALI 14a] Smali Hello World Example http://code.google.com/p/smali/source/browse/examples/HelloWorld/HelloWorld.smali

[SMALI 14b] Registers https://code.google.com/p/smali/wiki/Registers

[SMITH 14] Smith, D.;, “Context, What Context?”, http://www.doubleencore.com/2013/06/context/, 2014.

[THIEL 11] Thiel, A., “An Open Source API for the Android Market”, https://code.google.com/p/android-market-api/, Feb 2011.

[TITELBAUM 12] Titlebaum, Dave, “Hacking APKs for Fun and Profit (Mostly for Fun)”, http://www.slideshare.net/davtbaum/hacking-for-fun-and-for-profit, Dec. 2012.

[WEI 12] Wei, T.; Mao, C.; Jeng, A.; Lee, H.; Wang, H;Wu, D.;, “Android malware detection via a latent network behavior analysis”, *IEEE 11th International Conference on. Trust, Security and Privacy in Computing and Communications (TrustCom),* pp. 1251–1258, 2012.

[WELCH 13] Welch, C.;, “Google: Android app downloads have crossed 50 billion, over 1M apps in Play” http://www.theverge.com/2013/7/24/4553010/google-50-billion-android-app-downloads-1m-apps-available

[WISNIEWSKI 12] Winsniewski, R.;, “Android – Apktool: A Tool for Reverse Engineering Android apk Files” http://code.google.com/p/android-apktool/ May 2012.

[YANG 13] Yang, S.; Yan, D.; Rountev, A.;, "Testing for poor responsiveness in android applications," Engineering of Mobile-Enabled Systems (MOBS), 1st International Workshop on the , vol., no., pp.1,6, 25-25. May 2013

[ZHOU 12] Zhou, Y,; Wang, Z.; Zhou, W.; Jiang, X.; “Hey, You, Get Off My Market: Detecting Malicious Apps in Official and Alternative Android Markets”, *Proceedings of the 19th Annual Network and Distributed System Security Symposium*. 2012.

Appendix 1 – Source Code

Slide Me Targeted Crawler

```
'''
Created on Jan 26, 2012

@author: symonwi
@author: Eric Shaw
'''
import urllib
import re

def getPageCount(url):
    applicationsPage = url+"/applications"
    numOfPagesExp = "<li class=\"pager-last last\"><a
href=\"/applications\?page=(\d+)\""

    site = urllib.urlopen(applicationsPage)
    html = site.read()
    site.close()

    numOfPages = int(re.findall(numOfPagesExp, html)[0])
    return numOfPages

def getHTML(url):
    site = urllib.urlopen(url)
    html = site.read()
    site.close()
    return html

def getApplicationPageLinks(html):
    appPageExp = "<h2 class=\"title\"><a href=\"(/application/.*?)\""
    return re.findall(appPageExp, html)

def getDownloadLinks(html):
    downloadLinkExp = "<div class=\"download-button\"><a href=\"(.*?)\""
    return re.findall(downloadLinkExp, html)

def downloadApp(url, path, appName):
    f = open(path+"slideme_"+appName+".apk", "wb")
    app = urllib.urlopen(url)
    f.write(app.read())
    f.close()
    app.close()

def getAppName(html):
    appNameExp = "<div class=\"download-button\"><a href=\".*?\" title=\"(.+?)\""
    return re.findall(appNameExp, html)[0].translate(None, "<>:\"/\|?*")


#############################################################################

domain = "http://www.slideme.org"
outputPath = "C:\\apks\downloaded\\slideme\\"
```

```
appsExamined = 0
appsDownloaded = 0




#this is where we will any errors we experience while downloading apps such as
#a problem with the formatting or if the app cost money to buy
errorFile = open(outputPath+"slidme_errors.txt", "w")

numOfPages = getPageCount(domain)

print "Downloading apps from "+str(numOfPages)+" pages . . ."
print "Scanning approximately "+str(numOfPages*10)+" apps for canidates . . ."

##for i in range(numOfPages+1):
for i in range(873,numOfPages+1):
    #set up the page url
    print "!!!!!!begining page #"+str(i)+" !!!!!!"
    currentPage = domain+"/applications?page="+str(i)

    #get the html code
    html = getHTML(currentPage)

    #extract the links to the actual application pages
    applicationPages = getApplicationPageLinks(html)

    #for each page parse to see if it is a free app and if so download
    for page in applicationPages:
        #get the html code
        html = getHTML(domain+page)
        #extract the links
        downloadLinks = getDownloadLinks(html)
        appsExamined += 1
        #give status update for each 100 downloaded
        if appsExamined % 5 == 0:
            print "****** "+str(appsExamined) + " Apps examined ******"

        #if no links there is a formatting error with the page
        if len(downloadLinks) > 0:
            downloadLink = downloadLinks[0]
            #if it doesn't end with .apk it's a pay app
            if downloadLink[-4:] == ".apk":
                #print domain+downloadLink
                downloadApp(domain+downloadLink,outputPath,getAppName(html))
                appsDownloaded += 1
                if appsDownloaded % 5 == 0:
                    print "****** "+str(appsDownloaded) + " Apps downloaded ******"
            else:
                errorFile.write("$$$$$$ Pay app found at "+domain+page+" $$$$$$\n")
        else:
            errorFile.write("XXXXXX Formating Error with page "+domain+page+"
XXXXXX\n")

errorFile.close()
print str(appsExamined)+" apps examined"
print str(appsDownloaded)+" apps downloaded"
```

Apps Apk Targeted Crawler

```
'''
Created on Jan 26, 2012

@author: symonwi
@author: Eric Shaw
@version: 1/21/2014
'''
import urllib
import re

def getPageCount(url):
    applicationsPage = url+"/android/all-apps/page/1"
    numOfPagesExp = "http://www.appsapk.com/android/all-apps/page/(\d+)/"

    site = urllib.urlopen(applicationsPage)
    html = site.read()
    site.close()

    list = re.findall(numOfPagesExp, html)
    numOfPages = int(list[-1])
    return numOfPages

def getHTML(url):
    site = urllib.urlopen(url)
    html = site.read()
    site.close()
    return html

def getApplicationPageLinks(html):
    htmlsub = html[html.index("<h1 class=\"page-title\">All
Apps</h1>"):html.index("class=\"pagenav clearfix\">")]
    appPageExp ="href=\"(.*?)\" title"
    #appPageExp = "<h2 class=\"title\"><a href=\"(/application/.*?)\""
    return re.findall(appPageExp, htmlsub)

def getDownloadLinks(html):
    downloadLinkExp = "href=\"(.*?)\.apk"
    ret = re.findall(downloadLinkExp, html)
    return ret

def downloadApp(url, path, appName):
    f = open(path+"appsapk_"+appName+".apk", "wb")
    app = urllib.urlopen(url)
    f.write(app.read())
    f.close()
    app.close()

def getAppName(link):
    appNameExp = "http://www.appsapk.com/(.*?)/"
    list = re.findall(appNameExp, link)
    return list[-1].translate(None, "<>:\"/\|?*")


##############################################################################

domain = "http://www.appsapk.com/"
outputPath = "C:\\apks\\downloaded\\appsapk\\"
appsExamined = 0
appsDownloaded = 0
```

```
#this is where we will any errors we experience while downloading apps such as
#a problem with the formatting or if the app cost money to buy
errorFile = open(outputPath+"appsapk_errors.txt", "w")

numOfPages = getPageCount(domain)

print "Downloading apps from "+str(numOfPages)+" pages . . ."
print "Scanning approximately "+str(numOfPages*7)+" apps for candidates . . ."

start = 1
for i in range(start,numOfPages+1):
    #set up the page url
    print "!!!!!!beginning page #"+str(i)+" !!!!!!"
    currentPage = domain+"/android/all-apps/page/"+str(i)

    #get the html code
    html = getHTML(currentPage)

    #extract the links to the actual application pages
    applicationPages = getApplicationPageLinks(html)

    #for each page parse to see if it is a free app and if so download
    for page in applicationPages:
        #get the html code
        html = getHTML(page)
        #extract the links
        downloadLinks = getDownloadLinks(html)
        appsExamined += 1
        #give status update for each 100 downloaded
        if appsExamined % 5 == 0:
            print "****** "+str(appsExamined) + " Apps examined ******"

        #if no links there is a formatting error with the page
        if len(downloadLinks) > 0:
            downloadLink = downloadLinks[0]

            #print domain+downloadLink
            downloadApp(downloadLink + '.apk',outputPath,getAppName(page))
            appsDownloaded += 1
            if appsDownloaded % 5 == 0:
                print "****** "+str(appsDownloaded) + " Apps downloaded ******"

        else:
            errorFile.write("XXXXXX Formating Error with page "+page+" XXXXXX\n")

errorFile.close()
print str(appsExamined)+" apps examined"
print str(appsDownloaded)+" apps downloaded"
```

F-Droid Targeted Crawler

```
'''
Created on Jan 26, 2012

@author: symonwi
@author: Eric Shaw
@version: 1/19/2014
'''
import urllib
import re

def getPageCount(url):
    applicationsPage = url+"/repository/browse/"
    numOfPagesExp = "Page 1 of (\d+)"

    site = urllib.urlopen(applicationsPage)
    html = site.read()
    site.close()

    numOfPages = int(re.findall(numOfPagesExp, html)[0])
    return numOfPages

def getHTML(url):
    site = urllib.urlopen(url)
    html = site.read()
    site.close()
    return html

def getApplicationPageLinks(html):
    appPageExp ="https://f-droid.org/repository/browse/\?fdid=(.*?)\""
    #appPageExp = "<h2 class=\"title\"><a href=\"(/application/.*?)\""
    return re.findall(appPageExp, html)

def getDownloadLinks(html):
    downloadLinkExp = "<a href=\"(.*?).apk"
    ret = re.findall(downloadLinkExp, html)
    return ret

def downloadApp(url, path, appName):
    f = open(path+"fdroid_"+appName+".apk", "wb")
    app = urllib.urlopen(url)
    f.write(app.read())
    f.close()
    app.close()

def getAppName(html):
    appNameExp = "<span style=\"font-size:20px\">(.*?)</span>"
    return re.findall(appNameExp, html)[0].translate(None, "<>:\"/\|?*")


#############################################################################

domain = "https://f-droid.org"
outputPath = "C:\\apks\\downloaded\\fdroid\\"
appsExamined = 0
appsDownloaded = 0




#this is where we will any errors we experience while downloading apps such as
#a problem with the formatting or if the app cost money to buy
```

```
errorFile = open(outputPath+"fdroid_errors.txt", "w")

numOfPages = getPageCount(domain)

print "Downloading apps from "+str(numOfPages)+" pages . . ."
print "Scanning approximately "+str(numOfPages*30)+" apps for candidates . . ."

start = 1 #change to recover from an error and start where you left off
for i in range(start,numOfPages+1):
    #set up the page url
    print "!!!!!!beginning page #"+str(i)+" !!!!!!"
    currentPage = domain+"/repository/browse/?fdpage="+str(i)

    #get the html code
    html = getHTML(currentPage)

    #extract the links to the actual application pages
    applicationPages = getApplicationPageLinks(html)

    #for each page parse to see if it is a free app and if so download
    for page in applicationPages:
        #get the html code
        html = getHTML(domain+ '/repository/browse/?fdid=' + page)
        #extract the links
        downloadLinks = getDownloadLinks(html)
        appsExamined += 1
        #give status update for each 100 downloaded
        if appsExamined % 5 == 0:
            print "****** "+str(appsExamined) + " Apps examined ******"

        #if no links there is a formatting error with the page
        if len(downloadLinks) > 1:
            downloadLink = downloadLinks[1]

            #print domain+downloadLink
            downloadApp(downloadLink + '.apk',outputPath,getAppName(html))
            appsDownloaded += 1
            if appsDownloaded % 5 == 0:
                print "****** "+str(appsDownloaded) + " Apps downloaded ******"

        else:
            errorFile.write("XXXXXX Formating Error with page "+domain+page+"
XXXXXX\n")

errorFile.close()
print str(appsExamined)+" apps examined"
print str(appsDownloaded)+" apps downloaded"
```

Decompiler

```
'''
Created on Jan 31, 2012

@author: Billy Symon
'''
import os
import os.path

count = 0
for root, dirs, files in os.walk("C:\\apks\\downloaded\\slideme"):
    for file in files:
        print "********************************************"
        os.system("apktool -v d -f \""+os.path.join(root, file)+"\" 
\"C:\\apks\\decompiled\\slideme\\"+file[0:-4]+"\"")
        count += 1
        print "********************************************"
        print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
        print "           "+str(count)+" files decompiled"
        print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
```

Driver

```
'''
This is the driver program for the collection of quality related metrics on a set of
reverse engineered .apk files.
Before running, the user should set up his database to match the expected schemas.  I
would recommend commenting out
the db.write methods at first to ensure the program runs properly.  Once the DB is set
up, only the extractData method
needs to be changed to hold the correct paths to your apps.
Created on Jan 22, 2014

@author: Eric Shaw
'''
import os
import os.path
import xml.dom.minidom as minidom
import ManifestDataExtractor as mde
import SizeMetrics
import CKMetrics
import MVCMetrics
import OtherMetrics
import BadSmellMethodCalls
import UncheckedBadSmellMethodCalls
import BatteryMetrics
import BlackHole
import NetworkTimeout
import ANRMetrics
import IntentLaunchMetrics
import DBWriter
import sys
import traceback

ignoreFiles = ['BuildConfig.smali', 'R$attr.smali', 'R$dimen.smali',
'R$drawable.smali', 'R$id.smali',
'R$layout.smali','R$menu.smali','R$string.smali','R$style.smali','R.smali']


def getSourceCodeDirectoryPaths(root, packageName):
    os.chdir(root)
    dirNames = os.listdir(os.getcwd())
    firstIdentifier = packageName.split('.')[0]
    secondIdentifier = packageName.split('.')[1]
    dirPaths = []
    dirPaths.append(root + "\\" + firstIdentifier + "\\" + secondIdentifier)
    return dirPaths

def parseManifest(manifestDataExt,filePath):


    #EXTRACTING APP LABEL
    appLabel = manifestDataExt.extractAppLabel()
    try:
        print "App Label -> " + appLabel
    except:
        appLabel = "DATA NOT FOUND - INVALID ENCODING"
        print "App Label -> " + appLabel

    #EXTRACTING FULLY QUALIFIED APP NAME
    appFQName = manifestDataExt.extractFQName()
    print "Fully Qualified Name -> " + appFQName
    return appLabel, appFQName



decFolderPath =  'C:\\apks\\EmployeeListApp_dec'#'C:\\apks\\EmployeeListApp_dec'

def extractData(location, market):
    db = DBWriter.DBWriter()
    db.connect()

    path = location+market
    files = os.listdir(path)
    directorySize = len(files)
    errorFile = open(path+"_"+market+"_metric_errors.txt", "w")

    print "!"*150
    print "NOW EXTRACTING DATA FROM " + path
    print str(directorySize)+" APPS FOUND"
    print "!"*150

    i = 1
    for f in files:
        sourceCodePaths = []
        layoutFilePaths = []
        decFolderPath = path + "\\"+ f
        filename = f + ".apk"
        #PRINTING HEADER
        print "*"*50
        print "EXTRACTING DATA FROM FILE "+ str(i) +" OF "+ str(directorySize)
        print "*"*50
        i += 1

        try:
            manifestDataExt = mde.ManifestDataExtractor(decFolderPath)


            if manifestDataExt.validateManifest():
                #get app name and package name from manifest
                appLabel, packageName = parseManifest(manifestDataExt,decFolderPath)

                #get path for each source code file that we will consider
                smaliPath = decFolderPath + '\\smali'
                dirPaths = getSourceCodeDirectoryPaths(smaliPath, packageName)
                #navigate using fully qualified packageName
                for codeDir in dirPaths:
                    for root, dirs, files in os.walk(codeDir):
                        for file in files:
                            if file not in ignoreFiles:
                                sourceCodePaths.append(os.path.join(root, file))

                print sourceCodePaths

                if len(sourceCodePaths) == 0:
                    #errorFile.write(f + ": No source code files\n")
                    continue

                layoutPath = decFolderPath + '\\res\\layout'
                for root, dirs, files in os.walk(layoutPath):
                    for file in files:
                        layoutFilePaths.append(os.path.join(root, file))

                print layoutFilePaths
                #calculate size metrics
                sizeMetrics = SizeMetrics.SizeMetrics(sourceCodePaths)
                sizeMetrics.extractData()
                numInstructions = sizeMetrics.getNumInstructions()
                numMethods = sizeMetrics.getNumMethods()
                numClasses = sizeMetrics.getNumClasses()
                methodsPerClass = sizeMetrics.getMethodsPerClass()
                instrPerMethod = sizeMetrics.getInstructionsPerMethod()
                cyclomatic = sizeMetrics.getCyclomatic()
                wmc = sizeMetrics.getWMC()

                #calculate CK metrics
                ckMetrics = CKMetrics.CKMetrics(sourceCodePaths, packageName)
                ckMetrics.extractData()
                noc = ckMetrics.getNOC()
                dit = ckMetrics.getDIT()
                lcom = ckMetrics.getLCOM()
                cbo = ckMetrics.getCBO()
                ppiv = ckMetrics.getPPIV()
                apd = ckMetrics.getAPD()

                #calculate MVC metrics
                mvcMetrics =MVCMetrics.MVCMetrics(sourceCodePaths, layoutFilePaths)
                mvcMetrics.extractData()
                mvc = mvcMetrics.getSepVCScore()
                avgNumViewsInXML = mvcMetrics.getAvgNumViewsInXML()
                maxNumViewsInXML = mvcMetrics.getMaxNumViewsInXML()
                potBadToken = mvcMetrics.getPotentialBadTokenExceptions()
                numFragments = mvcMetrics.getNumFragments()


                #calculate other metrics
                otherMetrics = OtherMetrics.OtherMetrics(sourceCodePaths,
layoutFilePaths)
                otherMetrics.extractData()
                uncheckedBundles = otherMetrics.getNumUncheckedBundles()
                objMap = otherMetrics.getObjectMap()

                #bad smell methods
                bsmc = BadSmellMethodCalls.BadSmellMethodCalls(sourceCodePaths,
layoutFilePaths)
                bsmc.extractData()
                show = bsmc.getNumShowCalls()
                dismiss = bsmc.getNumDismissCalls()
                setContentView = bsmc.getNumSetContentViewCalls()
                createScaledBitmap = bsmc.getNumCreateScaledBitmapCalls()
                onKeyDown = bsmc.getNumOnKeyDownCalls()
                isPlaying = bsmc.getNumIsPlayingCalls()
                unregisterReceiver = bsmc.getNumUnregisterRecieverCalls()
                onBackPressed = bsmc.getNumOnBackPressedCalls()
                showDialog = bsmc.getNumShowDialogCalls()
                create = bsmc.getNumCreateCalls()

                #checked bad smell methods
                cbsmc =
UncheckedBadSmellMethodCalls.UncheckedBadSmellMethodCalls(sourceCodePaths,
layoutFilePaths)
                cbsmc.extractData()
                cshow = cbsmc.getNumShowCalls()
                cdismiss = cbsmc.getNumDismissCalls()
                csetContentView = cbsmc.getNumSetContentViewCalls()
                ccreateScaledBitmap = cbsmc.getNumCreateScaledBitmapCalls()
                conKeyDown = cbsmc.getNumOnKeyDownCalls()
                cisPlaying = cbsmc.getNumIsPlayingCalls()
                cunregisterReceiver = cbsmc.getNumUnregisterRecieverCalls()
                conBackPressed = cbsmc.getNumOnBackPressedCalls()
                cshowDialog = cbsmc.getNumShowDialogCalls()
                ccreate = cbsmc.getNumCreateCalls()

                #Battery Life metrics
                batteryMetrics = BatteryMetrics.BatteryMetrics(sourceCodePaths,
layoutFilePaths)
                batteryMetrics.extractData()
                noTimeoutWakeLocks = batteryMetrics.getNumNoTimeoutWakeLocks()
                locListeners = batteryMetrics.getNumLocationListeners()
                gpsUses = batteryMetrics.getNumGpsUses()
                domParsers = batteryMetrics.getNumDomParsers()
                saxParsers = batteryMetrics.getNumSaxParsers()
                xmlPullParsers = batteryMetrics.getNumXMLPullParsers()

                #network timeout metrics
                networkTimeout = NetworkTimeout.NetworkTimeout(sourceCodePaths,
layoutFilePaths)
                networkTimeout.extractData()
                httpClients = networkTimeout.getNumHttpClients()
                numConTimeouts = networkTimeout.getNumConTimeouts()
                numSoTimeouts = networkTimeout.getNumSoTimeouts()
                numNoConTimeouts = networkTimeout.getNumNoConTimeout()
                numNoSoTimeouts = networkTimeout.getNumNoSoTimeout()

                #black hole exception handling
                blackHole = BlackHole.BlackHole(sourceCodePaths, layoutFilePaths)
                blackHole.extractData()
                numCatchBlocks = blackHole.getNumCatchBlocks()
                numLogOnly = blackHole.getNumLogOnly()
                numNoAction = blackHole.getNumNoAction()

                #ANR Metrics
                anrMetrics = ANRMetrics.ANRMetrics(sourceCodePaths, layoutFilePaths)
                anrMetrics.extractData()
                network = anrMetrics.getNumNetworkOnMainThread()
                sqlLite = anrMetrics.getNumSQLLiteOnMainThread()
                fileIO = anrMetrics.getNumFileIOOnMainThread()
                bitmap = anrMetrics.getNumBitmapOnMainThread()
                networkBg = anrMetrics.getNumNetworkOnBgThread()
                sqlLiteBg = anrMetrics.getNumSQLLiteOnBgThread()
                fileIOBg = anrMetrics.getNumFileIOOnBgThread()
                bitmapBg = anrMetrics.getNumBitmapOnBgThread()

                #intent launch metrics
                intentLaunchMetrics =
IntentLaunchMetrics.IntentLaunchMetrics(sourceCodePaths, layoutFilePaths, packageName)
                intentLaunchMetrics.extractData()
                startActivities = intentLaunchMetrics.getNumStartActivities()
                startActivity = intentLaunchMetrics.getNumStartActivity()
                startInstrumentation =
intentLaunchMetrics.getNumStartInstrumentation()
                startIntentSender = intentLaunchMetrics.getNumStartIntentSender()
                startService = intentLaunchMetrics.getNumStartService()
                startActionMode = intentLaunchMetrics.getNumStartActionMode()
                startActivityForResult =
intentLaunchMetrics.getNumStartActivityForResult()
                startActivityFromChild =
intentLaunchMetrics.getNumStartActivityFromChild()
                startActivityFromFragment =
intentLaunchMetrics.getNumStartActivityFromFragment()
```

```
                startActivityIfNeeded =
intentLaunchMetrics.getNumStartActivityIfNeeded()
                startIntentSenderForResult =
intentLaunchMetrics.getNumStartIntentSenderForResult()
                startIntentSenderFromChild =
intentLaunchMetrics.getNumStartIntentSenderFromChild()
                startNextMatchingActivity =
intentLaunchMetrics.getNumStartNextMatchingActivity()
                startSearch = intentLaunchMetrics.getNumStartSearch()

                db.writeAppTable(filename, appLabel, packageName, market)
                db.writeSizeMetricsTable(filename, numInstructions, numMethods,
numClasses, methodsPerClass, instrPerMethod, cyclomatic, wmc)
                db.writeOOMetricsTable(filename, noc, dit, lcom, cbo, ppiv, apd)
                db.writeMVCMetricsTable(filename, mvc, avgNumViewsInXML,
maxNumViewsInXML)
                db.writeOtherMetricsTable(filename, uncheckedBundles, potBadToken)
                db.updateNumFragments(filename,numFragments)
                db.writeAndroidObjectsTable(filename, objMap)
                db.writeBadSmellMethodCallsTable(filename, show, dismiss,
setContentView, createScaledBitmap, onKeyDown, isPlaying, unregisterReceiver,
onBackPressed, showDialog, create)
                db.writeUncheckedBadSmellMethodCallsTable(filename, cshow, cdismiss,
csetContentView, ccreateScaledBitmap, conKeyDown, cisPlaying, cunregisterReceiver,
conBackPressed, cshowDialog, ccreate)
                db.writeBatteryMetrics(filename, noTimeoutWakeLocks, locListeners,
gpsUses, domParsers, saxParsers, xmlPullParsers)
                db.writeNetworkTimeoutMetrics(filename, httpClients, numConTimeouts,
numSoTimeouts, numNoConTimeouts, numNoSoTimeouts)
                db.writeBlackHole(filename, numCatchBlocks, numLogOnly, numNoAction)
                db.writeANRMetrics(filename, network, sqlLite, fileIO, bitmap,
networkBg, sqlLiteBg, fileIOBg, bitmapBg)
                db.writeIntentLaunchMetrics(filename, startActivities, startActivity,
startInstrumentation, startIntentSender, startService, startActionMode,
startActivityForResult, startActivityFromChild, startActivityFromFragment,
startActivityIfNeeded, startIntentSenderForResult, startIntentSenderFromChild,
startNextMatchingActivity, startSearch)
            else:
                print "ERROR FOUND WITH FILE AndroidManifest.xml"
                #errorFile.write(f + ": ERROR FOUND WITH FILE AndroidManifest.xml\n")
        except:
            exc_type, exc_value, exc_traceback = sys.exc_info()
            lines = traceback.format_exception(exc_type, exc_value, exc_traceback)
            #errorFile.write(f + ": " + ''.join('!! ' + line for line in lines) +"\n")
    errorFile.close()

if __name__ == "__main__":
    extractData("C:\\apks\\decompiled\\","slideme")
    extractData("C:\\apks\\decompiled\\","fdroid")
    extractData("C:\\apks\\decompiled\\","appsapk")
```

Manifest Data Extraction

```
import os
import os.path
import xml.dom.minidom as minidom


'''
Collects data from the AndroidManifest.xml file
Created on Jan 23, 2014

@author: Billy Simon
@author: Eric Shaw
'''

class ManifestDataExtractor(object):
    '''
    classdocs
    '''


    def __init__(self, filePath):
        '''
        Constructor
        '''
        self.filePath = filePath
        if self.validateManifest():
            self.valid = True
            self.manifest = minidom.parse(filePath+"\\AndroidManifest.xml")
        else:
            self.valid = False

    def validateManifest(self):
        if os.path.exists(self.filePath+"\\AndroidManifest.xml"):
            return True
        else:
            return False

    def extractAppLabel(self):
        tag = self.manifest.getElementsByTagName("application")
        for item in tag:
            appName = item.getAttribute("android:label")
            if len(appName) > 0:
                if appName[0] == "@":
                    return self.getResource(appName)
                else:
                    return appName
            else:
                return "DATA NOT FOUND"

    def getResource(self, appName):
        if os.path.exists(self.filePath+"\\res\\values\\strings.xml"):
            try:
                strings = minidom.parse(self.filePath+"\\res\\values\\strings.xml")
            except:
                return "DATA NOT FOUND - ERROR PARSING XML FILE"
            elements = strings.getElementsByTagName("string")
            for tag in elements:
                if tag.getAttribute("name") == appName[8:]:
                    return tag.firstChild.nodeValue
            return "DATA NOT FOUND - No STRINGS.XML"
        else:
            return "DATA NOT FOUND - NO STRINGS.XML"
```

```
def extractFQName(self):
    tag = self.manifest.getElementsByTagName("manifest")
    for item in tag:
        fQName = item.getAttribute("package")
        return fQName

def manifestIsValid(self):
    return self.isValid
```

Size Metrics

```
'''
This class collects metrics related to code size.
Created on Jan 23, 2014

@author: Eric Shaw
'''
import fileinput

class SizeMetrics(object):
    '''
    classdocs
    '''


    def __init__(self, sourceCodePaths):
        '''
        Constructor
        '''
        self.paths = sourceCodePaths
        self.numFiles = len(self.paths)
        self.numInstr = 0
        self.numMethods = 0
        self.numClasses = 0
        self.methodsPerClass = 0
        self.cyclomatic = 0
        self. wmc = 0

    def getNumberofFiles(self):
        return self.numFiles

    def extractData(self):
        for path in self.paths:
            self.extractFileData(path)
        if self.numMethods == 0:
            self.instrPerMethod = 0
        else:
            self.instrPerMethod = self.numInstr / float(self.numMethods)
        if self.numClasses == 0:
            self.methodsPerClass = 0
            self.wmc = 0
            self.cyclomatic = 0
        else:
            self.methodsPerClass = self.numMethods / float(self.numClasses)
            self.wmc = (self.cyclomatic + self.numMethods) / float(self.numClasses)
            self.cyclomatic = (self.cyclomatic + self.numClasses) /
float(self.numClasses) #number conditions plus one for each method
        self.printData()

    def extractFileData(self, path):
        fileinput.close()
        for line in fileinput.input([path]):
            if line.startswith('.class '):
                self.numClasses = self.numClasses + 1
            if line.startswith('.method '):
                self.numMethods = self. numMethods + 1
            if len(line.strip()) > 0 and not line.strip()[0] == '.' and not
line.strip()[0] == '#':
                self.numInstr = self.numInstr + 1
            if 'if-' in line or 'If-' in line:
                self.cyclomatic = self.cyclomatic + 1
```

```
def getNumFiles(self):
    return self.numFiles

def getNumClasses(self):
    return self.numClasses

def getNumMethods(self):
    return self.numMethods

def getNumInstructions(self):
    return self.numInstr

def getMethodsPerClass(self):
    return self.methodsPerClass

def getInstructionsPerMethod(self):
    return self.instrPerMethod

def getCyclomatic(self):
    return self.cyclomatic

def getWMC(self):
    return self.wmc


#for debugging - will write to database (this will drive schema)
def printData(self):
    print 'Number of files: ' + str(self.numFiles)
    print 'Number of Classes: ' + str(self.numClasses)
    print 'Number of Methods ' + str(self.numMethods)
    print 'Number of Bytecode Instructions: ' + str(self.numInstr)
    print 'Methods per Class: ' + str(self.methodsPerClass)
    print 'Bytecode Instructions per method: ' + str(self.instrPerMethod)
    print 'Cyclomatic complexity: ' + str(self.cyclomatic)
    print 'WMC: ' + str(self.wmc)
```

Object Oriented (CK) Metrics

```
'''
This class collects several object oriented metrics originally suggested by Chidamber
and Kemerer.
Created on Jan 23, 2014

@author: Eric Shaw
'''
import fileinput
import Class
import re
import Method

ignoreFiles = ["R.smali","R$attr.smali","R$dimen.smali","R$drawable.smali",

"R$id.smali","R$layout.smali","R$menu.smali","R$string.smali","R$style.smali"]

class CKMetrics(object):
    '''
    classdocs
    '''


    def __init__(self, sourceCodePaths, package):
        '''
        Constructor
        '''
        self.paths = sourceCodePaths
        self.numFiles = len(self.paths)
        self.lcom = 0.0
        self.cbo = 0.0
        self.maxDit = 0
        self.noc = 0
        self.classes = []
        self.ppiv = 0.0
        self.apd = 0
        self.package = package
        self.pkgpath = 'L' + self.package.replace('.','/') + '/'

    def getNumberofFiles(self):
        return self.numFiles

    def extractData(self):
        for path in self.paths:
            self.extractFileData(path)
        self.determineChildrenFromParents()
        self.noc = self.calculateNOC()
        self.maxDit = self.calculateMaxDIT()
        self.lcom = self.calculateLCOM()
        self.cbo = self.calculateCBO()
        self.apd = self.apd / float(len(self.classes))
        self.ppiv = self.calculatePercentPublicInstanceVariables()
        self.printData()

    def extractFileData(self, path):
        fileinput.close()
        curClass = Class.Class()
        for line in fileinput.input([path]):
            if line.startswith('.class '):
                if self.pkgpath not in line:
                    continue
                curClass.setPackage(self.extractPackage(line))
                curClass.setName(self.extractClassName(line))
            elif line.startswith('.super '):
                if(self.pkgpath in line):
                    curClass.setParent(self.extractClassName(line))
                matches = re.findall("Landroid/app/(.*?)Activity", line)
                curClass.isController = (len(matches) > 0)
            elif(line.startswith('.field')):
                fieldName = line[line.rfind(" ")+1:line.rfind(":")]
                curClass.addField(fieldName)
                if(' public ' in line or ' protected ' in line):
                    curClass.addPublicField(fieldName)
            elif(line.startswith('.method ')):
                if('<init>' in line):
                    curMethod = Method.Method('constructor')
                else:
                    curMethod = Method.Method(re.findall(" (.*?)\(", line[line.rfind('
'):])[-1])
                curClass.addMethod(curMethod)
            elif(";->" in line):
                if(self.usesField(curClass, line)): #for LCOM
                    fieldName = line[line.rfind(";->")+3:line.rfind(":")]
                    curMethod.addFieldUsed(fieldName)
                    if fieldName in curClass.getPublicFields():
                            self.apd = self.apd + 1
                else: #for CBO
                    if not curClass.isController and self.refsNonJavaClass(line):
                        objName = line[line.index("L"):line.index(";->")]
                        if not objName == curClass.getName() and not
self.inSameTree(curClass, objName):
                            curClass.addCoupledObject(objName)

        self.classes.append(curClass)


    def refsNonJavaClass(self, line):
        pkgRegex = "L(.*?);"
        javaPkgRegex = "Ljava/(.*?)"
        pkgMatches = re.findall(pkgRegex, line)
        javaPkgRegex = re.findall(javaPkgRegex, line)
        return len(pkgMatches) - len(javaPkgRegex) > 0

    def extractPackage(self, line):
        pkg = line.split()[-1]
        pkg = pkg[:pkg.rfind("/")]
        return pkg

    #Gets the class name
    def extractClassName(self, line):
        return line.split()[-1].replace(";","")

    def determineChildrenFromParents(self):
        for aClass in self.classes:
            if aClass.getParent() != '':
                parent = self.getClassByName(aClass.getParent())
                parent.addChild(aClass)


    def getClassByName(self, name):
        for aClass in self.classes:
            if aClass.getName() ==  name:
                return aClass

    def calculateLCOM(self):
        classLCOM = []
        for aClass in self.classes:
            if(len(aClass.getFields()) == 0) or (len(aClass.getMethods()) == 0):
                continue
            #numFieldCalls = 0
            methodLCOM = []
            fieldcount = {} #number of methods using each field in a class
            for aField in aClass.getFields():
                fieldcount[aField] = 0
            for aMethod in aClass.getMethods():
                for aField in aMethod.getFieldsUsed():
                    fieldcount[aField] = fieldcount[aField] + 1
            for aField in aClass.getFields():
                methodLCOM.append(float(fieldcount[aField]) /
len(aClass.getMethods()))
            classLCOM.append(1 - self.avg(methodLCOM))
        return self.avg(classLCOM) * 100

    #determine it objName is the name of any class in the inheritance tree of curClass
    def inSameTree(self, curClass, objName):
        try:
            while not curClass.getParent() == '':
                if curClass.getName() == objName:
                    return True
                curClass = self.getClassByName(curClass.getParent())

            return self.depthFirstTreeSearch(curClass, objName)
        except AttributeError:
            return False

    def depthFirstTreeSearch(self, curClass, objName):
        if(curClass.getName() == objName):
            return True
        for aChild in curClass.getChildren():
            return self.depthFirstTreeSearch(curClass, aChild.getName())
        return False

    def getClassByName(self,name):
        for aClass in self.classes:
            if aClass.getName() == name:
                return aClass

    def avg(self, list):
        if(len(list)) == 0:
            return 0
        total = 0.0
        for num in list:
            total = total + num
        return total / len(list)

    #calculate avg number of coupled objects per class
    def calculateCBO(self):
        totalCBO = 0
        length = 0
        for aClass in self.classes:
            totalCBO = totalCBO + len(aClass.getCoupledObjects())
            length = length + 1
        return totalCBO / float(length)

    def calculateNOC(self):
        noc = 0
        for aClass in self.classes:
            if(len(aClass.getChildren()) > self.noc):
                noc = len(aClass.getChildren())
        return noc

    def calculateMaxDIT(self):
        maxHeight = 0
        for aClass in self.classes:
            if aClass.getParent() == '':
                height = self.getTreeHeight(aClass)
                if height > maxHeight:
                    maxHeight = height
        return maxHeight

    def getTreeHeight(self, aClass):
        if len(aClass.getChildren()) == 0:
            return 1
        max = 0
        for child in aClass.getChildren():
            curHeight = 1 + self.getTreeHeight(child)
            if curHeight > max:
                max = curHeight
        return max

    def calculatePercentPublicInstanceVariables(self):
        public = 0
        total = 0
        for aClass in self.classes:
            public = public + len(aClass.getPublicFields())
            total = total + len(aClass.getFields())
        if total == 0:
            return 0
        return (public / float(total)) * 100

    def usesField(self, curClass, line):
        field = line[line.rfind(";->")+3:line.rfind(":")]
        return field in curClass.getFields()

    def getNOC(self):
        return self.noc

    def getDIT(self):
        return self.maxDit

    def getLCOM(self):
        return self.lcom

    def getCBO(self):
        return self.cbo

    def getPPIV(self):
        return self.ppiv

    def getAPD(self):
        return self.apd

    #for debugging - will write to database (this will drive schema)
    def printData(self):
        print 'Max NOC: ' + str(self.noc)
        print 'Max DIT: ' + str(self.maxDit)
        print 'LCOM: ' + str(self.lcom) + '%'
        print 'CBO: ' + str(self.cbo)
        print 'PPIV: ' + str(self.ppiv) + '%'
        print 'APD: ' + str(self.apd) + ' accesses per class'
```

Class

```
'''
Represents a java class
Created on Jan 23, 2014

@author: ess0006
'''

class Class(object):

    def __init__(self):
        '''
        Constructor
        '''
        self.name = ''
        self.package = ''
        self.parent = ''
        self.children = []
        self.fields = []
        self.methods = []
        self.coupledObjects = []
        self.publicFields = []
        self.isController = False

    def setName(self, name):
        self.name = name

    def setPackage(self, package):
        self.package = package

    def setParent(self, parent):
        self.parent = parent

    def setChildren(self, children):
        self.children = children

    def getName(self):
        return self.name

    def getPackage(self):
        return self.package

    def getParent(self):
        return self.parent

    def getChildren(self):
        return self.children

    def addChild(self, child):
        self.children.append(child)

    def addField(self, field):
        self.fields.append(field)

    def getFields(self):
        return self.fields

    def getMethods(self):
        return self.methods
```

```
def addMethod(self, method):
    self.methods.append(method)

def getCoupledObjects(self):
    return self.coupledObjects

def addCoupledObject(self, obj):
    if obj not in self.coupledObjects:
        self.coupledObjects.append(obj)

def getPublicFields(self):
    return self.publicFields

def addPublicField(self, field):
    if not field in self.publicFields:
        self.publicFields.append(field)

def toString(self):
    childNames = []
    for aChild in self.children:
        childNames.append(aChild.getName())
      return 'Name: ' + self.name + '\nParent: ' + self.parent + '\nChildren: ' +
                                str(childNames)
```

Method

```
'''
This class represents a java method.
Created on Jan 24, 2014

@author: Eric Shaw
'''

class Method(object):
    '''
    classdocs
    '''


    def __init__(self, name):
        '''
        Constructor
        '''
        self.name = name
        self.fieldsUsed = []

    def addFieldUsed(self, field):
        if field not in self.fieldsUsed:
            self.fieldsUsed.append(field)

    def getFieldsUsed(self):
        return self.fieldsUsed

    def setName(self, name):
        self.name = name

    def getName(self):
  return self.name
```

View Metrics

```
'''
This class collects view related metrics on the apps.
Created on Jan 27, 2014

@author: ess0006
'''
import re
import fileinput
import AndroidViews

class MVCMetrics(object):
    '''
    classdocs
    '''


    def __init__(self, sourceCodePaths, layoutPaths):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.numViewsInController = 0
        self.numViewsNotInController = 0
        self.numViewsInXML = 0
        self.maxNumViewsInXML = 0
        self.sepVCScore = 0.0
        self.potBadToken = 0
        self.numFragments = 0

    def getNumberofFiles(self):
        return self.numSrcFiles

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        for path in self.layoutPaths:
            self.extractLayoutFileData(path)
        self.sepVCScore = self.calculateSepVCScore()
        self.printData()

    def extractSrcFileData(self, path):
        fileinput.close()
        className = ""
        viewInitRegex = "new-instance(.*?)Landroid/widget/" +
AndroidViews.getAndroidViewsRegex()
        isController = False
        for line in fileinput.input([path]):
            if line.startswith(".source"):
                className = line.split()[-1].replace("\"","").replace(".java","")
            if line.startswith('.super '):
                matches = re.findall("Landroid/app/(.*?)Activity", line)
                isController = (len(matches) > 0)
            else:
                matches = re.findall(viewInitRegex, line)
                if(len(matches) > 0):
                    if(isController):
                        self.numViewsInController = self.numViewsInController + 1
                    else:
                        self.numViewsNotInController = self.numViewsNotInController +
1
                if(isController):
                    if"getApplicationContext()Landroid/content/Context" in line and
not className in line:
                        self.potBadToken = self.potBadToken + 1


    def extractLayoutFileData(self, path):
        fileinput.close()
        tempMax = 0
        viewRegex = "<" + AndroidViews.getAndroidViewsRegex()
        for line in fileinput.input([path]):
                if "<fragment" in line:
                    self.numFragments = self.numFragments + 1
                numMatches = len(re.findall(viewRegex, line))
                self.numViewsNotInController = self.numViewsNotInController +
numMatches
                self.numViewsInXML = self.numViewsInXML + numMatches
                tempMax = tempMax + numMatches

        if tempMax > self.maxNumViewsInXML:
            self.maxNumViewsInXML = tempMax

    def calculateSepVCScore(self):
        if(self.numViewsNotInController + self.numViewsInController == 0):
            return 0
        return self.numViewsNotInController / float(self.numViewsNotInController +
self.numViewsInController) * 100

    def getNumViewsInController(self):
        return self.numViewsInController

    def getNumViewsNotInController(self):
        return self.numViewsNotInController

    def getAvgNumViewsInXML(self):
        if len(self.layoutPaths) == 0:
            return 0
        return float(self.numViewsInXML) / float(len(self.layoutPaths))

    def getMaxNumViewsInXML(self):
        return self.maxNumViewsInXML

    def getSepVCScore(self):
        return self.sepVCScore

    def getPotentialBadTokenExceptions(self):
        return self.potBadToken

    def getNumFragments(self):
        return self.numFragments

    def printData(self):
        print "Num views in controllers: " + str(self.numViewsInController)
        print "Num views not in controllers: " + str(self.numViewsNotInController)
        print "# views in XML: " + str(self.numViewsInXML)
        print "Max # views in an XML file: " + str(self.maxNumViewsInXML)
        print "Percentage of Views Defined Outside of Controllers: " +
str(self.sepVCScore) + "%"
```

Android Views

```
ANDROID_VIEWS = ["AbsListView",
                 "AbsListView.LayoutParams",
                 "AbsoluteLayout",
                 "AbsoluteLayout.LayoutParams",
                 "AbsSeekBar",
                 "AbsSpinner",
                 "AnalogClock",
                 "Button",
                 "CalendarView",
                 "CheckBox",
                 "CheckedTextView",
                 "Chronometer",
                 "CompoundButton",
                 "DatePicker",
                 "DigitalClock",
                 "EditText",
                 "ExpandableListView",
                 "ExpandableListView.ExpandableListContextMenuInfo",
                 "FrameLayout",
                 "FrameLayout.LayoutParams",
                 "Gallery",
                 "Gallery.LayoutParams",
                 "GridLayout",
                 "GridLayout.Alignment",
                 "GridLayout.LayoutParams",
                 "GridLayout.Spec",
                 "GridView",
                 "HeaderViewListAdapter",
                 "HorizontalScrollView",
                 "ImageButton",
                 "ImageSwitcher",
                 "ImageView",
                 "LinearLayout",
                 "LinearLayout.LayoutParams",
                 "ListPopupWindow",
                 "ListView",
                 "ListView.FixedViewInfo",
                 "MediaController",
                 "MultiAutoCompleteTextView",
                 "MultiAutoCompleteTextView.CommaTokenizer",
                 "NumberPicker",
                 "OverScroller",
                 "PopupMenu",
                 "PopupWindow",
                 "ProgressBar",
                 "QuickContactBadge",
                 "RadioButton",
                 "RadioGroup",
                 "RadioGroup.LayoutParams",
                 "RatingBar",
                 "RelativeLayout",
                 "RelativeLayout.LayoutParams",
                 "Scroller",
                 "ScrollView",
                 "SearchView",
                 "SeekBar",
                 "SlidingDrawer",
                 "Space",
                 "Spinner",
                 "StackView",
                 "Switch",
                 "TabHost",
                 "TabHost.TabSpec",
                 "TableLayout",
                 "TableLayout.LayoutParams",
                 "TableRow",
                 "TableRow.LayoutParams",
                 "TabWidget",
                 "TextClock",
                 "TextView",
                 "TimePicker",
                 "ToggleButton",
                 "VideoView",
                 "ZoomButton",
                 "ZoomButtonsController",
                 "ZoomControls"]

 """ Gets a regular expression of all view objects in Android """
 def getAndroidViewsRegex():
     regex = "("
     first = True
     for view in ANDROID_VIEWS:
         if not first:
             regex = regex + "|"
         else:
             first = False
         regex = regex + view
     regex = regex + ")"
     return regex
```

Unchecked Bundles

```
'''
This class counts the number of unchecked Intent Bundles per app.
Created on Jan 27, 2014

@author: ess0006
'''
import re
import fileinput
import AndroidViews

class UncheckedBundles(object):
    '''
    classdocs
    '''


    def __init__(self, sourceCodePaths, layoutPaths):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.numBundles = 0
        self.numCheckedBundles = 0
        self.inTryCatch = False

    def getNumberofFiles(self):
        return self.numFiles

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        #for path in self.layoutPaths:
            #self.extractLayoutFileData(path)
        self.printData()

    def extractSrcFileData(self, path):
        fileinput.close()
        bundleAssignment = ".local (.*?):Landroid/os/Bundle;"
        bundleRegisters = []
        self.inTryCatch = False
        for line in fileinput.input([path]):
            if line.startswith(":try_start"):
                self.inTryCatch = True
            if line.startswith(":try_end"):
                self.inTryCatch = False
            matches = re.findall(bundleAssignment, line)
            if len(matches) > 0 and not ".end" in line:
                reg = matches[0][0:matches[0].find(",")]
                if not reg in bundleRegisters and not self.inTryCatch:
                    bundleRegisters.append(reg)
                self.numBundles = self.numBundles + 1
            else:
                if len(bundleRegisters) != 0:
                    nullCheck = "(if-nez|if-eqz) "
                    regName = ""
                    for reg in bundleRegisters:
                        matches = re.findall(nullCheck + reg, line)
                        if len(matches) > 0:
```

```
                        self.numCheckedBundles = self.numCheckedBundles + 1
                        regName = reg
                if regName != "":
                    bundleRegisters.remove(regName)


def extractLayoutFileData(self, path):
    fileinput.close()
    tempMax = 0
    viewRegex = "<" + AndroidViews.getAndroidViewsRegex()
    for line in fileinput.input([path]):
            numMatches = len(re.findall(viewRegex, line))

def getNumCheckedBundles(self):
    return self.numCheckedBundles

def getNumUncheckedBundles(self):
    return self.numBundles - self.numCheckedBundles

def getObjectMap(self):
    return self.objMap

def printData(self):
    print ""
```

Android Objects

```
'''
This class collects data on unchecked bundles and Android objects used
*****Bundle logic has been changed, use UncheckedBundles.py
Created on Jan 27, 2014

@author: ess0006
'''
import re
import fileinput
import AndroidViews

class OtherMetrics(object):
    '''
    classdocs
    '''


    def __init__(self, sourceCodePaths, layoutPaths):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.numBundles = 0
        self.numCheckedBundles = 0
        self.objMap = {}

    def getNumberofFiles(self):
        return self.numFiles

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        #for path in self.layoutPaths:
            #self.extractLayoutFileData(path)
        self.printData()

    def extractSrcFileData(self, path):
        fileinput.close()
        bundleAssignment = ".local (.*?):Landroid/os/Bundle;"
        androidObj = "Landroid/(.*?)/(.*?);";
        bundleRegisters = []
        for line in fileinput.input([path]):
            matches = re.findall(bundleAssignment, line)
            if len(matches) > 0 and not ".end" in line:
                reg = matches[0][0:matches[0].find(",")]
                if not reg in bundleRegisters:
                    bundleRegisters.append(reg)
                self.numBundles = self.numBundles + 1
            else:
                if len(bundleRegisters) != 0:
                    nullCheck = "(if-nez|if-eqz) "
                    regName = ""
                    for reg in bundleRegisters:
                        matches = re.findall(nullCheck + reg, line)
                        if len(matches) > 0:
                            self.numCheckedBundles = self.numCheckedBundles + 1
                            regName = reg
                    if regName != "":
                        bundleRegisters.remove(regName)
            objMatches = re.findall(androidObj, line)
            if len(objMatches) > 0:
                for tuple in objMatches:
                    list = []
                    list.append('android')
                    list.append(tuple[0])
                    list.append(tuple[1])
                    fulQual = '.'.join(list)
                    fulQual = fulQual.replace('$', '.')
                    if fulQual in self.objMap:
                        self.objMap[fulQual] = self.objMap[fulQual] + 1
                    else:
                        self.objMap[fulQual] = 1

    def extractLayoutFileData(self, path):
        fileinput.close()
        tempMax = 0
        viewRegex = "<" + AndroidViews.getAndroidViewsRegex()
        for line in fileinput.input([path]):
                numMatches = len(re.findall(viewRegex, line))


    def getNumUncheckedBundles(self):
        return self.numBundles - self.numCheckedBundles

    def getObjectMap(self):
        return self.objMap

    def printData(self):
        print ""
```

Bad Smell Method Calls

```
'''
This class looks for method calls that are known to cause exceptions, based on the
list presented in http://istlab.dmst.aueb.gr/~mkehagia/api-exceptions.pdf
Created on Jun 27, 2014

@author: Eric Shaw
'''
import re
import fileinput
import AndroidViews

class UncheckedBadSmellMethodCalls(object):

    def __init__(self, sourceCodePaths, layoutPaths):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.dismiss = 0
        self.show = 0
        self.setContentView = 0
        self.createScaledBitmap = 0
        self.onKeyDown = 0
        self.isPlaying = 0
        self.unregisterReceiver = 0
        self.onBackPressed = 0
        self.showDialog = 0
        self.create = 0
        self.inTryCatch = False

    def getNumberofFiles(self):
        return self.numFiles

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        #for path in self.layoutPaths:
            #self.extractLayoutFileData(path)
        self.printData()

    def extractSrcFileData(self, path):
        fileinput.close()
        for line in fileinput.input([path]):
            if not self.inTryCatch:
                if line.startswith(":try_start"):
                    self.inTryCatch = True
                else:
                    matches = re.findall("invoke-virtual (.*?), Landroid/(.*?);-
>dismiss\(", line)
                    if len(matches) > 0:
                        self. dismiss = self.dismiss + 1
                    matches = re.findall("invoke-virtual (.*?), Landroid/(.*?);-
>show\(", line)
                    if len(matches) > 0:
                        self.show = self.show + 1
                    matches = re.findall("invoke-virtual (.*?), (.*?);-
>setContentView\(", line)
                    if len(matches) > 0:
                        self.setContentView = self.setContentView + 1
                    matches = re.findall("invoke-virtual (.*?), Landroid/(.*?);-
>createScaledBitmap\(", line)
                    if len(matches) > 0:
                        self. createScaledBitmap = self.createScaledBitmap + 1
                    matches = re.findall("invoke-virtual (.*?), (.*?);->onKeyDown\(",
line)
                    if len(matches) > 0:
                        self.onKeyDown = self.onKeyDown + 1
                    matches = re.findall("invoke-virtual (.*?), Landroid/(.*?);-
>isPlaying\(", line)
                    if len(matches) > 0:
                        self.isPlaying = self.isPlaying + 1
                    matches = re.findall("invoke-virtual (.*?), (.*?);-
>unregisterReceiver\(", line)
                    if len(matches) > 0:
                        self.unregisterReceiver = self.unregisterReceiver + 1
                    matches = re.findall("invoke-virtual (.*?), (.*?);-
>onBackPressed\(", line)
                    if len(matches) > 0:
                        self. onBackPressed = self.onBackPressed + 1
                    matches = re.findall("invoke-virtual (.*?), (.*?);->showDialog\(",
line)
                    if len(matches) > 0:
                        self.showDialog = self.showDialog + 1
                    matches = re.findall("invoke-virtual (.*?), Landroid/(.*?);-
>create\(", line)
                    if len(matches) > 0:
                        self.create = self.create + 1
            else:
                if line.startswith(":try_end"):
                    self.inTryCatch = False



    def extractLayoutFileData(self, path):
        fileinput.close()




    def getNumShowCalls(self):
        return self.show

    def getNumDismissCalls(self):
        return self.dismiss

    def getNumSetContentViewCalls(self):
        return self.setContentView

    def getNumCreateScaledBitmapCalls(self):
        return self.createScaledBitmap

    def getNumOnKeyDownCalls(self):
        return self.onKeyDown

    def getNumIsPlayingCalls(self):
        return self.isPlaying

    def getNumUnregisterRecieverCalls(self):
        return self.unregisterReceiver

    def getNumOnBackPressedCalls(self):
        return self.onBackPressed
```

```
def getNumShowDialogCalls(self):
    return self.showDialog

def getNumCreateCalls(self):
    return self.create

def printData(self):
    print ""
```

Component Launch Metrics

```
'''
This class searches for the use of a set of methods.
Created on July 17, 2014

@author: ess0006
'''
import re
import fileinput

class IntentLaunchMetrics(object):

    def __init__(self, sourceCodePaths, layoutPaths,  package):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.startActivities = 0
        self.startActivity = 0
        self.startInstrumentation = 0
        self.startIntentSender = 0
        self.startService = 0
        self.startActionMode = 0
        self.startActivityForResult = 0
        self.startActivityFromChild = 0
        self.startActivityFromFragment = 0
        self.startActivityIfNeeded = 0
        self.startIntentSenderForResult = 0
        self.startIntentSenderFromChild = 0
        self.startNextMatchingActivity = 0
        self.startSearch = 0
        self.total = 0
        self.package = package
        self.pkgpath = 'L' + self.package.replace('.','/') + '/'

    def getNumberofFiles(self):
        return self.numFiles

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        #for path in self.layoutPaths:
            #self.extractLayoutFileData(path)
        self.printData()

    def extractSrcFileData(self, path):
        fileinput.close()
        startActionModeRegex = self.pkgpath + "(.*?);->startActionMode\("
        startActivitiesRegex = self.pkgpath + "(.*?);->startActivities\("
        startActivityRegex = self.pkgpath + "(.*?);->startActivity\("
        startActivityForResultRegex = self.pkgpath + "(.*?);-
>startActivityForResult\("
        startActivityFromChildRegex = self.pkgpath + "(.*?);-
>startActivityFromChild\("
        startActivityFromFragmentRegex = self.pkgpath + "(.*?);-
>startActivityFromFragment\("
        startActivityIfNeededRegex = self.pkgpath + "(.*?);->startActivityIfNeeded\("
        startInstrumentationRegex = self.pkgpath + "(.*?);->startInstrumentation\("
        startIntentSenderRegex = self.pkgpath + "(.*?);->startIntentSender\("
        startIntentSenderForResultRegex = self.pkgpath + "(.*?);-
>startIntentSenderForResult\("
        startIntentSenderFromChildRegex = self.pkgpath + "(.*?);-
>startIntentSenderFromChild\("
        startNextMatchingActivityRegex = self.pkgpath + "(.*?);-
>startNextMatchingActivity\("
        startSearchRegex = self.pkgpath +"(.*?);->startSearch\("
        startServiceRegex = self.pkgpath + "(.*?);->startService\("

        for line in fileinput.input([path]):

            matches = re.findall(startActivitiesRegex, line)
            if len(matches) > 0:
                self.startActivities = self.startActivities + 1

            matches = re.findall(startActivityRegex, line)
            if len(matches) > 0:
                self.startActivity = self.startActivity + 1

            matches = re.findall(startInstrumentationRegex, line)
            if len(matches) > 0:
                self.startInstrumentation = self.startInstrumentation + 1

            matches = re.findall(startIntentSenderRegex, line)
            if len(matches) > 0:
                self.startIntentSender = self.startIntentSender + 1

            matches = re.findall(startServiceRegex, line)
            if len(matches) > 0:
                self.startService = self.startService + 1

            matches = re.findall(startActionModeRegex, line)
            if len(matches) > 0:
                self.startActionMode = self.startActionMode + 1

            matches = re.findall(startActivityForResultRegex, line)
            if len(matches) > 0:
                self.startActivityForResult = self.startActivityForResult + 1

            matches = re.findall(startActivityFromChildRegex, line)
            if len(matches) > 0:
                self.startActivityFromChild = self.startActivityFromChild + 1

            matches = re.findall(startActivityFromFragmentRegex, line)
            if len(matches) > 0:
                self.startActivityFromFragment = self.startActivityFromFragment + 1

            matches = re.findall(startActivityIfNeededRegex, line)
            if len(matches) > 0:
                self.startActivityIfNeeded = self.startActivityIfNeeded + 1

            matches = re.findall(startIntentSenderForResultRegex, line)
            if len(matches) > 0:
                self.startIntentSenderForResult = self.startIntentSenderForResult + 1

            matches = re.findall(startIntentSenderFromChildRegex, line)
            if len(matches) > 0:
                self.startIntentSenderFromChild = self.startIntentSenderFromChild + 1

            matches = re.findall(startNextMatchingActivityRegex, line)
            if len(matches) > 0:
                self.startNextMatchingActivity = self.startNextMatchingActivity + 1
            matches = re.findall(startSearchRegex, line)
            if len(matches) > 0:
                self.startSearch = self.startSearch + 1

    def extractLayoutFileData(self, path):
        pass

    def getNumStartActivities(self):
        return self.startActivities

    def getNumStartActivity(self):
        return self.startActivity

    def getNumStartInstrumentation(self):
        return self.startInstrumentation

    def getNumStartIntentSender(self):
        return self.startIntentSender

    def getNumStartService(self):
        return self.startService
    #new
    def getNumStartActionMode(self):
        return self.startActionMode

    def getNumStartActivityForResult(self):
        return self.startActivityForResult

    def getNumStartActivityFromChild(self):
        return self.startActivityFromChild

    def getNumStartActivityFromFragment(self):
        return self.startActivityFromFragment

    def getNumStartActivityIfNeeded(self):
        return self.startActivityIfNeeded

    def getNumStartIntentSenderForResult(self):
        return self.startIntentSenderForResult

    def getNumStartIntentSenderFromChild(self):
        return self.startIntentSenderFromChild

    def getNumStartNextMatchingActivity(self):
        return self.startNextMatchingActivity

    def getNumStartSearch(self):
        return self.startSearch

    def printData(self):
        print ""
```

Battery Metrics

```
'''
Metrics gleaned from Google I/O presentation
http://www.youtube.com/watch?v=OUemfrKe65c&feature=player_embedded
Created on Jun 27, 2014

@author: Eric Shaw
'''
import fileinput
import re

class BatteryMetrics(object):


    def __init__(self, sourceCodePaths, layoutPaths):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.numNoTimeoutWakeLocks = 0
        self.numLocListeners = 0
        self.numGpsUses = 0
        self.numXMLPullParser = 0
        self.numSaxParser = 0
        self.numDomParser = 0

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        for path in self.layoutPaths:
            self.extractLayoutFileData(path)

    def extractSrcFileData(self, path):
        fileinput.close()
        isLocListener = False
        wakeLockAcqRegex = "invoke-virtual(.*?)Landroid/os/PowerManager$WakeLock;-
>acquire()"
        domRegex = "invoke-virtual(.*?)Ljavax/xml/parsers/DocumentBuilderFactory;-
>newDocumentBuilder()"
        saxRegex = "invoke-virtual(.*?)Ljavax/xml/parsers/SAXParserFactory;-
>newSAXParser()"
        xmlppRegex = "invoke-static(.*?)Landroid/util/Xml;->newPullParser()"
        for line in fileinput.input([path]):
            matches = re.findall(wakeLockAcqRegex, line)
            if len(matches) > 0:
                self.numNoTimeoutWakeLocks = self.numNoTimeoutWakeLocks + 1
            if line.startswith(".implements Landroid/location/LocationListener;"):
                self.numLocListeners = self.numLocListeners + 1
                isLocListener = True
            if isLocListener:
                if "\"gps\"" in line:
                    self.numGpsUses = self. numGpsUses + 1
            matches = re.findall(domRegex, line)
            if len(matches) > 0:
                self.numDomParser = self.numDomParser + 1
            matches = re.findall(saxRegex, line)
            if len(matches) > 0:
                self.numSaxParser = self.numSaxParser + 1
            matches = re.findall(xmlppRegex, line)
            if len(matches) > 0:
                self.numXMLPullParser = self.numXMLPullParser + 1

    def extractLayoutFileData(self, path):
        pass

    def getNumNoTimeoutWakeLocks(self):
        return self.numNoTimeoutWakeLocks

    def getNumLocationListeners(self):
        return self.numLocListeners

    def getNumGpsUses(self):
        return self.numGpsUses

    def getNumDomParsers(self):
        return self.numDomParser

    def getNumSaxParsers(self):
        return self.numSaxParser

    def getNumXMLPullParsers(self):
        return self.numXMLPullParser
```

Black Hole Exception Handling

```
'''
This class searches for exception handling without any corrective actions
Created on July 11, 2014

@author: Eric Shaw
'''
import fileinput
import re

class BlackHole(object):

    def __init__(self, sourceCodePaths, layoutPaths):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.numCatchBlocks = 0
        self.numLogOnly = 0
        self.numNoAction = 0

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        for path in self.layoutPaths:
            self.extractLayoutFileData(path)

    def extractSrcFileData(self, path):
        fileinput.close()
        catchNum = 0
        correct = False
        log = False
        logRegex1 = "invoke-virtual(.*?)Ljava/lang/Exception;->printStackTrace()"
        logRegex2 = "invoke-static(.*?)Landroid/util/Log;->e"
        for line in fileinput.input([path]):
            if line.strip().startswith(":catch_"):
                self.numCatchBlocks = self.numCatchBlocks + 1
                if not log and not correct:
                    self.numNoAction = self.numNoAction + 1
                elif log and not correct:
                    self.numLogOnly = self.numLogOnly + 1
                correct = False
                log = False
                catchNum = catchNum + 1
            if catchNum > 0 and line.startswith(".end method"):#catch clauses come at
the end of the method
                if not log and not correct:
                    self.numNoAction = self.numNoAction + 1
                elif log and not correct:
                    self.numLogOnly = self.numLogOnly + 1
                correct = False
                log = False
                catchNum = 0
                continue
            if catchNum > 0:
                if not line.strip().startswith(".") and not
line.strip().startswith("goto") and not line.strip().startswith("move-
exception"):#looking for developer method calls
                    matches1 = re.findall(logRegex1, line)
                    matches2 = re.findall(logRegex2, line)
                    if len(matches1) > 0 or len(matches2) > 0:
                        log = True
                    else:
                        correct = True


    def extractLayoutFileData(self, path):
        pass

    def getNumCatchBlocks(self):
        return self.numCatchBlocks

    def getNumNoAction(self):
        return self.numNoAction

    def getNumLogOnly(self):
        return self.numLogOnly
```

ANR Metrics

```
'''
This measures four metrics related to ANR, the use of each of the following on the
main thread:
networking
file IO
SQL Lite
bitmaps

Created on July 14, 2014

@author: Eric Shaw
'''
import re
import fileinput

class ANRMetrics(object):
    '''
    classdocs
    '''

    def __init__(self, sourceCodePaths, layoutPaths):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.numNetworkOnMainThread = 0
        self.numSQLLiteOnMainThread = 0
        self.numFileIOOnMainThread = 0
        self.numBitmapOnMainThread = 0
        self.numNetworkOnBgThread = 0
        self.numSQLLiteOnBgThread = 0
        self.numFileIOOnBgThread = 0
        self.numBitmapOnBgThread = 0

    def getNumberofFiles(self):
        return self.numFiles

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        #for path in self.layoutPaths:
            #self.extractLayoutFileData(path)
        self.printData()

    def extractSrcFileData(self, path):
        fileinput.close()
        isActivity = False
        activityRegex = ".super Landroid/app/(.*?)Activity;" #since it is difficult to
see how data is passed, consider only code in activities to be run on UI thread
        httpRequestRegex1 = "Landroid/net/http/AndroidHttpClient;->execute\("
        httpRequestRegex2 = "Lorg/apache/http/impl/client/DefaultHttpClient;-
>execute\("
        fileIORegex = "new-instance(.*?)Ljava/io/File"
        sqlLiteMethodRegex = "Landroid/database/sqlite/SQLiteDatabase;->(.*?)\("
        bitmapRegex = "Landroid/graphics/BitmapFactory;->decode(.*?)\("

        for line in fileinput.input([path]):
            matches = re.findall(activityRegex, line)
            if len(matches) > 0:
                isActivity = True

            if isActivity:
                matches1 = re.findall(httpRequestRegex1, line)
                matches2 = re.findall(httpRequestRegex2, line)
                if len(matches1) > 0 or len(matches2) > 0:
                    self.numNetworkOnMainThread = self.numNetworkOnMainThread + 1

                matches = re.findall(sqlLiteMethodRegex, line)
                if len(matches) > 0:
                    self.numSQLLiteOnMainThread = self.numSQLLiteOnMainThread + 1

                matches = re.findall(fileIORegex, line)
                if len(matches) > 0:
                    self.numFileIOOnMainThread = self.numFileIOOnMainThread + 1

                matches = re.findall(bitmapRegex, line)
                if len(matches) > 0:
                    self.numBitmapOnMainThread = self.numBitmapOnMainThread + 1

            else:
                matches1 = re.findall(httpRequestRegex1, line)
                matches2 = re.findall(httpRequestRegex2, line)
                if len(matches1) > 0 or len(matches2) > 0:
                    self.numNetworkOnBgThread = self.numNetworkOnBgThread + 1

                matches = re.findall(sqlLiteMethodRegex, line)
                if len(matches) > 0:
                    self.numSQLLiteOnBgThread = self.numSQLLiteOnBgThread + 1

                matches = re.findall(fileIORegex, line)
                if len(matches) > 0:
                    self.numFileIOOnBgThread = self.numFileIOOnBgThread + 1

                matches = re.findall(bitmapRegex, line)
                if len(matches) > 0:
                    self.numBitmapOnBgThread = self.numBitmapOnBgThread + 1

    def extractLayoutFileData(self, path):
        pass

    def getNumNetworkOnMainThread(self):
        return self.numNetworkOnMainThread

    def getNumSQLLiteOnMainThread(self):
        return self.numSQLLiteOnMainThread

    def getNumFileIOOnMainThread(self):
        return self.numFileIOOnMainThread

    def getNumBitmapOnMainThread(self):
        return self.numBitmapOnMainThread

    def getNumNetworkOnBgThread(self):
        return self.numNetworkOnBgThread

    def getNumSQLLiteOnBgThread(self):
        return self.numSQLLiteOnBgThread

    def getNumFileIOOnBgThread(self):
        return self.numFileIOOnBgThread
```

```
def getNumBitmapOnBgThread(self):
    return self.numBitmapOnBgThread

def printData(self):
    print ""
```

Network Timeout

```
'''
This  class counts the number of networking operations with and without timeout
values.
Created on Jan 27, 2014

@author: ess0006
'''
import re
import fileinput
import AndroidViews

class NetworkTimeout(object):
    '''
    classdocs
    '''


    def __init__(self, sourceCodePaths, layoutPaths):
        '''
        Constructor
        '''
        self.srcpaths = sourceCodePaths
        self.layoutPaths = layoutPaths
        self.numSrcFiles = len(self.srcpaths)
        self.numLayoutFiles = len(self.layoutPaths)
        self.numHTTPClients = 0
        self.numConTimeouts = 0 #connection timeouts
        self.numSoTimeouts = 0 #socket timeouts

    def getNumberofFiles(self):
        return self.numFiles

    def extractData(self):
        for path in self.srcpaths:
            self.extractSrcFileData(path)
        #for path in self.layoutPaths:
            #self.extractLayoutFileData(path)
        self.printData()

    def extractSrcFileData(self, path):
        fileinput.close()
        androidClientRegex = "invoke-static(.*?)Landroid/net/http/AndroidHttpClient;-
>newInstance(.*?)Landroid/net/http/AndroidHttpClient;"
        httpClientRegex = "new-
instance(.*?)Lorg/apache/http/impl/client/DefaultHttpClient"
        conTimeoutRegex = "invoke-
static(.*?)Lorg/apache/http/params/HttpConnectionParams;->setConnectionTimeout"
        soTimeoutRegex = "invoke-
static(.*?)Lorg/apache/http/params/HttpConnectionParams;->setSoTimeout"
        for line in fileinput.input([path]):

            matches1 = re.findall(androidClientRegex, line)
            matches2 = re.findall(httpClientRegex, line)
            if len(matches1) > 0 or len(matches2) > 0:
                    self.numHTTPClients = self.numHTTPClients + 1
            matches3 = re.findall(conTimeoutRegex, line)
            if len(matches3) > 0:
                self.numConTimeouts = self.numConTimeouts + 1
            matches4 = re.findall(soTimeoutRegex, line)
            if len(matches4) > 0:
                self.numSoTimeouts = self.numSoTimeouts + 1
```

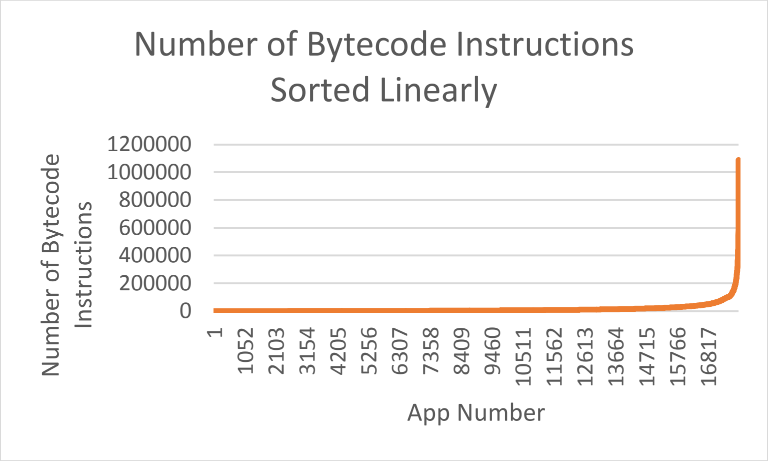
```
def extractLayoutFileData(self, path):
    fileinput.close()
    tempMax = 0
    viewRegex = "<" + AndroidViews.getAndroidViewsRegex()
    for line in fileinput.input([path]):
            numMatches = len(re.findall(viewRegex, line))


def getNumHttpClients(self):
    return self.numHTTPClients

def getNumConTimeouts(self):
    return self.numConTimeouts

def getNumSoTimeouts(self):
    return self.numSoTimeouts

def getNumNoConTimeout(self):
    return self.numHTTPClients - self.numConTimeouts

def getNumNoSoTimeout(self):
    return self.numHTTPClients - self.numSoTimeouts

def printData(self):
    print ""
```

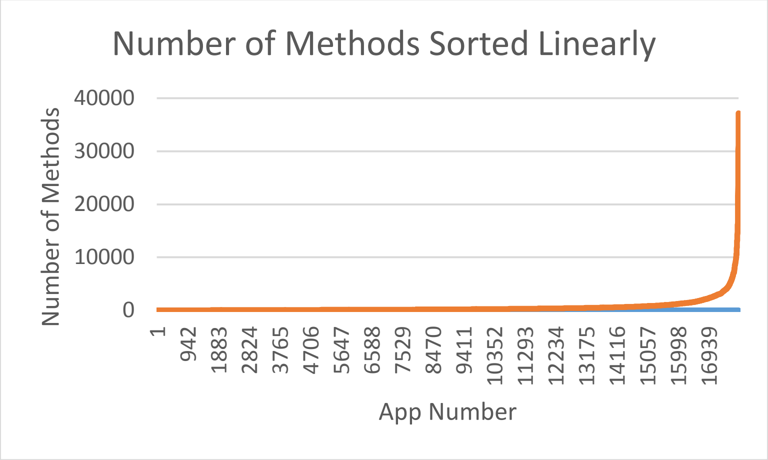
Slide Me Rating and Metadata Collection

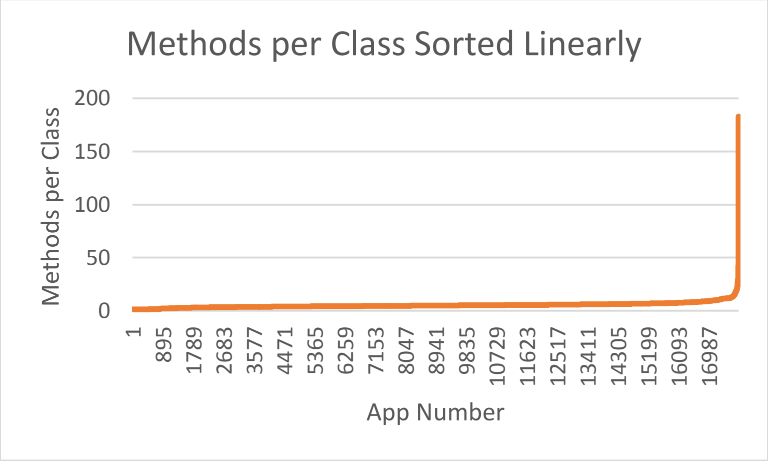
```
'''
This class collects app ratings and other metadata about the slideme apps.
Created on Jan 28, 2014

@author: Eric
'''
import urllib
import re
import os
import ManifestDataExtractor as mde
import sys
import DBWriter

def getHTML(url):
    site = urllib.urlopen(url)
    html = site.read()
    site.close()
    return html

def parseManifest(manifestDataExt, filePath):


    # EXTRACTING APP LABEL
    appLabel = manifestDataExt.extractAppLabel()
    try:
        print "App Label -> " + appLabel
    except:
        appLabel = "DATA NOT FOUND - INVALID ENCODING"
        print "App Label -> " + appLabel

    # EXTRACTING FULLY QUALIFIED APP NAME
    appFQName = manifestDataExt.extractFQName()
    print "Fully Qualified Name -> " + appFQName

    return appLabel, appFQName

if __name__ == '__main__':
    pass


path = "C:\\apks\\decompiled\\slideme"
files = os.listdir(path)
directorySize = len(files)
errorFile = open(path + "_ratings_errors.txt", "w")

print "!"*150
print "NOW EXTRACTING DATA FROM " + path
print str(directorySize) + " APPS FOUND"
print "!"*150

db = DBWriter.DBWriter()
db.connect()

i = 1
for f in files:

    try:
        filename = f + ".apk"

        decFolderPath = "C:\\apks\\decompiled\\slideme" + "\\" + f
        manifestDataExt = mde.ManifestDataExtractor(decFolderPath)


        if manifestDataExt.validateManifest():
            # get app name and package name from manifest
            appname, packageName = parseManifest(manifestDataExt, decFolderPath)
            appname = appname.replace(' ', '-').lower()


            url = "http://slideme.org/application/" + appname
            html = getHTML(url)
            ratingExp = "average-rating\">Average: <span>(\d+\.\d*|\d+)"
            appRating = float(re.findall(ratingExp, html)[0])

            numRatingsExp = "total-votes\">\(<span>(\d+)"
            numRatings = int(re.findall(numRatingsExp, html)[0])

            numDownloadsExp = "downloads\">(\d+)"
            numDownloads = int(re.findall(numDownloadsExp, html)[0])

            db.writeRatingsTable(filename, appRating, numRatings, numDownloads)

            print appname
            print 'Rating: ' + str(appRating)
            print 'Num Ratings: ' + str(numRatings)
            print 'Num Downloads: ' + str(numDownloads)

            i += 1
    except:
            e = sys.exc_info()[0]
            errorFile.write(f + ": " + str(e) + "\n")
print str(i) + " apps examined"
```

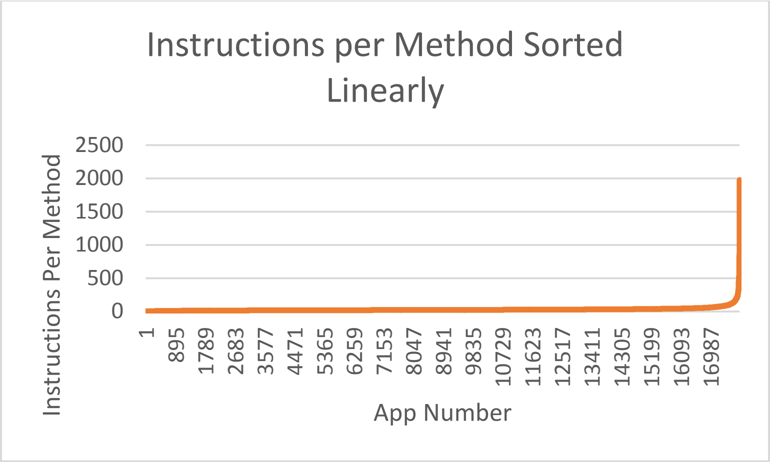
Appendix 2 – Additional Data

Size Metrics

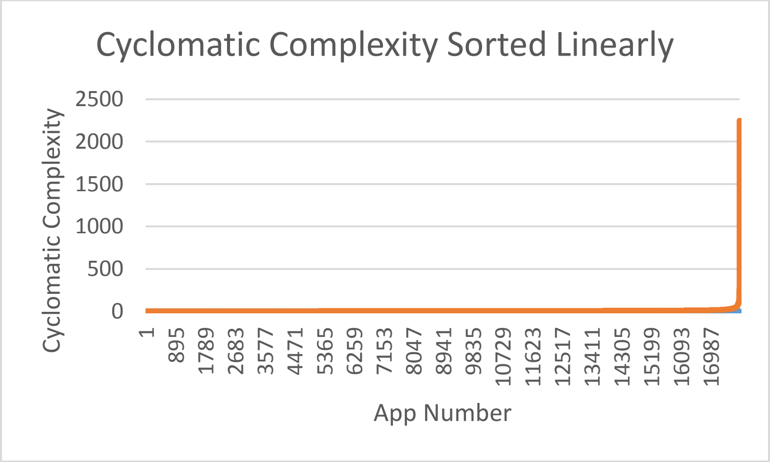


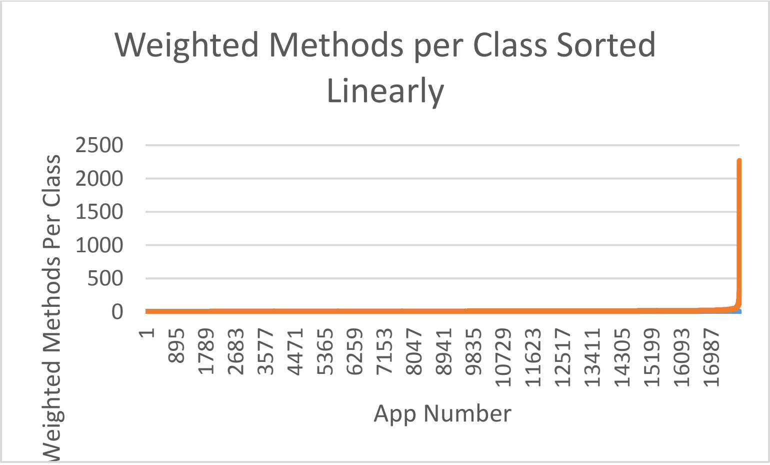




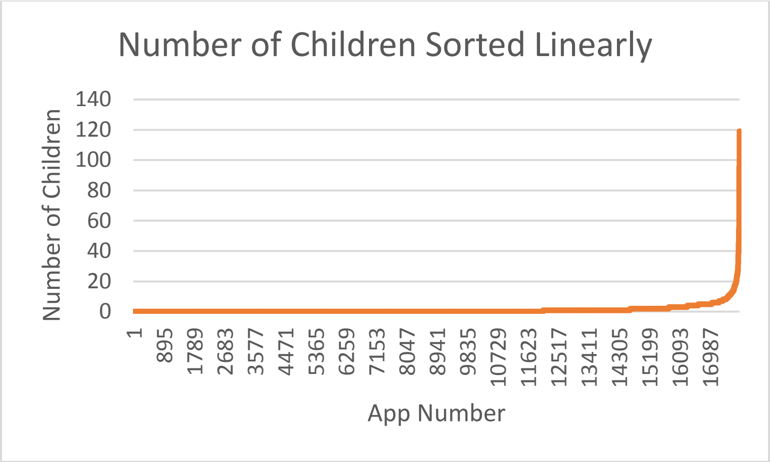


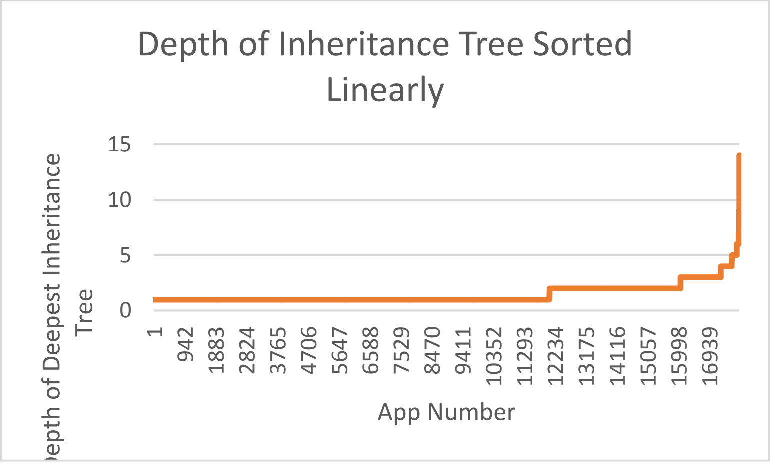
Complexity Metrics

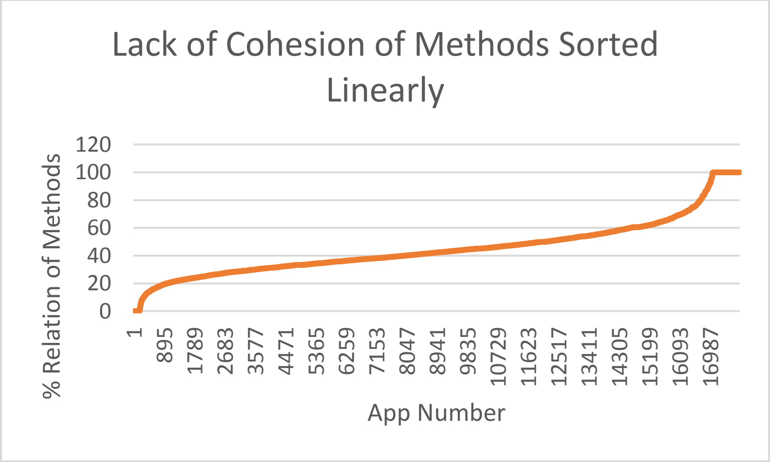


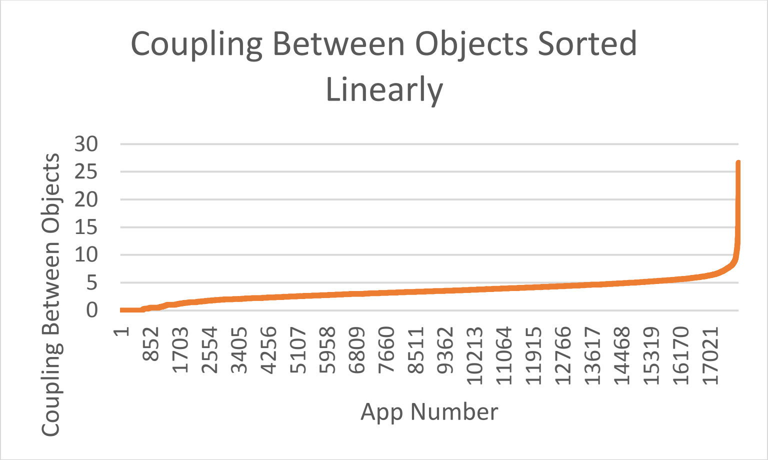


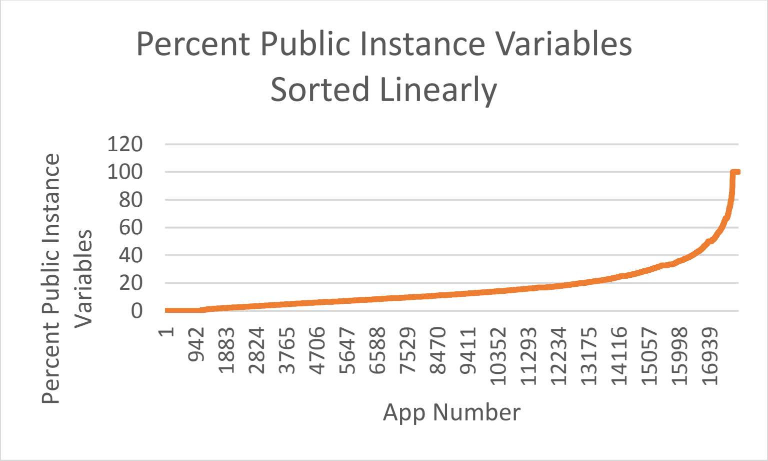
OO Metrics

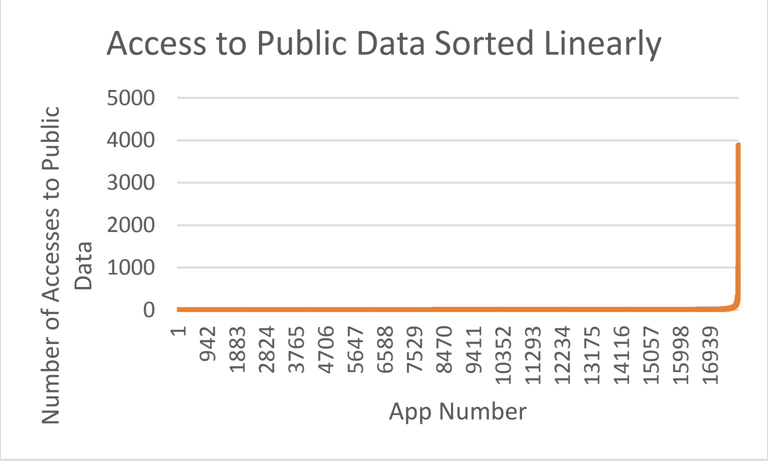

Number of Children Sorted Linearly
Number of Children
140
120
100
80
60
40
20
0
1
895
1789
2683
3577
4471
5365
6259
7153
8047
8941
9835
10729
11623
12517
13411
14305
15199
16093
16987
App Number


Depth of Inheritance Tree Sorted Linearly
epth of Deepest Inheritance
Tree
15
10
5
0
1
942
1883
2824
3765
4706
5647
6588
7529
8470
9411
10352
11293
12234
13175
14116
15057
15998
16939
App Number

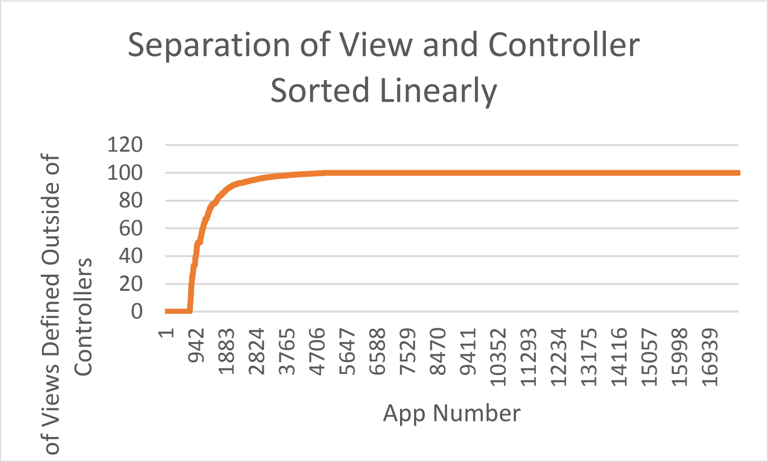

Lack of Cohesion of Methods Sorted Linearly
% Relation of Methods
120
100
80
60
40
20
0
1
895
1789
2683
3577
4471
5365
6259
7153
8047
8941
9835
10729
11623
12517
13411
14305
15199
16093
16987
App Number

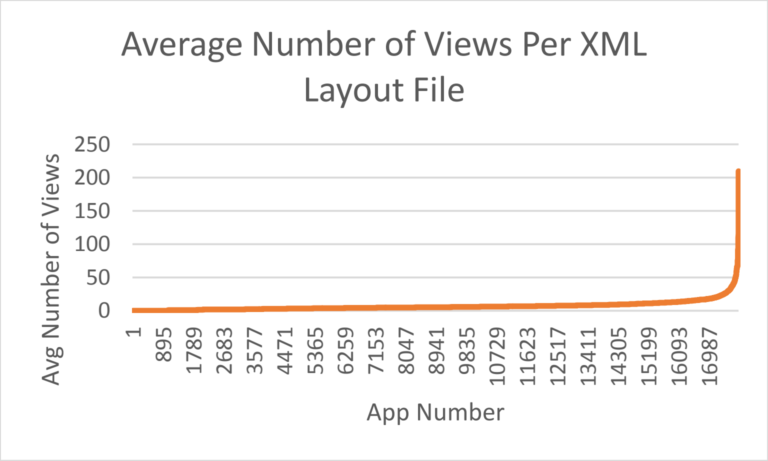

Coupling Between Objects Sorted Linearly
Coupling Between Objects
30
25
20
15
10
5
0
1
852
1703
2554
3405
4256
5107
5958
6809
7660
8511
9362
10213
11064
11915
12766
13617
14468
15319
16170
17021
App Number

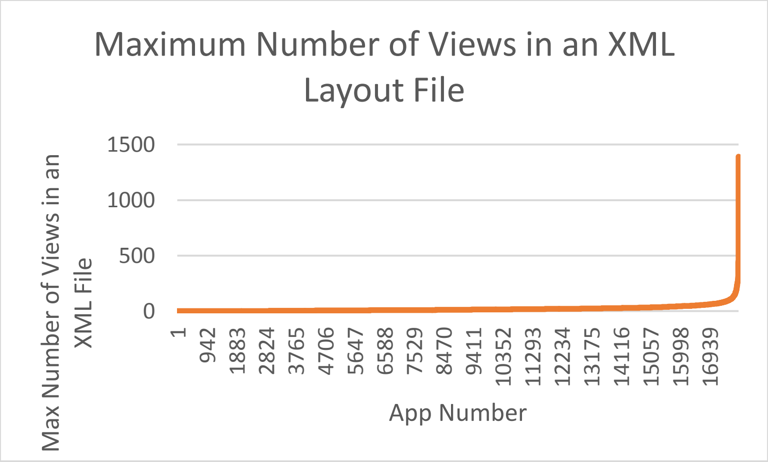

Percent Public Instance Variables Sorted Linearly
Percent Public Instance Variables
120
100
80
60
40
20
0
1
942
1883
2824
3765
4706
5647
6588
7529
8470
9411
10352
11293
12234
13175
14116
15057
15998
16939
App Number


Access to Public Data Sorted Linearly
Number of Accesses to Public Data
5000
4000
3000
2000
1000
0
1
942
1883
2824
3765
4706
5647
6588
7529
8470
9411
10352
11293
12234
13175
14116
15057
15998
16939
App Number

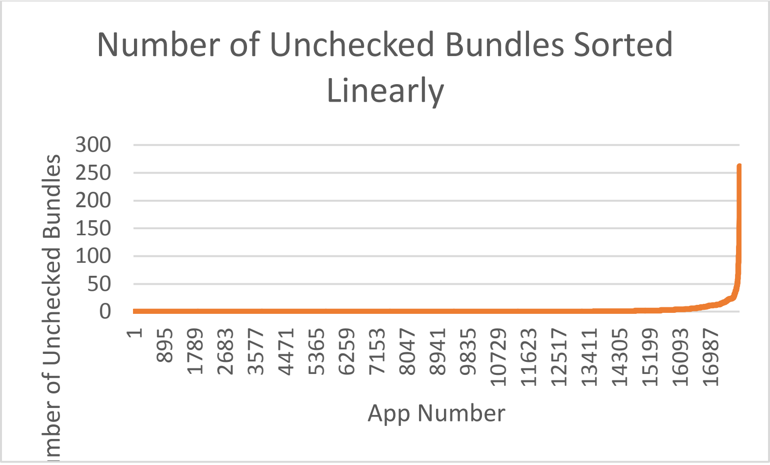
View Metrics



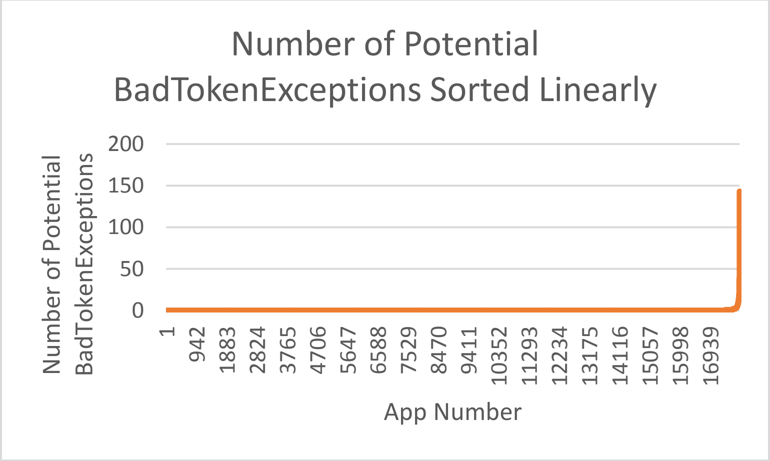




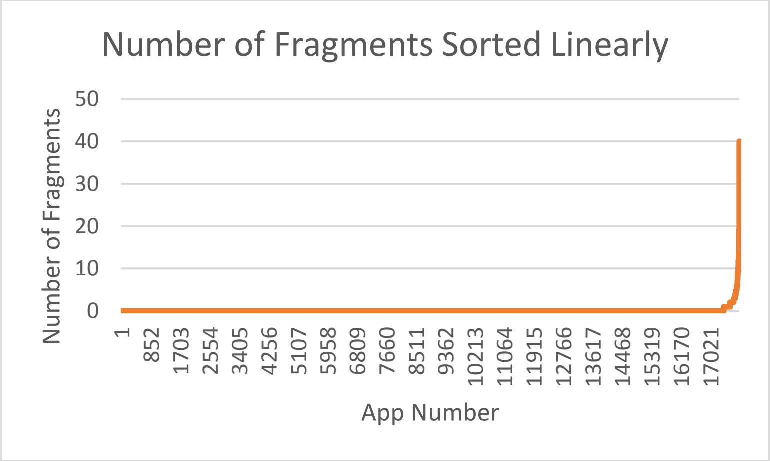
Unchecked Bundles



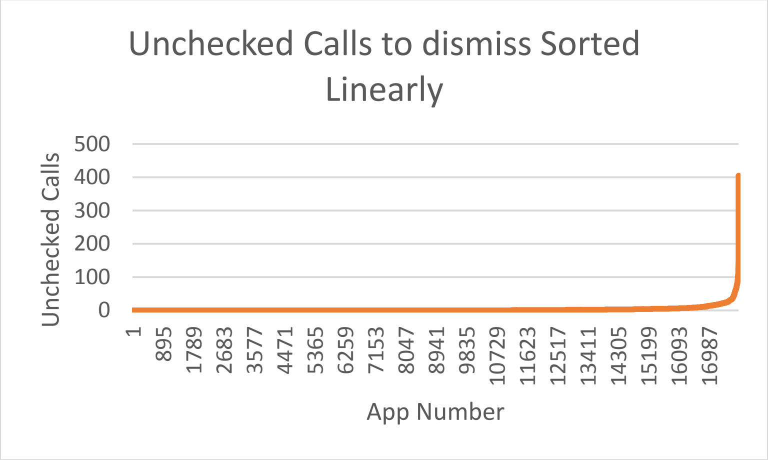
BadTokenException

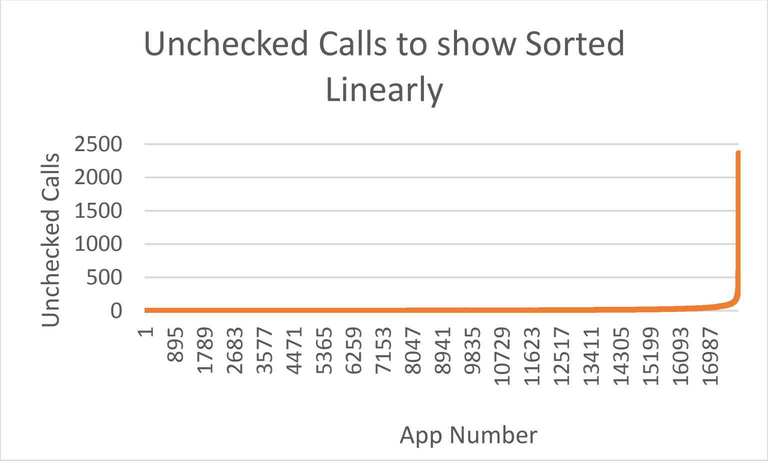


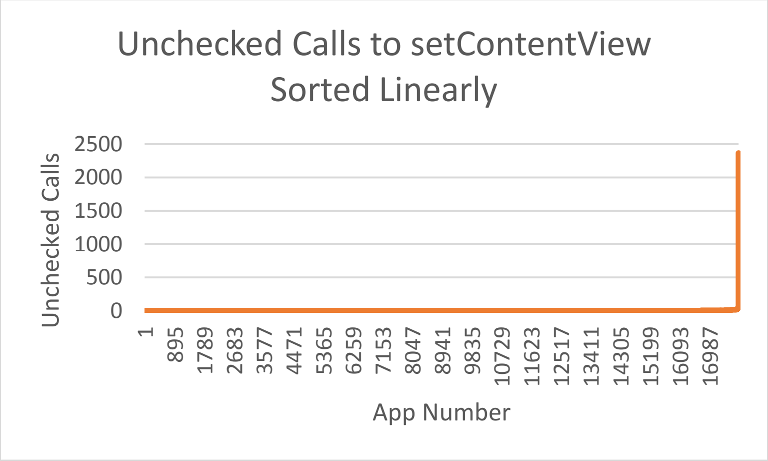
Fragments

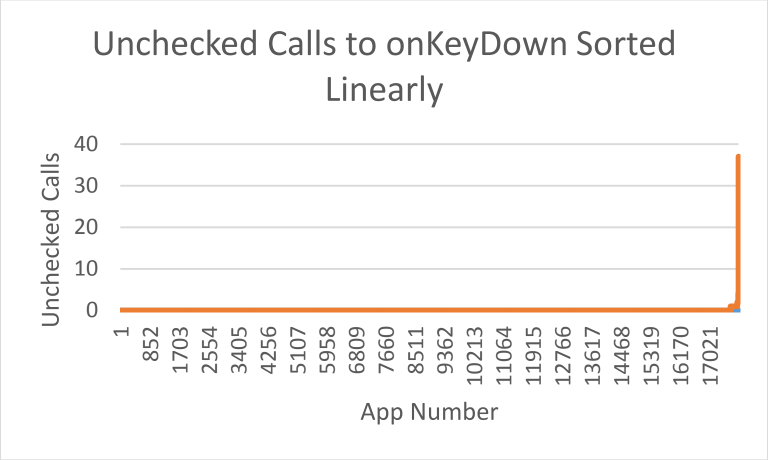


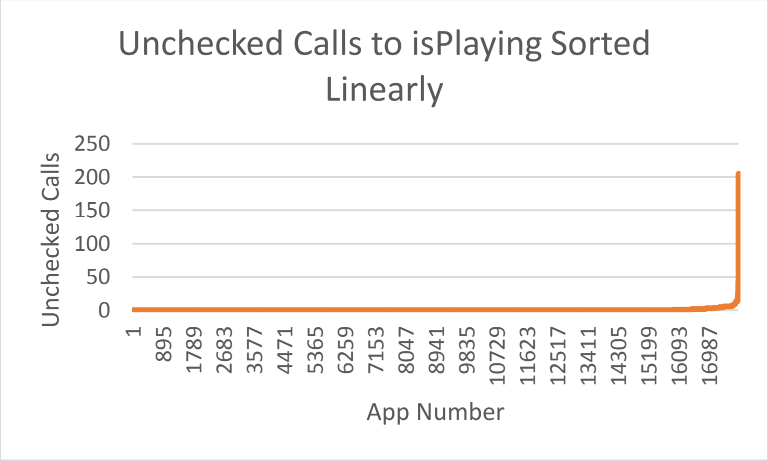
Bad Smell Method Calls

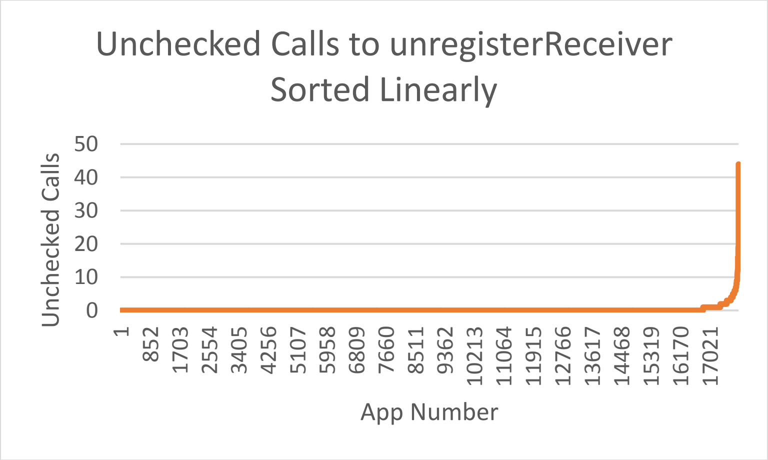


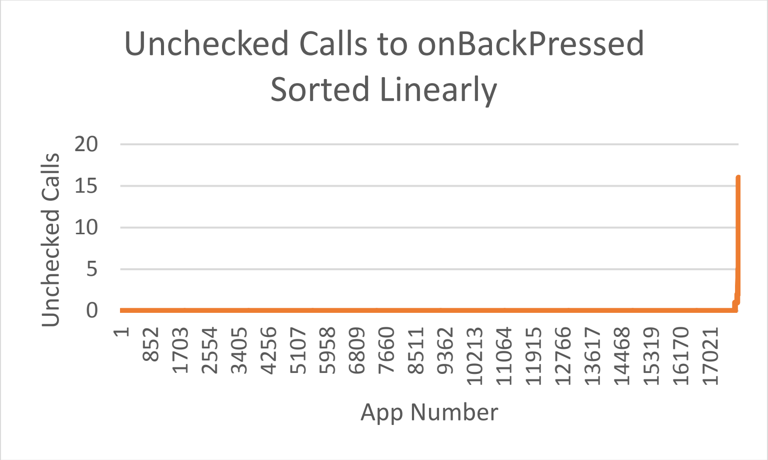


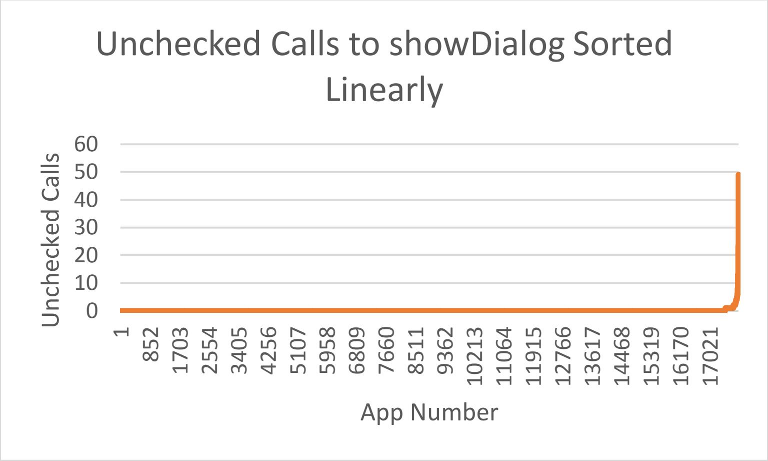


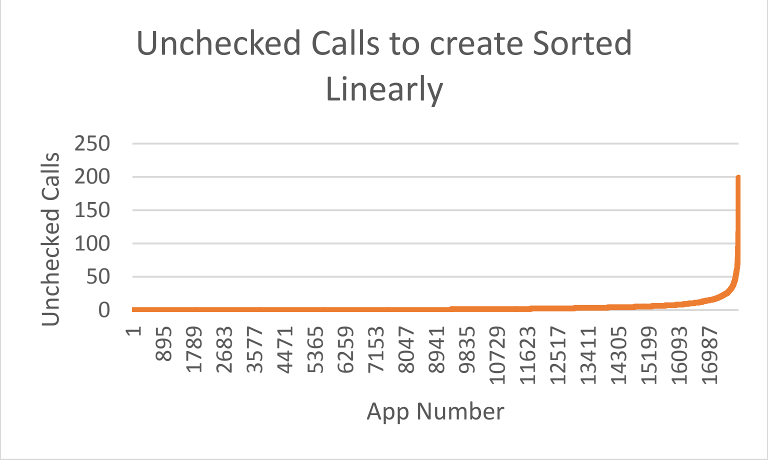

Unchecked Calls to onKeyDown Sorted Linearly
Unchecked Calls
40
30
20
10
0
1
852
1703
2554
3405
4256
5107
5958
6809
7660
8511
9362
10213
11064
11915
12766
13617
14468
15319
16170
17021
App Number


Unchecked Calls to isPlaying Sorted Linearly
Unchecked Calls
250
200
150
100
50
0
1
895
1789
2683
3577
4471
5365
6259
7153
8047
8941
9835
10729
11623
12517
13411
14305
15199
16093
16987
App Number

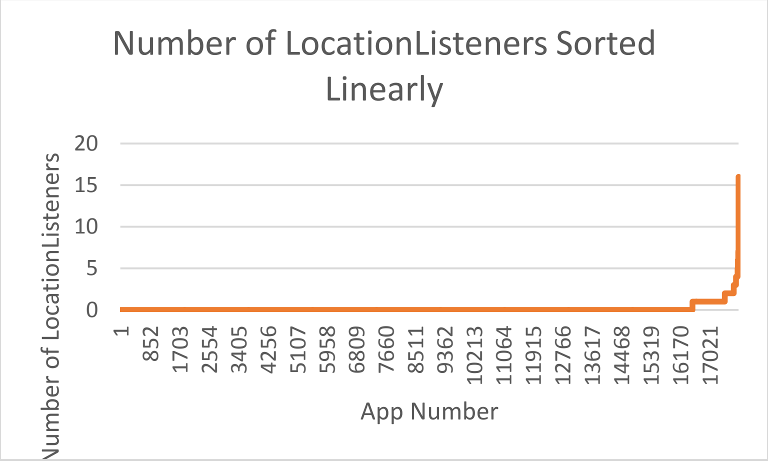

Unchecked Calls to unregisterReceiver Sorted Linearly
Unchecked Calls
50
40
30
20
10
0
1
852
1703
2554
3405
4256
5107
5958
6809
7660
8511
9362
10213
11064
11915
12766
13617
14468
15319
16170
17021
App Number

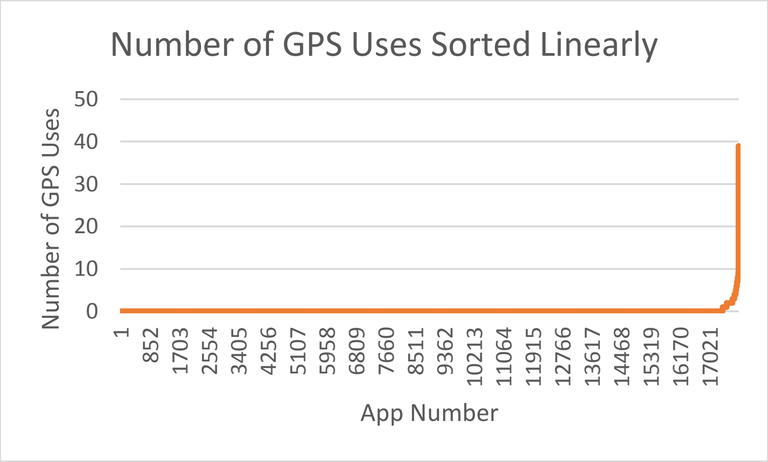

Unchecked Calls to onBackPressed Sorted Linearly

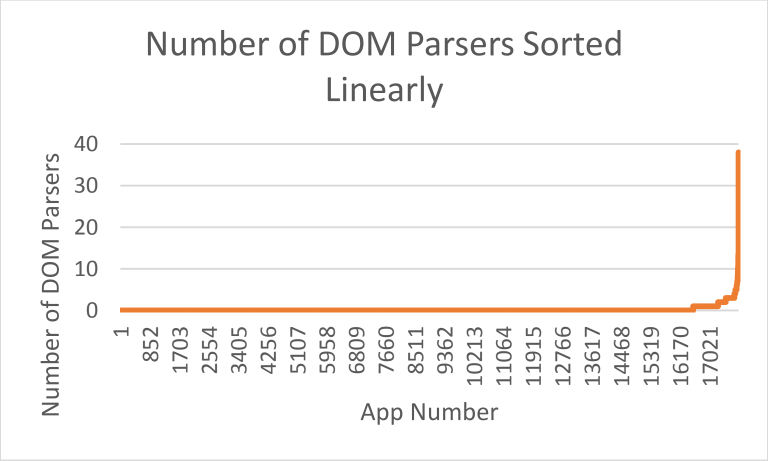

Unchecked Calls to showDialog Sorted Linearly

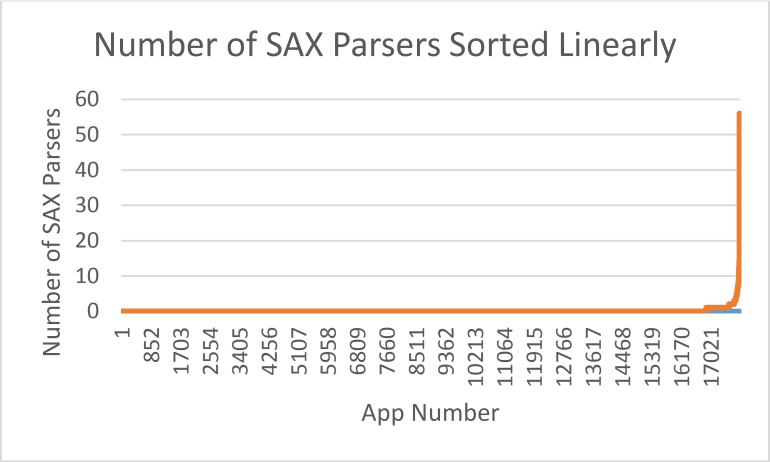

Unchecked Calls to create Sorted Linearly

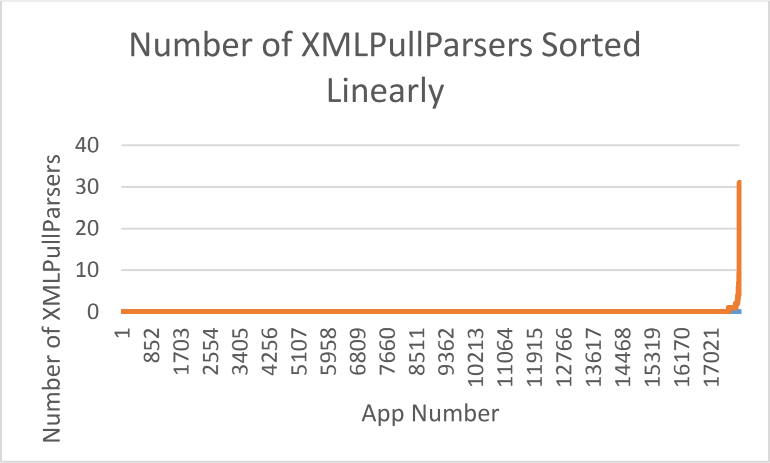
Battery Metrics


Number of LocationListeners Sorted Linearly

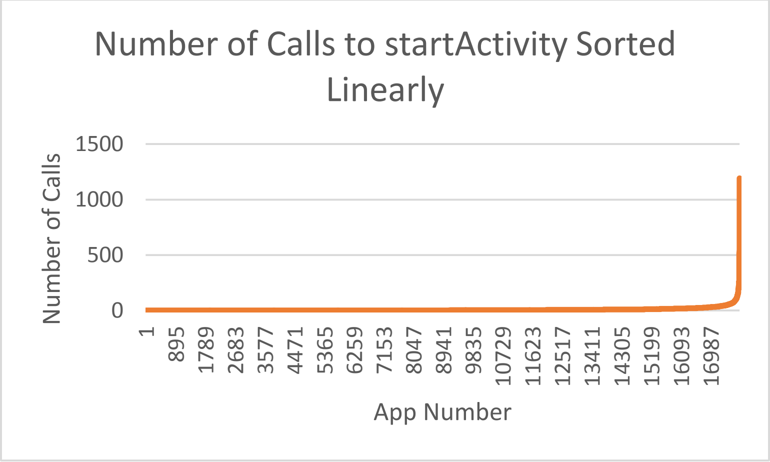

Number of GPS Uses Sorted Linearly

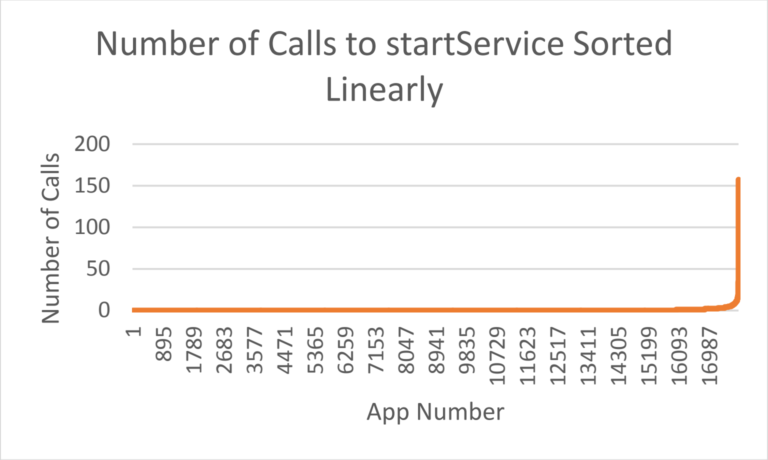

Number of DOM Parsers Sorted Linearly



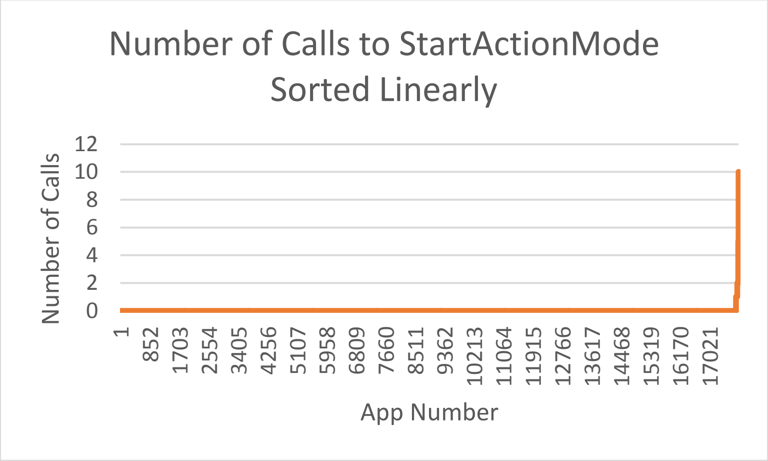


Component Launch Metrics

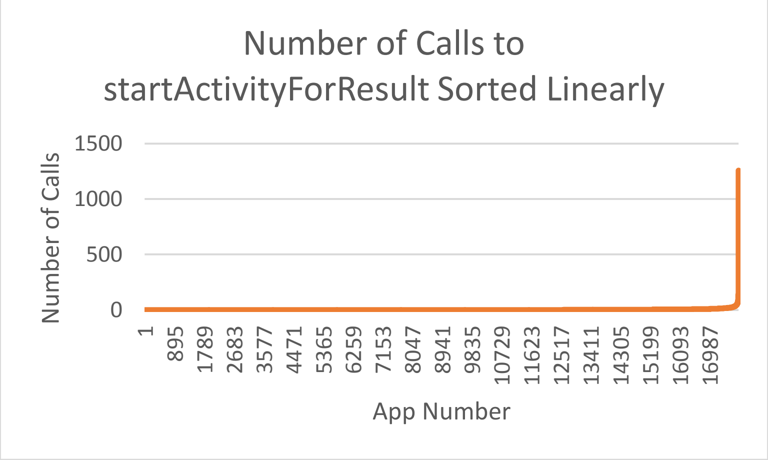




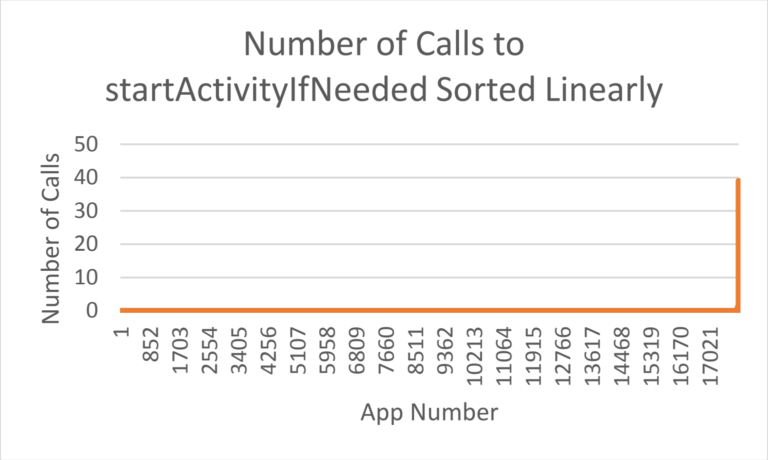
Number of Calls to startService Sorted Linearly

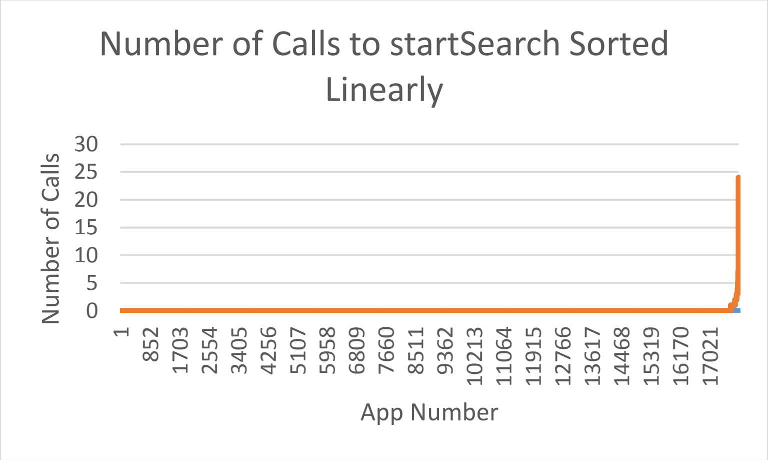


Number of Calls to StartActionMode Sorted Linearly



Number of Calls to startActivityForResult Sorted Linearly



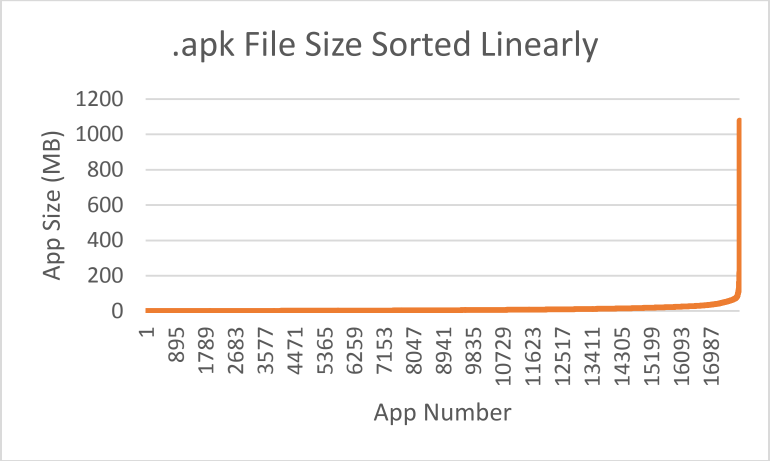


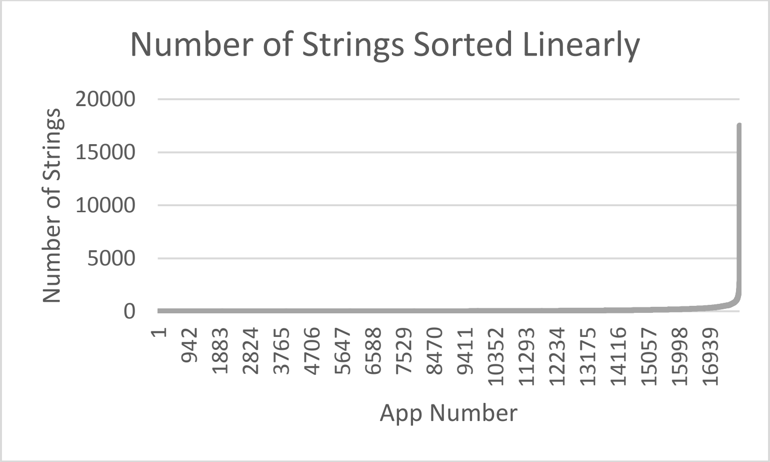
50 Most Frequently Occurring Android Objects

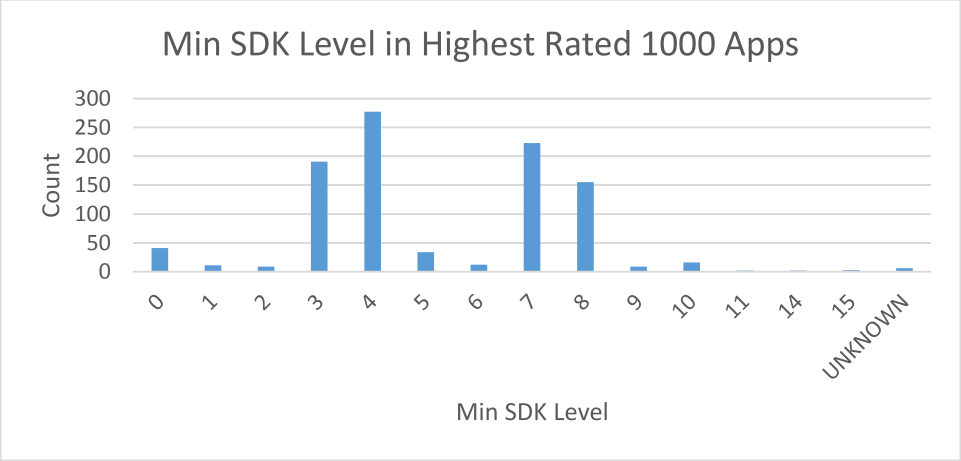
Object	Count
android.os.Bundle	17106
android.content.Context	16911
android.view.View	16434
android.app.Activity	16347
android.content.Intent	15454
android.view.View.OnClickListener	12677
android.content.res/Resources	12528
android.widget.TextView	12275
android.net.Uri	11534
android.content.SharedPreferences	11243
android.app.AlertDialog	11128
android.content.DialogInterface.OnClickListener	10983
android.widget.Button	10972
android.content.DialogInterface	10943
android.app.AlertDialog.Builder	10937
android.util.Log	10921
android.widget.Toast	10520

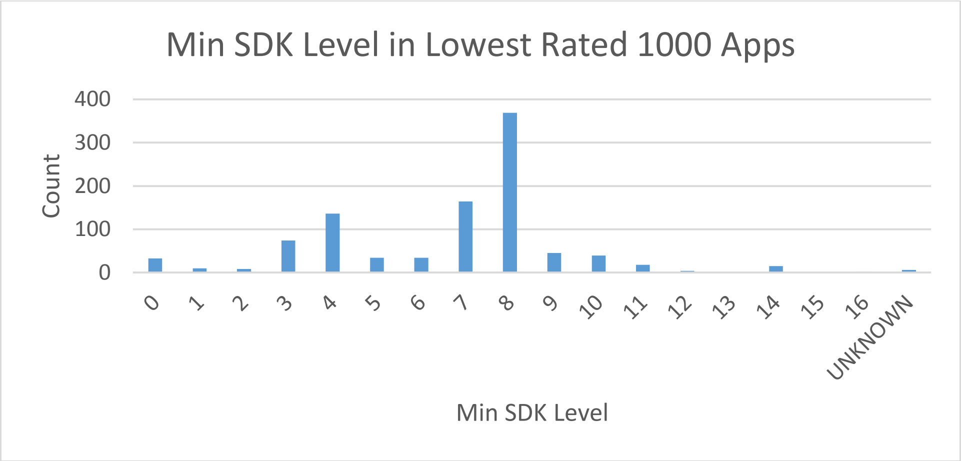
android.content.SharedPreferences.Editor	10391
android.os.Handler	10134
android.view.ViewGroup	9966
android.widget.ImageView	9585
android.graphics.Bitmap	8892
android.view.Menu	8778
android.view.KeyEvent	8638
android.view.MenuItem	8605
android.view.ViewGroup.LayoutParams	8581
android.widget.LinearLayout	8481
android.view.LayoutInflater	8265
android.view.MotionEvent	8166
android.graphics.drawable/Drawable	8113
android.widget.ListAdapter	7902
android.view.Window	7859
android.app.Dialog	7717
android.widget.EditText	7301
android.text.Editable	7267
android.widget.ListView	7060
android.graphics.BitmapFactory	7031
android.widget.AdapterView	6903
android.graphics.Canvas	6730
android.content.pm/PackageManager	6635
android.util.AttributeSet	6562
android.view.WindowManager	6430
android.database.Cursor	6374
android.util.DisplayMetrics	6313
android.view.Display	6292
android.graphics.Paint	6215
android.widget.ArrayAdapter	5844
android.os.Message	5803
android.widget.AdapterView.OnItemClickListener	5803

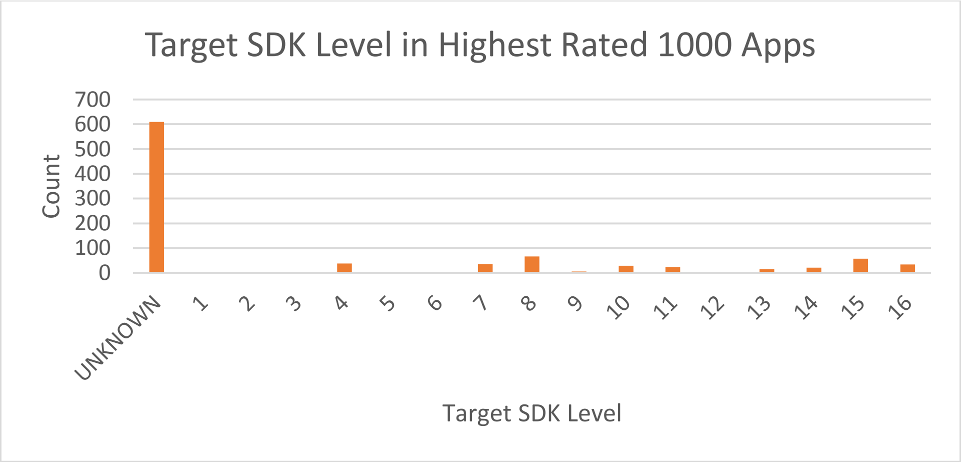
Demographic Metrics

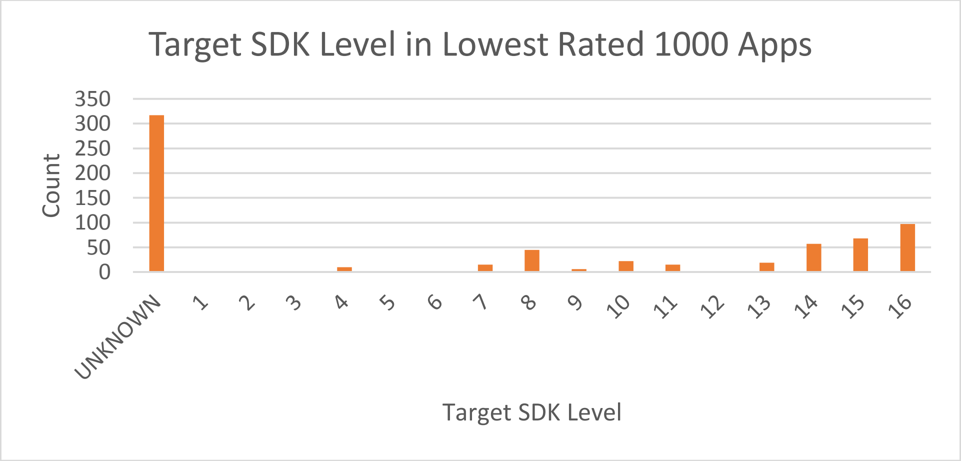




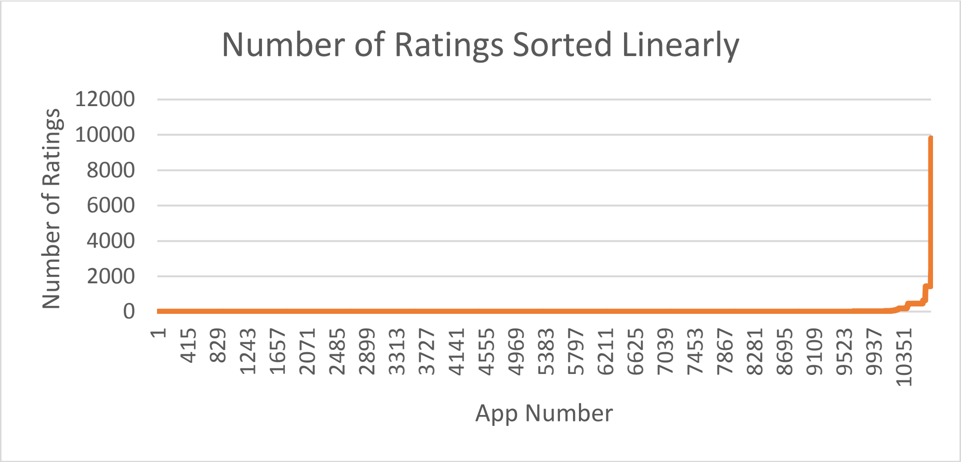


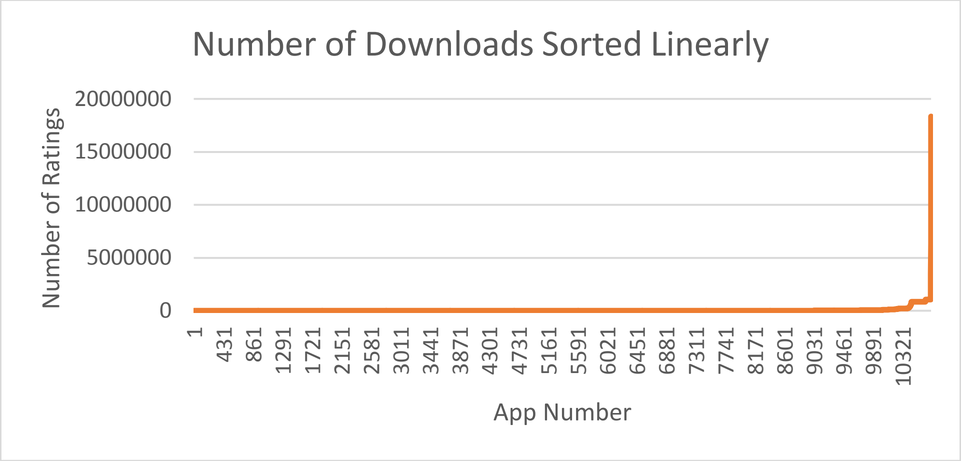

Min SDK Level in Lowest Rated 1000 Apps
Count
400
300
200
100
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
UNKNOWN
Min SDK Level


Target SDK Level in Highest Rated 1000 Apps
Count
700
600
500
400
300
200
100
0
UNKNOWN
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Target SDK Level


Target SDK Level in Lowest Rated 1000 Apps
Count
350
300
250
200
150
100
50
0
UNKNOWN
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Target SDK Level

Metadata for Slide Me Apps


Number of Ratings Sorted Linearly
Number of Ratings
App Number


Number of Downloads Sorted Linearly
Number of Ratings
App Number