Empirical Studies of Mobile Apps and Their Dependence on Mobile Platforms

by

Mark D. Syer

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2013

Abstract

Our increasing reliance on mobile devices has given rise to a new class of software applications (i.e., mobile apps). Tens of thousands of mobile app developers have developed hundreds of thousands of mobile apps that are available across multiple platforms. These apps are used by millions of people around the world every day. However, currently most software engineering research is performed on large desktop or server applications like the Eclipse IDE and Apache HTTPD Server.

We believe that research efforts must begin to examine mobile apps. Mobile apps are growing at a rapid pace, yet they differ from traditionally-studied desktop and server applications.

In this thesis, we examine such apps by performing three quantitative studies. First, we study differences in the size of the code bases and development teams of desktop/server applications and mobile apps. We then study differences in the code, dependency and churn properties of the mobile apps of two different mobile platforms. Finally, we study the impact of size, coupling, cohesion and code reuse on the quality of mobile apps.

Some of the most notable findings of this thesis are that mobile apps are much smaller than traditionally-studied desktop/server applications and that most mobile apps tend to be developed by only one or two developers. Mobile app developers

tend to rely heavily on functionality provided by the underlying mobile platform through platform-specific APIs. We find that Android app developers tend to rely on the Android platform more than BlackBerry app developers rely on the BlackBerry platform. We find that defects in Android apps tend to be concentrated in a small number of source code files. We also find that source code files that depend on the Android platform tend to have more defects.

Our results indicate that major differences exist between mobile apps and traditionally-studied desktop/server applications. However, the mobile apps of two different mobile platforms also differ. Further, our results suggest that mobile app developers should avoid excessive platform dependencies and focus their testing efforts on source code files that rely heavily on the underlying mobile platform. Given the widespread use of mobile apps and the lack of research surrounding these apps, we believe that our results will have significant impact on software engineering research.

Co-Authorship

Earlier versions of the work in this thesis were published as listed below:

- *Exploring the Development of Mobile Apps Across the Android and BlackBerry Platforms (Chapter 4)*

  Mark D. Syer, Bram Adams, Ying Zou and Ahmed E. Hassan, Exploring the Development of Micro-Apps A Case Study on the BlackBerry and Android Platforms, in Proceedings of the 11th International Working Conference on Source Code Analysis and Manipulation (SCAM), Sept. 2011. (Acceptance rate: 16 / 52 = 30.7%)

  My contribution – drafting the research plan, gathering and analyzing the data, writing the manuscript and presenting the paper.

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. Ahmed E. Hassan, for his constant support and advice. Few people have had such a profound impact on my life and I am deeply indebted to Ahmed.

I am deeply grateful to Dr. Bram Adams and Dr. Meiyappan Nagappan who have been excellent colleagues and mentors. Bram's energy and dedication has been truly inspiring and Mei's insightful feedback on my thesis has been invaluable.

My sincerest thanks to my colleagues at the Software Analysis and Intelligence Lab (SAIL) who are outstanding researchers, role models and friends. I consider myself lucky to have worked with them.

I would like to thank Queen's University and the School of Computing for providing a fantastic environment for work and study.

I would also like to express my deepest thanks to my friends, and my roommates in particular, who patiently stood by while I completed my thesis and helped provide a relaxing environment outside the labs.

Finally, I would like to dedicate this work to my parents. This thesis would not have been possible without their continuous support and encouragement.

Statement of Originality

I, Mark D. Syer, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobile devices have changed the software development world by providing a platform for the rapid emergence of a new class of software applications (i.e., mobile apps). Mobile apps, commonly referred to as apps, distinguish themselves from the applications that run on typical desktops or servers by their limited functionality, low memory and CPU footprint, touch interfaces, and limited screen sizes and resolutions. Similar to web applications (Hassan and Holt, 2002), mobile apps are rapidly developed by small teams who may only have limited experience with software development (Butler, 2011; Gavalas and Economou, 2011; Lohr, 2010; Wen, 2011).

Mobile apps have become hugely popular among both consumers and developers since Apple opened its App Store in 2008 (Chetan Sharma Consulting, 2010; Gartner Inc., 2011). Downloads of mobile apps have risen from 7 billion in 2009 to 15 billion in 2010 and 50 billion in 2012 (projected) (Chetan Sharma Consulting, 2010). The annual number of mobile app downloads may even reach 183 billion by 2016 (International Data Corp., 2011). Simultaneously, the number of mobile apps has also increased. Table 1.1 shows the number of mobile apps available for the iOS, Android and BlackBerry mobile operating systems. Thousands of developers have been drawn to mobile app ecosystems and more than 140,000 developers had made their apps available through app stores by July 2011 (research2guidace, 2011).

Table 1.1: Number of Mobile Apps By Platform (Android Market, 2012; Apple App Store, 2012; BlackBerry App World, 2012).

Company - Platform	Number of Mobile Apps		
	September 2010	September 2011	September 2012
Apple - iOS	250,000	350,000	650,000
Google - Android	80,000	300,000	500,000
RIM - BlackBerry	10,000	15,000	100,000

Despite the ubiquity of mobile devices and the popularity of mobile apps, few software engineering researchers have studied mobile apps. Software engineering researchers have proposed and evaluated several empirical theories of how high quality, successful software is developed and maintained. These "software engineering empirical theories" aim to tie aspects of software artifacts (e.g., size and complexity) (Chidamber and Kemerer, 1994), their development (e.g., number of changes) (Nagappan and Ball, 2005) and their developers (e.g., developer experience) (Bird et al., 2011) to various definitions of software quality (e.g., number of defects). However, such empirical theories have primarily been evaluated against large-scale projects such as the Mozilla Firefox web browser, Eclipse IDE, Linux kernel and Apache HTTPD web server (Brian et al., 2010). The relationship between these large-scale projects and mobile apps and the applicability of these empirical theories to mobile apps is unclear.

1.1 Research Statement

The mobile app sector is rapidly becoming the largest sector of software today. Yet there is very limited research done to understand the development practices and the quality of such mobile apps. We believe that these mobile apps bring a unique set of challenges to software engineering practice and research.

1.2 Thesis Overview

We conduct three empirical studies to better understand mobile app development practices and the quality of such apps.

1. *Revisiting Empirical Observations Using Mobile Apps (Chapter 3)*

   In our first study, we revisit several empirical observations regarding development practices in the context of mobile apps. We also compare our findings to traditionally-studied desktop/server applications.

   We find that the development practices of mobile apps differ from traditionally-studied systems – highlighting the need for software engineering researchers to closely examine this rapidly growing sector.

2. *Studying the Development of Mobile Apps Across the Android and BlackBerry Platforms (Chapter 4)*

   In our second study, we seek to understand whether development practices vary between different mobile platforms.

   We find that development practices do differ between platforms – highlighting the importance of using case studies from different platforms in future empirical studies regarding mobile apps.

3. *Platform Dependence and Source Code Quality (Chapter 5)*

   In our third study, we examine the quality of mobile apps with a specific focus on their dependence on their underlying mobile platform.

   We find that their high dependence on their platform plays a significant role in their quality; source code files are more defect-prone when they are highly dependent on the mobile platform.

1.3 Major Thesis Contributions

In this thesis, we empirically study mobile apps and their dependence on mobile platforms. In particular, we make three contributions to our understanding of mobile apps.

1. This thesis presents the first study to explore some of the characteristics of mobile app development and compares mobile apps to traditionally-studied desktop/server applications – highlighting the importance of studying such apps and the risks of blindly applying prior empirical theories to the world of mobile apps.

2. We present the first study of its kind to compare mobile apps across two of the most popular mobile platforms.

3. We are the first to revisit empirical theories of software quality in the context of mobile apps, while considering some of the unique characteristics of mobile apps (i.e., high platform dependence).

1.4 Organization of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 describes keys terms used throughout this paper and provides an overview of current research regarding mobile apps.

Chapter 3 presents our study of two differences (i.e., the size of the code base and development and the time to fix defects) between desktop/server applications and mobile apps.

Chapter 4 presents a study of three differences (i.e., source code, dependency and code churn) between Android and BlackBerry apps.

Chapter 5 presents a study of four factors (i.e., lines of code, coupling, cohesions and the degree of dependence on a mobile platform) that influence the quality of mobile apps.

Finally, Chapter 6 concludes the thesis with a discussion of our results, the limitations and threats to validity of our research and potential directions for future research.

Chapter 2

Background and Related Work

The rise of mobile apps is a relatively recent trend in software engineering. However, mobile apps are beginning to garner interest within the software engineering community and the future importance of mobile apps is being recognized. Researchers are beginning to explore the challenges, issues and opportunities in studying mobile apps and platforms (Workshop on Mobile Software Engineering, 2011). For example, the focus of the mining challenge at the international working conference on mining software repositories (MSR) in 2012 was to uncover interesting findings related to the Android platform (Shihab et al., 2012).

Researchers have also studied mobile apps from other perspectives.

Education – Teng and Helps have designed a project for an introductory level operating systems course in which 35 students were asked to develop a mobile app for one of the major mobile platforms. Upon completion of the course, the authors surveyed the students to evaluate the overall development and learning experience (Teng and Helps, 2010). The authors found that the majority of the students who took the course felt that mobile app development is an important component of an information technology curriculum. Hu et al. have also developed a course focusing on mobile app development (Hu et al., 2010). The course combines a detailed course syllabus and hands-on-labs to give students an overview of mobile devices and mobile platforms and prepare students to develop mobile apps.

Security – Enck et al. have studied Android's security model and outlined the complexity of developing secure mobile apps (Enck et al., 2009). Shabtai et al. performed a security assessment of the Android platform, identified potential security threats and provided solutions for mitigating these threats (Shabtai et al., 2010). Grace et al. have performed large-scale studies of Android apps to detect zero-day Android malware (Grace, Zhou, Zhang, Zou and Jiang, 2012) and security and privacy threats from mobile app advertising libraries (Grace, Zhou, Jiang and Sadeghi, 2012).

However, to date, there are few studies of mobile apps from a software engineering perspective

Research regarding mobile app development has largely focused on a small number of specific issues.

Development tools and frameworks – Gasimov et al., Hammershoj et al., and Tracy surveyed the challenges facing mobile app developers (Gasimov et al., 2010; Hammershoj et al., 2010; Tracy, 2012). These challenges include working within a highly fragmented marketplace and working within the constraints of mobile devices. In addition, Gasimov et al. also surveyed the tools available to mobile app developers. Charland and LeRoux compared two methods of mobile app development (i.e., web apps and native apps) to identify the strengths and weaknesses of each method (Charland and LeRoux, 2011). Although web apps are cheaper to develop and deploy than native apps, native apps have much better performance. However, the gap between web apps and native apps is quickly closing with new and improving technologies (e.g., 3D in-browser gaming with WebGL).

Cross platform development – Wu et al. use a mobile web application engine that can run C++ code on the Android platform through the Java Native Interface (Wu et al., 2010). Xin presents two development tools, Swerve Studio and X-Forge, that can be used to achieve cross-platform development for mobile games (Xin, 2009).

Research regarding the software quality aspects of mobile apps is largely lacking, however, researchers are beginning to study the platforms that support these apps. Maji et al. studied bug reports in the Android and Symbian platforms to understand which modules are most defect-prone and how defects are fixed (Kumar Maji et al., 2010). The authors determine that development tools, web browsers and multimedia modules are most defect-prone and that most defects are fixed with minor code changes. The authors also determine that despite the high cyclomatic complexity of the Android and Symbian platforms, defect densities are surprisingly low. In this thesis, we study Android apps, not the Android platform itself. Han et al. have also studied bug reports in the Android platform (Han et al., 2012). The authors used topic analysis to compare the bug reports from two different Android vendors (i.e., HTC and Motorola). They found that bug report topics differed between the two Android vendors and concluded that the fragmentation of the Android platform has led to significant quality (i.e., compatibility and portability) issues.

Researchers are also beginning to study code reuse in mobile apps. Mojica et al. performed a case study of code reuse in 4,323 Android apps and found that, on average, 61% in the classes in a mobile app are reused by other mobile apps in the same domain, e.g., social networking (Israel et al., 2012). The authors also found that 23% of the 534,057 classes in their case study inherit from a base class in the Android platform. Given the wide-spread dependence on the Android platform, it is important to study the source code quality implications of this dependence.

Finally, researchers are also beginning to study the larger mobile app ecosystem (i.e,. mobile app stores). Harman et al. studied 32,108 paid (i.e., not free) mobile apps in BlackBerry App World (Harman et al., 2012). The authors found that there is a strong correlation between customer ratings and downloads, while there was no correlation found between price and either ratings or downloads. The authors found that these correlations are stronger within subdomains of the app store (e.g, games and utilities).

In this thesis, we study mobile apps from a software engineering perspective (e.g., source code quality). Such research on mobile apps is largely lacking in the existing software engineering literature.

Chapter 3

Revisiting Empirical Theories Using Mobile Apps

Despite the ubiquity of mobile devices and the popularity of mobile apps, few software engineering researchers have studied them. During the thirty-five years following Fred Brooks' seminal text, The Mythical Man Month (Brooks, 1975), the software engineering community has proposed, evaluated and noted several theories of how high quality, successful software is developed and maintained. These "software engineering empirical theories" aim to tie aspects of software artifacts (e.g., size and complexity) (Chidamber and Kemerer, 1994), their development (e.g., number of changes) (Nagappan and Ball, 2005) and their developers (e.g., developer experience) (Bird et al., 2011) to definitions of quality (e.g., post-release defects). Such empirical theories are derived from a large number of empirical observations. However, to date, most observations have been made using large-scale projects such as Apache and Eclipse (Brian et al., 2010). Determining how these empirical theories hold in mobile apps may reduce the effort in developing and maintaining high quality mobile apps.

Our existing software engineering knowledge may not hold in mobile apps due to differences between the mobile app and desktop/server ecosystems. Two potentially influential differences are 1) the hardware limitations and diversity of mobile devices and 2) the distribution channel provided by app stores.

First, the hardware limitations of mobile devices has led to mobile apps with small memory and CPU footprints that are intended for mobile devices with small screen sizes. These hardware limitations may limit the scope of mobile apps. Indeed, even the latest generation of mobile devices does not meet the minimum system requirements for best-selling games (e.g., World of Warcraft) and applications (e.g, Adobe Photoshop CS5). Further, the shift in usage from desktop/server systems to mobile devices (mobile devices are intended to be used "on-the-go") limits the scope of mobile apps.

While the capabilities of mobile devices have been rapidly increasing (e.g., many new devices boast dual-core processors) some of these limitations are characteristic of mobile devices. This is particularly true of the screen size and resolution, which limits the information and functionality displayed at one time. Further, the diversity of hardware (e.g., accelerometers and touch interfaces with gesture recognition) may complicate development as developers experiment with these features. A mobile platform (e.g., Android) consists of numerous APIs that provide mobile apps with an interface to these hardware accessories. In addition, platform APIs provide access to commonly required functionality.

The diversity and limitations of mobile device hardware may limit the scope of mobile apps. Therefore, our first research question compares the size of the code base and development team of open source Android apps and desktop/server applications.

Second, major mobile platforms (e.g., Android) provide centralized app stores for users to download mobile apps. Centralized app stores are easily browsed and directly available from a user's device. Therefore, the effort to install mobile apps is minimal. Further, the cost of listing a mobile app in an app store is very low (occasionally there is no cost) while the potential to reach millions of consumers is high.

The low cost to enter the mobile market and the potential to generate revenue has attracted a large number of developers (Chetan Sharma Consulting, 2010; ComScore Inc., 2010; Gartner Inc., 2011). Tens of thousands of developers have made their apps available through centralized app stores (All Facebook, 2010; research2guidace, 2011). Further, development tools have been released to the general public so that anyone can develop a mobile app, even without prior development experience (Butler, 2011; Gavalas and Economou, 2011; Lohr, 2010; Wen, 2011). The large and constantly expanding development community leads to high competition between developers. For example, Google Play contains hundreds of competing weather apps, media players and instant messaging apps.

The high competition between developers may affect the quality of mobile apps for fear of losing users to competing apps. The emphasis on defect fixing may be high and hence the defect fix times may be low. Therefore, our second research question compares the time it takes to fix defects in open source Android apps and desktop/server applications.

As a first step towards understanding how established software engineering empirical theories hold in mobile apps, we perform an exploratory study to compare mobile apps and desktop/server applications. Are there clear differences between the size of the code base/development team and quality improvement process (i.e., the time it takes to fix a defect) of these two classes of software? We study fifteen open source Android apps and five desktop/server applications (two large, commonly studied applications and three smaller Unix utilities). Our study addresses two research questions:

- *RQ1: How does the size of the code base compare between mobile apps and desktop/server applications?* – mobile apps and Unix utilities tend to have smaller code bases than larger desktop/server applications. In addition, the development of mobile apps and Unix utilities tends to be driven by one or two core developers. Further, mobile apps tend to rely heavily on their underlying platform.

- *RQ2: How does the defect fix time compare between mobile apps and desktop/server applications?* – mobile app developers typically fix defects faster than desktop/server developers, regardless of the size of the project. In addition, defects in mobile apps tend to be concentrated in a smaller portion of the source code files.

In the course of addressing our research questions, we reevaluate three empirical theories:

1. *Core developers* – empirical studies show that a small subset of the developers (≈20%) are responsible for the majority (≈80%) of the development and maintenance of an application(Dinh-Trong and Bieman, 2005; Geldenhuys, 2010; Loukas et al., 2011; Mockus et al., 2000, 2002).

2. *Defect Reporters* – empirical studies show that the number of people reporting defects is an order of magnitude larger than the number of developers (Dinh-Trong and Bieman, 2005; Mockus et al., 2002).

3. *Defect distribution* – empirical studies show that the majority of defects (≈80%) occur in a small subset (≈20%) of the source code files (Gittens et al., 2005; Ostrand et al., 2005).

The chapter is organized as follows: Section 5.1 describes the setup of our case study and Section 5.2 describes and discusses the results. In Section 5.3, we outline the threats to the validity of our case study. Finally, Section 5.4 concludes the chapter.

3.1 Case Study Setup

This section outlines our approach to exploring the differences between mobile apps and desktop/server applications. First, we selected representative mobile apps and desktop/server applications. Second, we measured the size of the code base and the time it takes to fix defects for each of the subject apps/applications. We then compared the measurements across the subjects.

3.1.1 Mobile App Selection

In this chapter, we study mobile apps written for the Android platform. The Android platform is the largest (by user base) and fastest growing mobile platform. In addition, the Android platform itself is open source and has more free mobile apps than any other major mobile platform (Distimo, 2011*a*).

Mobile apps for Android devices are primarily hosted in Google Play (Android Market, 2012). The Android Market classifies mobile apps into two groups (i.e., free and paid) and records details such as cost and the number of times each app has been installed in the previous 30 days. We use the Android Market to list the top 2,000 free apps and the top 2,000 paid apps (2,000 is the maximum number of apps that the Android Market ranks). However, we limit our study to free Android apps, because 1) the majority (63%) of Android apps are free (Distimo, 2011*a*), 2) free apps are downloaded significantly more than paid mobile apps (Distimo, 2011*b*) and 3) the source code repositories and issue tracking systems are not available for paid apps.

The term "free" within Google Play refers to mobile apps that are downloaded at no cost. Free apps are not necessarily open source. One reason is that many of these apps are developed internally by organizations as mobile interfaces to their online services (e.g., Facebook, Google Maps and Skype). Another reason is that many paid apps have versions that are available for free. This is because a common revenue model is to use a multi-tiered structure (e.g., a free app with ads and a paid app without ads, or a limited-time demo version and a paid app). However, we are unable to study these apps because they are only available as bytecode files (i.e., we do not have access to their source code repositories or issue tracking systems). Therefore, we must determine which apps in the top 2,000 free app list are open source.

While most Android apps are hosted in Google Play, there are other app stores as well (Android Market, 2012; Chetan Sharma Consulting, 2010). One such app store, albeit a small one, is the FDroid repository (F-Droid, 2012). The FDroid repository exclusively lists free and open source (FOSS) Android apps that are also listed in Google Play. The FDroid repository contains links to the homepage, source code repository and issue tracking system (if available). However, it does not contain any information regarding the user base, e.g., the number of downloads.

We select apps for our case study by cross-referencing the list of the top 2,000 free apps in Google Play, with the list of apps in the FDroid repository. Mobile apps in the resulting list are 1) popular amongst users (allowing us to study "successful" apps) and 2) open source (allowing us to access their source code and issue tracking systems). The resulting list contains twenty mobile apps. However, we exclude five mobile apps from our case study: three did not have an issue tracking system, one had a different source control system (other than SVN/GIT) and one was a mobile port of a desktop application (we could not differentiate the code of the mobile app from the desktop application). Therefore, our case study includes fifteen open source Android apps.

Our selection of mobile apps based on popularity (downloads) has one caveat. The number of downloads is not an ideal measure of success (unlike user retention or engagement). However, it is the best measure currently available.

Table 5.1 contains the list of mobile apps that we include in our case study. We assign an ID to each mobile app for brevity. For each of the selected mobile apps, Table 5.1 contains 1) a brief description, 2) the minimum number of downloads during July 2011 (Google Play releases monthly download numbers within ranges), 3) the date of the first commit to the repository (our analysis was performed on the source code repository and issue tracking system from the date of the first commit until August 8, 2011) and 4) link to the homepage (which contains links to the source code and issue tracking systems). Table 5.1 contains mobile apps from a number of different domains, including utilities, networking, multimedia and games.

Table 3.1: Selected Mobile Apps

ID	Name	Description	Downloads	First Commit	Homepage
M1	3	Music Player	>500,000	26/01/2010	github.com/fabrantes/rockonnggl/
M2	Apps Organizer	Utility	>1,000,000	15/08/2009	code.google.com/p/appsorganizer/
M3	Barcode Scanner	Utility	>10,000,000	23/10/2007	code.google.com/p/zxing/
M4	ConnectBot	SSH Client	>1,000,000	16/11/2007	code.google.com/p/connectbot/
M5	Cool Reader	E-Book Reader	>500,000	09/03/2007	sourceforge.net/projects/crengine/
M6	Frozen Bubble	Game	>1,000,000	15/11/2009	code.google.com/p/frozenbubbleandroid/
M7	K-9 Mail	Email Client	>1,000,000	27/10/2008	code.google.com/p/k9mail/
M8	KeePassDroid	Password Vault	>100,000	01/24/2009	google.com/p/keepassdroid/
M9	Quick Settings	Utility	>1,000,000	25/05/2010	code.google.com/p/quick-settings/
M10	Reddit is fun	Social Networking	>100,000	07/30/2009	github.com/talklittle/reddit-is-fun/
M11	Scrambled Net	Game	>500,000	26/07/2009	code.google.com/p/moonblink/
M12	Sipdroid	VOIP client	>500,000	04/04/2009	code.google.com/p/sipdroid/
M13	Solitaire	Game	>10,000,000	18/11/2008	code.google.com/p/solitaire-for-android/
M14	Tricorder	Game	>500,000	25/10/2009	code.google.com/p/moonblink/
M15	Wordpress	CMS Client	>500,000	10/09/2009	android.svn.wordpress.org/

Table 3.2: Selected Desktop/Server Applications

ID	Name	Description	First Commit	Last Commit	Homepage
D1	Apache HTTP Server	HTTP Server	07/03/1996	07/02/2011	httpd.apache.org/
D2	Eclipse UI Component	User Interface	05/23/2001	09/12/2011	eclipse.org/eclipse/development/
D3	aspell	Spell Checker	01/01/2000	01/01/2004	savannah.gnu.org/cvs/?group=aspel
D4	joe	Text Editor	04/19/2001	04/19/2005	sourceforge.net/projects/joe-editor/
D5	wget	File Retriever	02/12/1999	01/11/2003	savannah.gnu.org/bzr/?group=wget

3.1.2 Desktop/Server Application Selection

In this chapter, we use two types of desktop/server applications.

First, we study large, commonly studied desktop applications. In particular, we study the User Interface (UI) component of the Eclipse platform and the Apache HTTP server projects. These applications are two of the most commonly studied desktop/server applications in software engineering literature (Brian et al., 2010). We have extended our analysis to these projects so that our results may be viewed in the context of two of the most commonly studied desktop/server software applications. We do not claim that these projects are the baseline against which all projects should be measured. However, we wish to determine the similarities and differences between these two projects and our mobile apps as a first step towards understanding how the software engineering empirical theories developed from studying desktop/server applications hold in mobile apps.

Second, we study smaller Unix utilities. In particular, we study the aspell, joe and wget Unix utilities. These applications are rarely studied by software engineering researchers, however, they may be more similar to mobile apps than the larger, more commonly studied desktop/server applications. Similar to mobile apps, these applications have small feature sets and are readily available to large user bases (typically pre-installed on all Unix-like operating systems). To enhance the similarity between these applications and mobile apps, we consider only the first four years of development (i.e., four years from the date of the initial commit to the source code repository). Therefore, the modified aspell, joe and wget data sets represent relatively young projects with small feature sets that are available to a large user base (these characteristics are very similar to the mobile apps in our case study).

Table 3.2 contains the list of desktop/server applications that we include in our case study. We have assign an ID to each desktop/server application for brevity. For each desktop/server application, Table 3.2 contains 1) a brief description and 2) link to the homepage (which contains links to the source code and issue tracking systems).

3.1.3 Research Questions

As a first step toward determining how existing software engineering knowledge holds in mobile apps, we perform an exploratory study comparing mobile apps and desktop/server applications. We study fifteen open source Android apps, two large desktop/server applications (Apache HTTP server and Eclipse UI component) and three smaller Unix utilities (aspell, joe and wget) to address the following two research questions:

- RQ1: How does the size of the code base compare between mobile apps and desktop/server applications?
- RQ2: How does the defect fix time compare between mobile apps and desktop/server applications?

3.2 Case Study Results

RQ1: How does the size of the code base compare between mobile apps and desktop/server applications?

Motivation

Many problems typically addressed by software engineering researchers and faced by developers are caused by large code bases and development teams. For example, the difficulty of code navigation increases as the code base grows. In addition, the size of the code base, the number of source code files and lines of code, has been shown to be highly correlated to the complexity of a software application (Herraiz et al., 2007; Lind and Vairavan, 1989). The difficulty of understanding, evolving and maintaining a software application is strongly tied to the complexity of the application. Therefore, we measure the size of the code base of mobile apps and compare it to small Unix utilities and large desktop/server applications.

Approach

We explore the size of the code base by extracting the total number of lines of code and number of source code files. We use the Understand tools by Scitools (Scitools, 2012) to extract these metrics. Understand is a mature toolset of static source code analysis tools for measuring and analyzing software projects written in a number of programming languages. Table 3.3 presents the total number of source code files and lines of code in the code base of each mobile app and desktop/server application. These measurements were made on the latest version of the projects, either 1) August 8, 2011 for the mobile apps in Table 5.1 or 2) the date of the last commit for the desktop/server applications in Table 3.2.

Table 3.3: Size of the Code Base

ID	#Files	LOC
M1	69	20,050
M2	131	12,282
M3	237	22,785
M4	229	32,692
M5	39	14,014
M6	15	2,846
M7	157	47,927
M8	306	23,808
M9	63	5,288
M10	43	10,524
M11	6	2,332
M12	250	24,259
M13	14	4,196
M14	32	5,470
M15	64	17,287
Median	**64**	**14,014**
D1	386	123,293
D2	2,360	276,980
Median	**1373**	**200,137**
D3	129	12,311
D4	91	27.419
D5	61	17,376
Median	**91**	**17,376**

Table 3.4: Size of the Development Team

ID	#Devs	LOC/#Devs	#Core Devs	LOC/#Core Devs
M1	4	5,013	1	20,050
M2	1	12,282	1	12,282
M3	11	2,071	2	11,393
M4	12	2,724	1	32,692
M5	12	1,168	2	7,007
M6	1	2,846	1	2,846
M7	16	2,995	4	11,982
M8	4	5,952	1	23,808
M9	1	5,288	1	5,288
M10	21	501	1	10,524
M11	1	2,332	1	2,332
M12	22	1,103	5	4,852
M13	2	2,098	1	4,196
M14	2	2,735	1	5,470
M15	2	8,644	1	17,287
Median	**4**	**2,735**	**1**	**10,524**
D1	96	1,284	27	4,566
D2	71	3,901	18	15,388
Median	**84**	**2,593**	**23**	**9,977**
D3	3	4,104	1	12,311
D4	6	4,570	2	13,710
D5	6	2,896	1	17,376
Median	**6**	**4,104**	**1**	**13,710**

Results

Size of the Code Base

From Table 3.3, we find that the size of the selected mobile apps ranges between 2,332 and 47,927 lines of code with a median value of 14,014. This is the same order of magnitude as aspell, joe and wget. However, this value is approximately 9 times smaller than the Apache HTTP server project and 20 times smaller than the Eclipse UI component (note that the Eclipse UI component is only one component of the larger Eclipse development environment).

From Table 3.3, we find that many games tend to be small. For example, M6 is 2,846 LOC, M11 is 2,332 LOC, M13 is 4,196 LOC and M14 is 5,470 LOC. These apps offer simple board games, puzzles and card games. Conversely, the largest mobile app, M7, is a full-fledged email client designed to replace the default email client that ships with an Android device. M7 has an extensive feature set, including support for IMAP, POP3 and Exchange 2003/2007, multiple identities, customizable viewing preferences and alternate themes.

Since the selected mobile apps tend to have small code bases, we further explore two factors that may influence the size of the code base. First, we explore the size of the development team. Second, we explore platform usage.

Discussion

Size of the Development Team

One factor that may influence the size of the code base is the number of developers. If few developers contribute to a project, then the size of the project is limited by the developers combined effort.

We explore the size of the development team by extracting the number of developers and the number of source code commits that each developer makes to a project. We extract the commit logs from the source code repository and isolate the committer field for each source code commit, i.e., a commit that includes at least one source code file (e.g., ".java," or ".c" files). To measure the number of *unique* developers, we extract the local part of each email address (i.e., the characters before the @ symbol), and count the number of unique local parts across all source code commits. We then count the number of source code commits made by each developer. We perform manual checks to verify that the list of developers contains only unique entries and we merge any uncaught cases (e.g., john_doe@gmail.com and doe.john@google.com). We then calculate the minimum number of developers who, when combined, are responsible for at least 80% of the commits. These "core" developers are responsible for the majority of the development and maintenance effort (Mockus et al., 2000, 2002). Table 3.4 presents the number of developers and core developers for each mobile app and desktop/server application and the average number of lines of code per developer and core developer.

From Table 3.4, we find that the number of core developers participating in the development and maintenance of one of the selected mobile app is approximately the same as the number of developers participating in aspell, joe and wget, but much smaller than the number of core developers participating in the Apache HTTP server and Eclipse UI component. We also find that, despite the large variability in the number of developers among the selected mobile apps, from 1 to 22 developers, the number of core developers is one in three quarters of the selected mobile apps. Therefore, the development of mobile apps tends to be driven by a single developer.

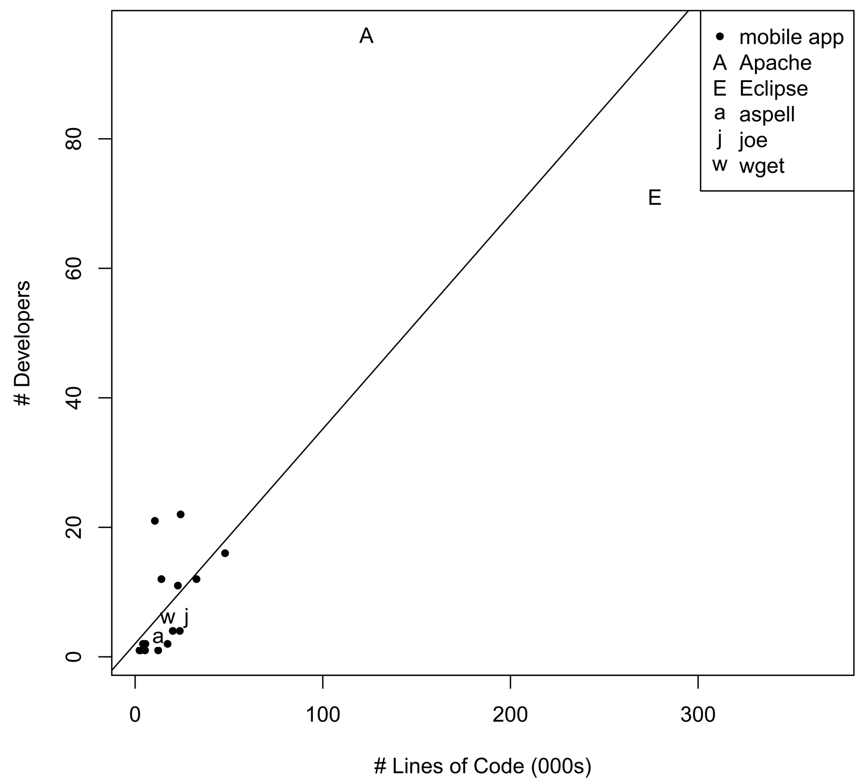


Figure 3.1: Size of the code base and development team.

We further explore the size of the development team by comparing the size of the code base and development team. Figure 3.1 shows the number of developers plotted against the number of lines of code in the code base. Figure 3.1 also shows a trend line generated by fitting a straight line to the data from our mobile apps.

From Figure 3.1, we find that the number of developers among the selected mobile apps increases as the number of lines of code increases. However, this trend line is not a perfect fit to the data, indicating that the relationship between the size of the code base and development team varies between projects. This is supported by Table 3.4. From Table 3.4, we find that the average number of lines of code per developer varies significantly, from 501 to 12,282 LOC.

There are many reasons why mobile apps have varying number of developers. Some developers request that the user community contribute towards the project. For example, the core developer of M10 informed users that he was unable to continue maintaining the project and asked the user community contribute. Conversely, some developers act as gate keepers to their source code repository and are responsible for each commit. For example, the developer of M2 asked that translations of his app be submitted via email, whereas translators were given commit privileges in M11 (many of whom would later contribute defect fixes).

Platform Usage

Another factor that may influence the size of the code base is the reuse of existing functionality through libraries. The Android and Java APIs (Android apps are written in Java) provide basic and commonly required functionalities, thereby reducing the size of the code base and the amount of functionality that the development team must implement themselves.

We explored the extent of platform usage by extracting the number of references (e.g., method calls) to the Android library. We used the Understand tool to extract, for each source code file in the subject mobile apps, a list of classes on which the file depends. We classify these dependencies into one of the following categories based on the class name:

- Platform – a dependency on a class that is part of the mobile platform (e.g., `android.app.Activity`).
- Other – a dependency on a class in the code base or Java platform (e.g., `java.io.IOException`).

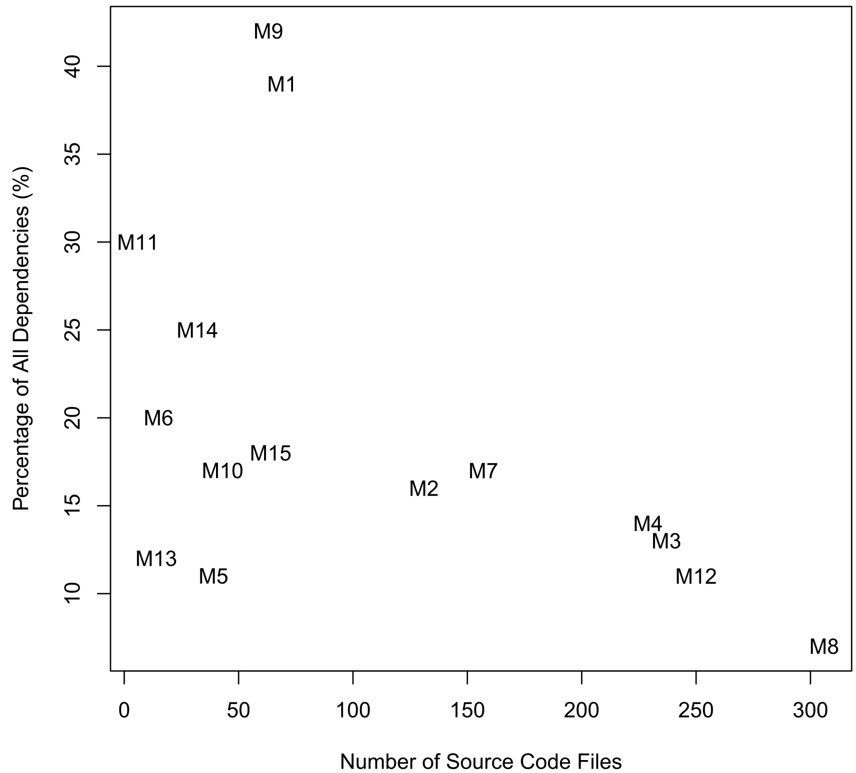


Figure 3.2: Size of the code base and platform usage.

We then determine platform usage, i.e., the ratio of the number of platform dependencies to the total number of dependencies. Figure 3.2 presents platform usage plotted against the number of source code files for each mobile app.

From Figure 3.2, we find that, as the number of source code files increases, the extent of platform usage also decreases. This indicates that the smaller mobile apps in our case study tend to depend on the Android platform more heavily than the larger mobile apps.

Platform usage is high (42%) in M9 (Quick Settings). The app is designed to interface with the Android platform and give users control over device settings, e.g., screen brightness. In contrast, platform usage is low (7% and 11% respectively) in M8 (KeePassDroid) and M5 (Cool Reader). KeePassDroid is a port of the KeePass Password safe and Cool Reader is a cross-platform e-reader.

Uncovering the rationale for the relatively small code bases in mobile apps requires more analysis of other systems and consumer usage trends (are consumer demands satisfied by mobile apps with limited functionality?), however, our current findings suggest the following:

One potential factor that may have led to relatively small code bases may be the fact that fewer developers contribute to mobile app projects. Another potential factor that may have led to relatively small code bases is the reuse of source code. Mobile app developers tend to leverage the functionality provided by the Android platform.

> We find that mobile app and Unix utilities have smaller code bases and development teams compared to commonly studied desktop/server applications. We also find that mobile apps tend to rely heavily on the underlying mobile platform.

RQ2: How does the defect fix time compare between mobile apps and desktop/server applications?

Motivation

Software quality is a major draw for users, and with access to thousands of mobile apps through the app store, users demand the highest quality. If a user installs a low quality app from the app store, he or she can typically find and install a replacement from the app store within minutes. Competition may force developers to place a greater emphasis on fixing defects, or risk losing users to competing apps. Thus, the emphasis on defect fixing, and hence defect fix times, may differ between mobile apps and desktop/server applications. Therefore, we must compare the defect fix times of mobile apps and desktop/server applications.

Approach

We explore the time it takes to fix defects by extracting the list of issues that have been resolved as "closed" with a "fixed" status from the issue tracking system. We do not include issues that we classify as "not a bug," "duplicates" or "not reproducible", because these issues did not involve any time to fix. The resulting issues describe unique defects that have been fixed. We then calculate the number of days between the date the issue was opened and the date it was closed. For issues with multiple close dates (i.e., issues that have been reopened) we take the last close date. We then calculate the percentage of defects that have been fixed within one week, one month and one year of being reported. Figure 3.3 presents these measures for each mobile app and desktop/server application and Table 3.5 presents the median value of these measures across all 1) mobile apps, 2) large desktop/server applications and 3) Unix utilities.

Table 3.5: Median Percentage of Defects Fixed in One Week/Month/Year

	Week	Month	Year
Mobile Apps	36	68	100
Large Desktop/Server Applications	33	69	92
Unix Utilities	21	36	80

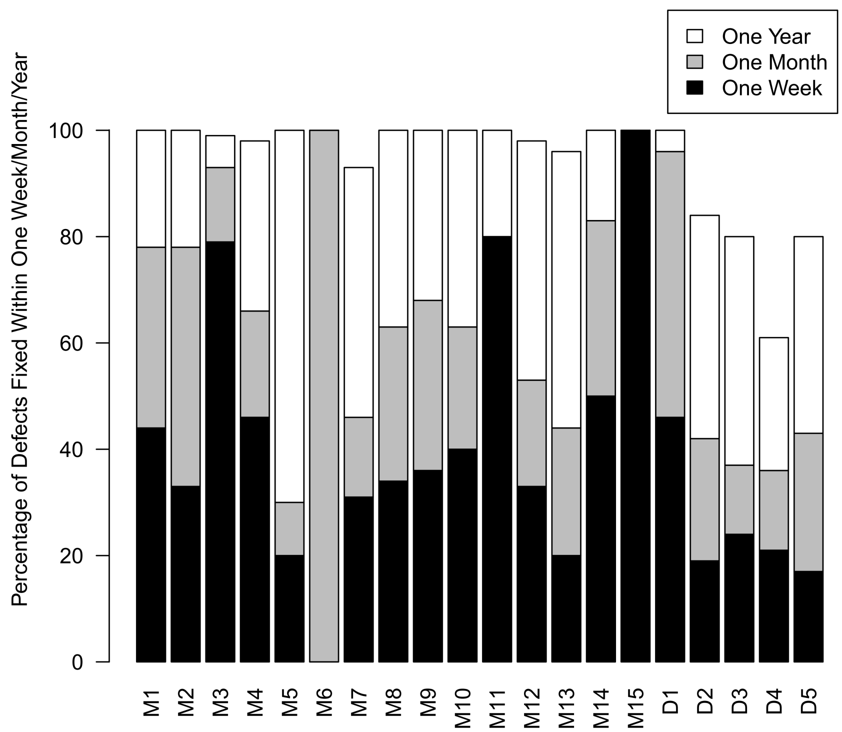


Figure 3.3: Percentage of Defects Fixed in One Week, One Month and One Year.

Results

Defect Fix Times

From Figure 3.3, we find that developers of four of the selected mobile apps fix 50% of reported defects in one week and developers of eleven of the selected mobile apps fix 33% of reported defects in one week. This is greater than the aspell, joe and wget utilities, which fix 24%, 21% and 17% of the reported defects in one week respectively. This is also greater than the Eclipse UI component, where 19% of the reported defects are fixed in one week. However, it is less than the Apache HTTP server, where 46% of the reported defects are fixed in one week.

The number of defects fixed within one week in M6 (Frozen Bubble) is zero because only a single defect was reported in total (it was fixed in twenty-six days).

From Table 3.5, we find that the defect fix times in the selected mobile apps are more similar to large desktop/server applications than Unix utilities. For example, the median percentage of defects fixed within one week is 36% in mobile apps and 33% in large desktop/server applications compared to only 21% in Unix utilities.

From Figure 3.3, the percentage of reported defects fixed in one year in D4 is 61%, compared to 80% in D3 and 80% in D5. However, the developers of D4 close groups of defects at the same time. For example, between May 5, 2003 and February 1, 2006, the developers did not close any issues, however, on February 2, 2006, ten issues were closed (on average, these ten issues were open for three years). Therefore, it is possible that developers fix defects within a short time span, but fail to update the issue tracking system until a later date.

Since the defect fix times of the selected mobile apps tends to be less than the defect fix times of large desktop/server applications, we explore two factors that may influence the defect fix time. First, we explore the number of defects reported in the issue tracking system. Second, we explore the distribution of defects across source code files.

Discussion

Number of Reported Defects

One factor that may influence the time it takes to fix defects is the number of defects reported in the issue tracking system. If few defects are reported in the issue tracking system, then fewer defects need to be fixed and developers can focus more of their attention on these defects.

We explore the number of defects reported by counting the number of issues in the issue tracking system that have been marked as defects. Similar to our analysis of defect fix times, we do not include issues that we classify as "not a bug," "duplicates" or "not reproducible." However, unlike our analysis of defect fix times, we do not limit our analysis to issues that have been closed. This is because we are including defects that have been reported, but have yet to be fixed.

We also explore the number of reporters who have reported at least one defect in the issue tracking system. This analysis is similar to our identification of unique developers, except that we use the list of people who have reported issues, instead of the list of people who have committed source code.

Table 3.6 presents the number of reported defects and the number of users reporting defects from the issue tracking system for each of the selected mobile app and desktop/server application. We find that few defects are reported and few users report defects in both mobile apps and Unix utilities compared to large desktop/server applications.

Android users primarily download apps through Google Play, where they can also rate apps and provide comments. However, user ratings and comments do not provide the same level of structure as issue tracking systems. The developers of M7 have specifically asked users to report defects in their issue tracking system.

Table 3.6: Number of Defects Reported and Unique Defect Reporters

ID	Reporters	Reported Defects
M1	4	21
M2	13	15
M3	267	342
M4	315	425
M5	8	20
M6	6	7
M7	1,293	2,518
M8	178	192
M9	99	120
M10	18	62
M11	6	7
M12	591	803
M13	77	100
M14	33	38
M15	15	98
Median	**33**	**98**
D1	3,425	5,104
D2	1,206	6,287
Median	**2316**	**5696**
D3	35	268
D4	26	64
D5	25	55
Median	**26**	**64**

From Table 3.6, we find that the greatest number of defects are reported in M7 (email client), M12 (VOIP client), M4 (SSH client) and M3 (barcode scanner).

Whereas, few defects are reported in mobile apps related to entertainment, M1 and M5 (multimedia players), M6, M11 and M14 (games) and M10 (social networking).

Distribution of Defects

Another factor that may influence the time it takes to fix defects is the distribution of defects across source code files. If defects tend to be concentrated in a few files and developers are aware of these files, then they may be able to locate these defects with less effort. Ostrand et al. found that, in large software systems, most defects are found within a small subset of the source code files (Ostrand et al., 2005). The authors found that 80% of the defects are found within 20% of the source code files (this is often referred to as the 80-20 rule). Hence, developers can prioritize their code reviews and test cases to focus on these files and reduce the effort required to locate most defects.

We explore the distribution of defects across source code files by extracting the number of defect fixing changes made to each source code file. We assume that each defect fixing change corresponds to a defect in the source code file. We find defect fixing changes by mining the commit log messages for a specific set of key words (Hassan, 2008*a*; Mockus and Votta, 2000). The keywords (i.e., "bug(s)", "fix(es,ed,ing)", "issue(s)", "defect(s)" and "patch(s)" are developed based on manual analysis of a large sample of commit log messages. We find the total number of defects in a source code file by counting the number of times the file is changed by the set of defect fixing changes.

We were unable to use the issues reported in the issue tracking system because these tend not to include information regarding how the defect was fixed (e.g., the

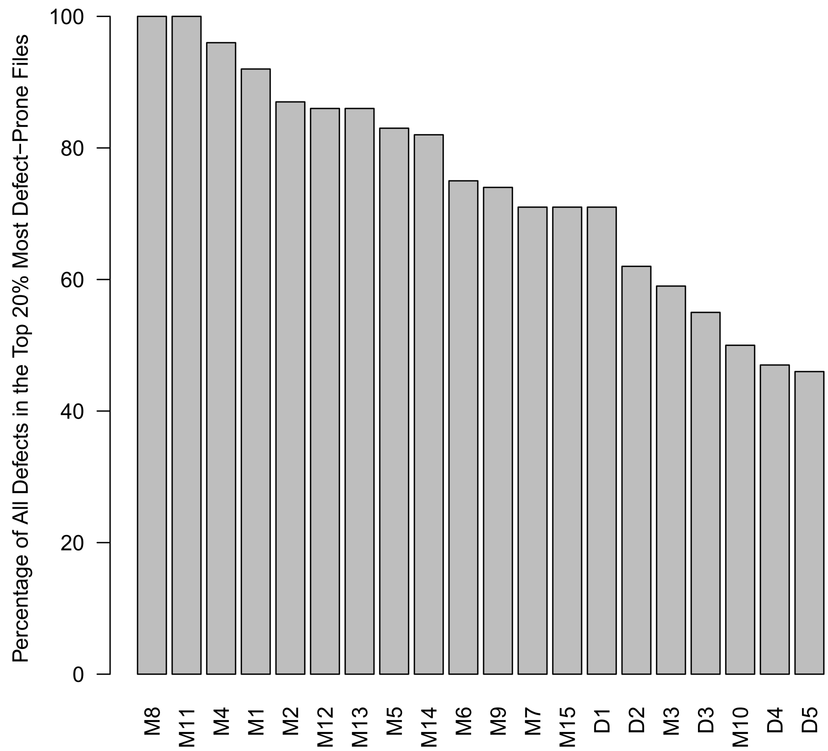


Figure 3.4: Percentage of All Defects in the Top 20% Most Defect-Prone Files.

location of the defect). We were also unable to trace defect fixing changes to specific issues in the issue tracking system because mobile app developers tend not to reference specific issues in their commit messages. Hence, heuristics based on the commit message need to be used to identify defect fixing changes despite the lack of a connection between the source code repository and issue tracking system.

We extract the number of defects in each source code file. We then calculate the number of defects in the top 20% most defect-prone files by sorting the list of files by the number of defects in descending order and summing the number of defects in the first 20% of the list. Figure 3.4 presents the percentage of defects in the top 20% most defect-prone files for each mobile app and desktop/server application.

We find that the concentrations of defects in a few defect-prone files is the highest across the selected mobile apps, followed by large desktop/server applications and finally Unix utilities.

In our first research question, we found that the size of the selected mobile apps is small. Combined with the distribution of defects across source code files, mobile app developers can find the majority of defects in a very small number of files, typically in the 10s of files.

From Figure 3.4, we find that a higher percentage of defects are concentrated in the top 20% most defect-prone files in mobile apps, compared to desktop/server applications. At least 80% of the defects are concentrated in the top 20% most defect-prone files in nine of our mobile apps and at least 70% of the defects are concentrated in the top 20% most defect-prone files in thirteen of our mobile apps.

Finding the rationale for the relatively quick defect fix times in the selected mobile apps requires more analysis on other systems; however, our current findings suggest the following:

One potential factor that may have led to relatively quick defect fix times may be the fact that fewer defects are being reported by users. However, the number of people reporting defects appears to be related to the application type. Another potential factor that may have led to relatively quick defect fix times may be the distribution of defects across source code files. Defects tend to be concentrated in a few files, therefore, if developers are aware of these files, then they may be able to locate these defects with less effort.

We find that mobile app developers tend to fix defects in less time than desktop/server application developers, regardless of the size of the project. In addition, fewer defects tend to be reported in mobile apps and Unix utilities. Further, defects in mobile apps tend to be concentrated in a smaller portion of the source code files.

3.3 Threats to Validity

3.3.1 Threats to Internal Validity

Threats to interval validity describe concerns regarding alternate explanations for our results.

Wget was first released in January 1996 and Aspell was first released in September 1998; however, the development history of this time is not available. Therefore, the first three years of the publicly available source code repository and issue tracker do not correspond to the first three years of development.

The number of downloads is not an ideal measure of success (unlike user retention or engagement) (Localytics, 2011), however, it is the best measure currently available. In addition, we only have the number of downloads during a single month. Without a longer term trend, it is unclear whether this number is an anomaly (i.e., a month with an unusually low or high number of downloads) or whether it represents an increase or decrease over the previous months. Therefore, it is possible that we have mistakenly included or excluded mobile apps from our analysis.

3.3.2 Threats to Construct Validity

Threats to construct validity describe concerns regarding the measurement of our metrics.

The number of unique developers was found by counting the number of unique local parts (i.e., the characters before the @ symbol) for each developer who made at least one commit to a source code file in the repository. Similarly, the number of unique reporters was found by counting the number of unique local parts for each reporter who submitted at least one defect report. While we did perform a manual verification of this analysis, it is possible that we misidentified two local parts as either unique or distinct. For example, john_doe@gmail.com and admin@my_project.com were counted as two distinct developers/reporters, although they may be a single developer/reporter with multiple (distinct) email address. Conversely, j_doe@gmail.com and j_doe@yahoo.com were counted as one distinct developer/reporter, although they may be two distinct developers/reporters (e.g., John Doe and Jane Doe). Furthermore, the number of unique developers is based on the list of people who commit to the source code repository and the number of unique reporters is based on the list of people who submit issues to the issue tracking system. This does not capture people who submit code or issues using other mediums (e.g., email or forums).

The time to fix a defect was calculated by counting the number of days between the date an issue describing a unique defect was opened and the date it was closed. It is possible that the defect was fixed within one week, but the issue tracking system was not immediately updated to reflect the fix. In addition, this analysis does not take into account defects that were not reported within the issue tracking system. This may be particularly true in mobile apps with only one or two developers.

The number of defects in each source code file was measured by identifying the source code files that were changed in a defect fixing change. Although this technique has been found to be effective (Hassan, 2008*a*; Mockus and Votta, 2000), it is not without flaws. We identified defect fixing changes by mining the commit logs for a set of keywords. Therefore, we are unable to identify defect fixing changes (and therefore defects) if we failed to search for a specific keyword, if the committer misspelled the keyword or if the committer failed to include any commit message. We are also unable to determine which source code files have defects when defect fixing modifications and non-defect fixing modifications are made in the same commit. However, such problems are common when mining software repositories (Hassan, 2008*b*).

3.3.3 Threats to External Validity

Threats to external validity describe concerns regarding the generalizability our results.

The studied mobile apps and desktop/server applications represent a small subset of the total number of mobile apps and desktop/server applications available. We have limited our study to open source mobile apps and desktop/server applications. In addition, we have only studied the mobile apps of a single mobile platform (i.e., the Android Platform). Therefore, it is unclear how our results will generalize to 1) closed source mobile apps and desktop/server applications and 2) other mobile platforms.

3.4 Conclusions

This chapter presents an exploratory study to compare mobile apps and desktop/server applications, as a first step toward understanding how the software engineering empirical theories developed by studying desktop/server will hold in mobile apps. We study fifteen open source Android apps and five desktop/server applications (two large, commonly studied systems and three smaller Unix utilities).

We find that, in some respects, mobile apps are similar to Unix utilities and differ from large desktop/server applications. Mobile apps and Unix utilities are smaller than traditionally studied desktop/server applications, e.g., the Apache HTTP server and Eclipse compiler. This is true in terms of the size of the code base (the number of source code files and lines of code) and the development team. Furthermore, we find that the number of core developers, i.e., those responsible for at least 80% of the commits, is very small in both mobile apps and Unix utilities, typically only one or two. We also find that few users report defects and few defects are reported in both mobile apps and Unix utilities.

We also find that, in other respects, mobile apps differ from both Unix utilities and large desktop/server applications. We find that mobile app developers place a great deal of emphasis on rapidly responding to quality issues and most projects fix over a third of reported defects within one week and two thirds within one month. This is greater for the Eclipse UI component and the aspell, joe and wget utilities, which typically fix only 20% of the defects in one week and 40% of the defects in one month. However, developers of the Apache HTTP server project fix 46% of all reported defects in one week and 96% of all reported defects in one month. We also find that the concentrations of defects in a few defect-prone files is the highest in mobile apps, followed by large desktop/server applications and finally Unix utilities. Most mobile apps have more than 80% of the defects in 20% of the most defect-prone files. This compares to two third and half for large desktop/server applications and Unix utilities respectively.

Finally, we observe that many mobile apps depend highly on their underlying platform (i.e., the Android platform). A lower dependence on the platform indicates that developers do not rely significantly on the platform APIs. For example, their app may be simple or self-contained, or the platform may be too difficult to use. Such mobile apps may be easily ported to other platforms. Conversely, a higher dependence on the platform indicates that mobile app developers heavily exploit platform APIs. However, this leads to platform "lock-in", which may complicate porting to other platforms and potentially introduces instability due to the rapid evolution of mobile platforms. While these issues are relevant to all software that is built on an underlying platform or framework, it is particularly acute in mobile apps. This is because the Android platform averages one major release every year. Hence, researchers, should look at the impact of platform dependence on quality and how backward compatibility issues could affect quality. We will study the relationship between platform dependence and quality in Chapter 5.

Although we found that differences do exist between mobile apps and traditionally-studied desktop/server applications, the underlying cause(s) of these differences are unclear. However, the purpose of this study was to establish that such differences do exist and to motivate a reexamination of software engineering empirical theories in the context of mobile apps.

In particular, we reexamine three software engineering empirical theories and found that:

1. *Core developers* – a small subset of the developers are responsible for the majority of the development and maintenance of mobile apps, Unix utilities and desktop/server applications. However, the number of core developers in both mobile apps and Unix utilities is much smaller than the number of core developers in desktop/server applications.

2. *Defect Reporters* – the number of people reporting defects is typically an order of magnitude larger than the number of developers of mobile apps, Unix utilities and desktop/server applications. However, the number of people reporting defects in both mobile apps and Unix utilities is much smaller than the number of people reporting defects in desktop/server applications.

3. *Defect distribution* – the majority of defects occur in a small subset of the source code files mobile apps, Unix utilities and desktop/server applications. However, defects tend to be concentrated in a smaller portion of the source code files of mobile apps.

Future studies could include browser plugins and desktop widgets. Similar to mobile apps, these applications are often developed by small teams and are dependent on an underlying platform. However, we believe that our key findings hold. Therefore, many software engineering empirical theories should be reexamined in the context of mobile apps and other small applications.

Our findings suggest that mobile app developers may be faced with unique challenges. For example, many mobile apps have a very high frequency of releases (e.g., K9Mail typically has two internal releases every week and one release to Google Play every month). These quick release cycles may be required to remain competitive within the marketplace.

We observe that some mobile apps do not follow a formal development or maintenance process. These apps are developed in an ad hoc manner to get to the marketplace as quickly as possible. For example, as we discussed in Subsection 5.1.1, three mobile apps were excluded from our case study because they did not have a public issue tracking system. These apps had been downloaded hundreds of thousands of times, and yet they did not have a system in place where users could report defects. In addition, the source code repositories of eleven mobile apps in our case study do not contain any test cases. Such ad hoc development and maintenance processes may adversely affect the quality or maintainability of mobile apps. Researchers should also investigate the relationship between these two factors (frequent releases and lack of formal testing) and the quality of code. Khomh et al. have studied the Mozilla Firefox project and found that a shorter release cycle 1) allows defects to be fixed faster and 2) does not introduce significantly more defects (Khomh et al., 2012). However, several open questions remain. Does such a high frequency of releases mitigate the lack of testing? If new releases can be pushed with ease, then does the quality of a particular release matter as much? Is the project in a constant beta testing state? Does the platform provide the sufficient support for building high quality apps quickly? Is the frequent release only influenced by the demand factor in the app store? Are the developers of mobile apps more skilled or do they have more resources at hand? Or, are mobile apps themselves less complex to develop?

Another challenge that mobile app developers may face is project continuity. Many mobile apps have very small development teams, often with only one or two core developers. In large scale systems, stringent code ownership policies are typically associated with high quality software. However, stringent code ownership policies in mobile apps lead to a continuity issue. In mobile apps, knowledge is typically concentrated with a small number (one or two) developers who become critical to the future development and maintenance of the project. If these developers withdraw from the project, a significant portion of the collective knowledge becomes unavailable and the future of the project may be called into question. This is compounded by the ad hoc nature of mobile app development discussed previously. However, as we have seen in the smaller Unix utilities, software projects with small teams have been able to mitigate the issue of continuity. Hence, researchers should explore the development practices of these utilities, in addition to large desktop software, to understand the relationship between code ownership and project continuity.

Our findings suggest that mobile apps may be faced with unique challenges and, in order to support the thousands of mobile app developers, researchers should begin to study mobile apps alongside traditionally studied desktop/server applications.

In the following chapters, we delve deeper into mobile app development practices of two different mobile platforms (Chapter 4) and the relationship between platform dependency and mobile app quality Chapter 5).

Chapter 4

Exploring the Development of Mobile Apps

In the previous chapter we performed an exploratory study to compare fifteen open-source mobile apps and five open-source desktop/server applications. We found that mobile apps differ from traditionally studied desktop/server applications in many ways. However, our study was restricted to Android apps, the largest (by user base) and fastest growing mobile platform. In this chapter we study the difference between the mobile apps of two different platforms.

Although the number of mobile apps has seen an explosive growth in the past few years (Butler, 2011; ComScore Inc., 2010; International Data Corp., 2011), the number of mobile apps available on each of the popular mobile platforms (Apple's iPhone, Google's Android and Research In Motion's BlackBerry) is not equal.

The number of mobile apps available on each platform is affected by many factors, including marketing, public perception and the overall development experience (Murai, 2011; Nielsen Co., 2011*b*). Notwithstanding the considerable differences presented in Table 1.1, four of the top five most popular mobile apps on each platform are actually the same. This can be seen in Table 4.1, which shows the most popular mobile apps by platform based on a survey of 4,000 mobile app users in August 2010 (Nielsen Co., 2010*a*). Further, a 2012 study by the independent research firm Distimo found that 33% of mobile apps are available for multiple platforms, including many of the most popular apps (Distimo, 2012). Therefore, in order to reach the largest consumer base, mobile app developers need to develop for each of these mobile platforms.

Table 4.1: Most Popular mobile apps By Platform (Nielsen Co., 2010*a*).

iPhone	Android	BlackBerry
1. Facebook	1. Google Maps	1. Facebook
2. Weather Channel	2. Facebook	2. Weather Channel
3. Google Maps	3. Weather Channel	3. Google Maps
4. iPod/iTunes	4. Pandora	4. Pandora
5. Pandora	5. YouTube	5. Twitter

However, companies have a hard time porting and maintaining their mobile apps on multiple mobile platforms. First, the explosion of new mobile devices, operating systems and frameworks has resulted in a highly fragmented market (Gasimov et al., 2010). Developers need to make their code aware of different features and quirks of the supported devices, and update their mobile app for every new major device or new version of the operating system. Second, development tools have been released freely to the general public so that anyone can develop a mobile app, even without prior development experience (Butler, 2011). However, to our knowledge there have been no detailed studies on the mobile app development or maintenance processes. A good understanding of these processes is necessary to grasp the speed and scale of mobile app development as well as the need for mechanisms to defend against platform changes and maintain backwards comparability.

The purpose of this study is to further explore the new world of mobile apps by comparing the mobile apps of different mobile platforms. Mobile apps are expected to be one of the major challenges for software maintenance and program understanding in the near future. This software, and the hardware it relies on, is constantly and rapidly evolving. When mobile apps first rose to prominence in 2008 there were very few mobile platforms. Now, three years later, there are several major mobile platforms, each of which have taken turns as the most popular one (Nielsen Co., 2010*c*, 2011*a*,*b*). Developers need to target multiple platforms in response to shifting consumer preferences. However, amidst market pressure and with limited resources and experience, how companies do (and should do) this is an open research question.

As a first step towards addressing this question, we perform an exploratory study to compare the differences between mobile apps for different platforms: are there clear differences in code characteristics, dependencies and churn? We study mobile apps that have feature-equivalent versions for different platforms. In total, we selected three pairs of mobile apps from two platforms. Our study addresses the following three research questions:

- *RQ1: How different are the code characteristics between platforms?* – We find that less code is required to implement a feature on the Android platform. BlackBerry mobile apps include and customize more third party source code.

- *RQ2: How different are the number and type of dependencies between platforms?* – We find that Android mobile apps rely much more on the underlying platform than BlackBerry mobile apps do. Over 50% of the dependencies on the BlackBerry platform can be attributed to user interface APIs. Mobile apps written for either the Android or BlackBerry platforms rely on the Java APIs for at least one third of their dependencies.

- *RQ3: How different is the amount of code churn between platforms?* – We find that the maintained third party library code changes very little. Code churn is very high on both platforms.

This chapter is organized as follows: Section 4.1 describes the setup of our case study and Section 4.2 describes and discusses the results of our case study. Section 4.3 outlines the threats to validity. Finally, Section 4.4 concludes the chapter.

4.1 Case Study Setup

This section outlines our approach to explore the development of mobile apps. First, we select mobile apps that have feature-equivalent versions for the Android and BlackBerry platforms. Second, we measure the source code, dependency and churn properties of theses mobile apps. We then compare the measurements across the subject mobile apps.

4.1.1 Mobile App Selection

We select mobile app pairs for our case study based on the following criteria.

- Open-source – mobile apps must be open source, as we require access to the source code repository.

- Feature equivalence – we require mobile app pairs that have feature-equivalent versions that run on different platforms. This requirement allows us to directly compare the effort it takes to implement equivalent functionality on the two platforms, possibly written by different developers (or companies).

- Programming language equality – we require that mobile app pairs are developed in the same programming language. This requirement simplifies our case study, since it is hard to compare the source code characteristics of a mobile app written in Objective-C for the Apple iPhone platform versus a mobile app written in Java for the Google Android or RIM BlackBerry platforms.

We select the Android and BlackBerry platforms as the focus of our study, because 1) they are two of the most popular mobile platforms and 2) mobile apps for these platforms are written mostly in Java (Butler, 2011; Nielsen Co., 2011*a*,*b*).

We select three mobile apps (i.e., WordPress, Google Authenticator and Facebook SDK) for our case study. To ensure that the mobile app pairs are feature equivalent, we verify feature differences using feature lists on the mobile app webpages, change logs for each release and feature requests in the forum or issue tracking systems.

WordPress is one of the most popular content management systems in use today. The WordPress mobile app is open-source, available on the Android and BlackBerry platforms and the features of both versions are nearly identical (W3Techs - Web Technology Surveys, 2012; WordPress for Android, 2012; WordPress for BlackBerry, 2012). The WordPress mobile apps allow users to manage their blog or web page from their mobile device. Source code for the WordPress for Android mobile app was first committed to the repository in September 2009, while code for the BlackBerry mobile app was first committed to the repository in April 2009.

Google Authenticator is a mobile app that allows users to generate 2-step verification codes on their mobile devices without an Internet connection. This adds an extra layer of security to a user's Google Account (e.g., Gmail) by requiring the user to have access to his/her phone (in addition to the typical username and password) (Google Authenticator, 2012). Both versions of the Google Authenticator mobile apps are developed by Google. Hence, developers for both the Android and BlackBerry versions of the Google Authenticator mobile app share the same source code repository and bug database. Source code for both versions of the Google Authenticator mobile app was first committed to the repository in March 2010.

The Facebook SDK is an open source project that allows developers to integrate Facebook's functionality into their own mobile apps (Facebook SDK for Android, 2012; Facebook SDK for BlackBerry, 2012). The Facebook SDK for BlackBerry was developed by Research in Motion, whereas the Facebook SDK for Android was developed by Facebook. Source code for the Facebook SDK for Android was first committed to the repository in May 2010. Source code for the Facebook SDK for BlackBerry was first committed to the repository in July 2010.

The source code for each version of these mobile apps is available in the repositories listed in Table 4.2. We perform our analysis on the source code in the repository up to, and including, the last commit which was tagged as a release. The specific release of each mobile apps is listed in Table 4.2.

4.1.2 Source Code Properties

We use the Understand tool by SciTools (Scitools, 2012) to extract the metrics in Table 4.3 for each mobile app. Understand is a static analysis toolset for measuring and analyzing the source code of small- to large-scale software projects written in a number of programming languages.

Table 4.2: mobile app Repositories.

WordPress		
Android	android.svn.wordpress.org	1.4.0
BlackBerry	blackberry.svn.wordpress.org	1.4.6.2
Google Authenticator		
Android	google-authenticator.googlecode.com/hg	0.54
BlackBerry	google-authenticator.googlecode.com/hg	1.1.2
Facebook SDK		
Android	github.com/facebook/facebook-android-sdk.git	1.5.0
BlackBerry	facebook-bb-sdk.svn.sourceforge.net	0.4.5

Table 4.3: Source Code Volume Metrics.

Metric	Definition
Files	Total Number of Files Containing Source Code
Classes	Total Number of Classes
Lines Code	Total Number of Lines of Code

We measure the source code volume metrics for the entire mobile app, and for two subsets of the mobile app base: mobile app specific source code and third party library source code.

Third party libraries consist of reusable software components developed and maintained by developers unaffiliated with the mobile app. For example, CWAC (CommonsWare Android Components) is a collection of open source libraries specifically developed to help Android mobile app developers tackle common and recurring issues (CommonsWare Android Components, 2012). Mobile apps often include, customize and maintain the source code of third party libraries, therefore it is important to study the project-specific source code metrics and the maintained third party library source code independently.

In order to identify third party libraries, we examine the projects' directory structure looking for utility directories or directories commonly associated with third party libraries (e.g., src/com/ on the BlackBerry platform). In all six mobile apps, the third party libraries are included as `.java` files. Therefore, we are able to examine each source code file for license agreements, disclaimers, documentation or links to other projects in the source code comments.

After we classify each source code file as either third party or project-specific, we compare the source code volume metrics for both groups of files to determine how much of the mobile app the developers have to develop and maintain themselves (i.e., everything other than the third party code).

4.1.3 Dependency Properties

Similar to desktop and web applications, mobile apps make use of APIs that provide access to functionality that the developers would otherwise have to implement themselves. Three types of APIs are provided to developers: the Java API, the platform API (i.e., Android- or BlackBerry-specific APIs) and third party libraries. Android developers have access to nearly all of the Java 2 Standard Edition APIs, whereas BlackBerry developers have access to the Java 2 Micro Edition.

We use the Understand tool introduced in Section 4.1.2 to extract, for each class in the mobile app, a list of classes on which the class depends. We classify these dependencies into one of the following categories based on the class name:

- **Language dependency** - dependency on a class that is part of the Java platform (e.g., `java.io.IOException` or `java.lang.Thread`).
- **User Interface dependency** - dependency on a class that is part of the device platform and that is responsible for the user interface (e.g., `android.view` or `net.rim.device.api.ui`).
- **Platform dependency** - dependency on a class that is part of the device platform and not responsible for the user interface (e.g., `android.app.Activity` or `net.rim.device.api.system.EventLogger`).
- **Third Party dependency** - dependency on a class that is part of a third party library.
- **Project dependency** - dependency on some class in the mobile app code base other than a third party class.

Listing 4.1: Hello, World - An Android Developer's First mobile app (Android Developers, 2012).

```
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloAndroid extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    TextView tv = new TextView(this);
    tv.setText("Hello, Android");
    setContentView(tv);
  }
}
```

Understand extracts and counts the following types of class dependencies:

- Calls - call to a method in another class
- Casts - cast to an object type defined in another class
- Creates - creation of an object whose type is defined in another class
- Extends - extending another class
- Implements - implementing an interface
- Sets - setting a variable or object defined in another class
- Typeds - use of an object type defined in another class
- Uses - use of a variable or object defined in another class

As an example of this analysis, consider the "Hello, World" code for the Android in Listing 4.1 (Android Developers, 2012). From this example, Understand extracts the class dependencies in Table 4.4.

Table 4.4: Listing 4.1 Class Dependencies

Dependency	Cause	Line
android.app.Activity	HelloAndroid **Extends** Activity	5
android.os.Bundle	savedInstanceState **Typeds** Bundle	6
android.widget.TextView	tv **Typeds** TextView	8
	HelloAndroid.onCreate **Creates** TextView	8
	HelloAndroid.onCreate **Calls** TextView	8
	HelloAndroid.onCreate **Calls** setText	9

In this study, we count the total and unique number of dependencies on each dependency category. For example, we can summarize the dependency information of Listing 4.1 as in Table 4.5. Such a class dependency summary is a measure of how strongly a mobile app is tied to Java, the underlying platform, the UI, third party libraries or itself. For example `android.widget.TextView` is the only dependency in Table 4.4 that is an Android UI API (`android.widget.TextView` is responsible for providing the functionality for text boxes). From Table 4.4, there are four total User Interface dependencies (i.e., four dependencies on `android.widget.TextView`) and one unique User Interface dependency (i.e., `android.widget.TextView`).

Table 4.5: Listing 4.1 Class Dependency Summary

Dependency Class	Number of Dependencies	
	Total	Unique
Language	0	0
User Interface	3	1
Platform	2	2
Third Party	0	0
Project	0	0

We define the "platform dependency ratio" (PDR) as the ratio between the number of platform and user interface dependencies, and the total number of dependencies.

$$PDR = \frac{\text{User Interface} + \text{Platform}}{\text{Language} + \text{User Interface} + \text{Platform} + \text{Third Party} + \text{Project}} \qquad (4.1)$$

A low platform dependency ratio indicates that developers do not rely significantly on the platform APIs. For example, their mobile app may be simple or self-contained, or the platform may be too difficult to use. Such mobile apps can be easily ported to other platforms. Conversely, a high platform dependency ratio indicates that mobile app developers heavily exploit platform APIs. However, this leads to platform "lock-in", which complicates porting to other platforms and potentially introduces instability due to the rapid evolution of mobile device platforms. For example, Listing 4.1 has a platform dependency ratio of 100%, i.e. it is highly tied to the Android platform and might need to be rewritten completely to port it to another platform.

The effort required to port a mobile app from one platform to another depends on a number of factors (e.g., the number and complexity of features and the platform dependency ratio). In our specific case study, we study three pairs of feature equivalent mobile apps. Therefore, we expect that the platform dependency ratio is a good measure of platform lock-in.

4.1.4 Code Churn Properties

Source code is constantly changing throughout the development process in response to maintenance and evolution activities. Code churn measures how much source code changes over time. We measure code churn using the following metrics:

- Number of files changed per change set.
- Number of project specific files changed per change set.
- Number of third-party files changed per change set.
- Number of lines changed per change set.
- Number of project-specific lines changed per change set.
- Number of third party lines changed per change set.

For our metrics, we ignore the initial commit, since this would heavily skew the churn metrics, as well as change sets that did not change any Java source code file. In addition, we measure the number of non-white space lines of code. We did not include changes to comment lines, since these are hard to detect from the change set diffs. We calculate our line churn metrics as the sum of all the added and deleted lines, for each file, for each commit.

Given the wide variety of source code repository formats, we use the following tools/commands to extract the number of changes, changed files and changes lines from the following repository formats:

- Subversion (svn) - We use statsvn (StatSVN, 2012) to extract the total number of changes and total number of changed lines for each file and directory in a project. Statsvn is an open source tool for generating project development metrics that characterize developer activity, project growth and code churn.
- GIT - we use `git log --numstat`, a standard git command (git-log, 2012), on the entire repository to extract the number of lines added and deleted from each file during each commit.

- Mercurial (hg) - we use `hg churn -f '\%s'`, a standard mercurial extension (ChurnExtension, 2012), on each file in the repository to extract the number of lines modified in each file in each commit.

4.2 Case Study Results

This section presents the results of our case study on the three pairs of mobile apps selected in Section 4.1.1.

RQ1: How different are the code characteristics between platforms?

Motivation

Source code volume metrics have been shown to be highly correlated to the complexity of a software system (Herraiz et al., 2007; Lind and Vairavan, 1989). The complexity of a software system measures the difficulty of understanding, evolving and maintaining the system. We measure and compare the source code volume metrics for each pair of mobile apps to determine if differences between the Android and BlackBerry platforms require developers on either platform to write more complex source code when implementing similar functionality.

We also measure and compare the source code volume metrics for the mobile app and third party libraries. Reuse of third party libraries has many possible benefits including reduced development time and increased quality. However, third party software has potential drawbacks, as mobile app developers should ensure their copy of the third party API is always in sync and up to date with the most recent updates and bug fixes, although they may not be familiar with the third party library code. In addition, mobile app developers can be negatively impacted if support for the third party library or parts of its functionalities are is abandoned.

Approach

We use the methodology presented in Section 4.1.2 to extract source code volume metrics from each mobile app. Table 4.6 presents the results of these metrics and the percentage of increase of each metric for the BlackBerry version relative to the Android version for each mobile app pair. After identifying which source code files belong to third party code, and which files contain actual mobile app code, we extract the source code volume metrics of both groups of files. Table 4.8 presents the values of these metrics and the percentage (in parentheses) of the total code base for each category of each mobile app pair.

To better understand the distribution of project-specific file sizes, we visualize the file size, in terms of the number of lines of code, using bean plots. Bean plots are an alternative to box plots to summarize and compare the distribution of different sets of data (Kampstra, 2008). The x-axis of a bean plot shows the estimated density of the distribution and varies between 0 to 1. Figure 4.1, 4.2 and 4.3 also show the median file size of each mobile app (solid black line).

Results

Table 4.6 shows that a mobile app written for the BlackBerry platform contains two (+125%) to more than six (+553%) times as many lines of code as the equivalent Android mobile app. The differences in number of files are even larger (+850%).

Table 4.6: Global Source Code Volume Metrics and Difference Relative to Android.

WordPress			
Metric	Android	BlackBerry	Difference
# Files	55	241	+338%
# Classes	360	575	+60%
# Lines Code	15,928	35,775	+125%
Google Authenticator			
Metric	Android	BlackBerry	Difference
# Files	10	31	+210%
# Classes	26	61	+135%
# Lines Code	1,344	3,322	+147%
Facebook SDK			
Metric	Android	BlackBerry	Difference
# Files	6	57	+850%
# Classes	12	77	+542%
# Lines Code	777	5,070	+553%

Table 4.7: Breakdown of Source Code Volume Metrics Across Project-Specific Code.

WordPress			
Metric	Android	BlackBerry	Difference
# Files	40 (73%)	211 (88%)	528%
# Classes	331 (92%)	543 (94%)	164%
# Lines of Code	12,948 (81%)	30,764 (86%)	237%
Google Authenticator			
Metric	Android	BlackBerry	Difference
# Files	10 (100%)	17 (55%)	170%
# Classes	26 (100%)	47 (77%)	181%
# Lines of Code	1,344 (100%)	1,421 (43%)	106%
Facebook SDK			
Metric	Android	BlackBerry	Difference
# Files	6 (100%)	21 (37%)	350%
# Classes	12 (100%)	33 (43%)	275%
# Lines of Code	777 (100%)	1,723 (34%)	222%

Table 4.8: Breakdown of Source Code Volume Metrics Across Third Party Code.

WordPress			
Metric	Android	BlackBerry	Difference
# Files	15 (27%)	30 (12%)	200%
# Classes	29 (8%)	32 (6%)	110%
# Lines of Code	2,980 (19%)	5,011 (14%)	168%
Google Authenticator			
Metric	Android	BlackBerry	Difference
# Files	0 (0%)	14 (45%)	—
# Classes	0 (0%)	14 (23%)	—
# Lines of Code	0 (0%)	1,901 (57%)	—
Facebook SDK			
Metric	Android	BlackBerry	Difference
# Files	0 (0%)	36 (63%)	—
# Classes	0 (0%)	44 (57%)	—
# Lines of Code	0 (0%)	3,347 (66%)	—

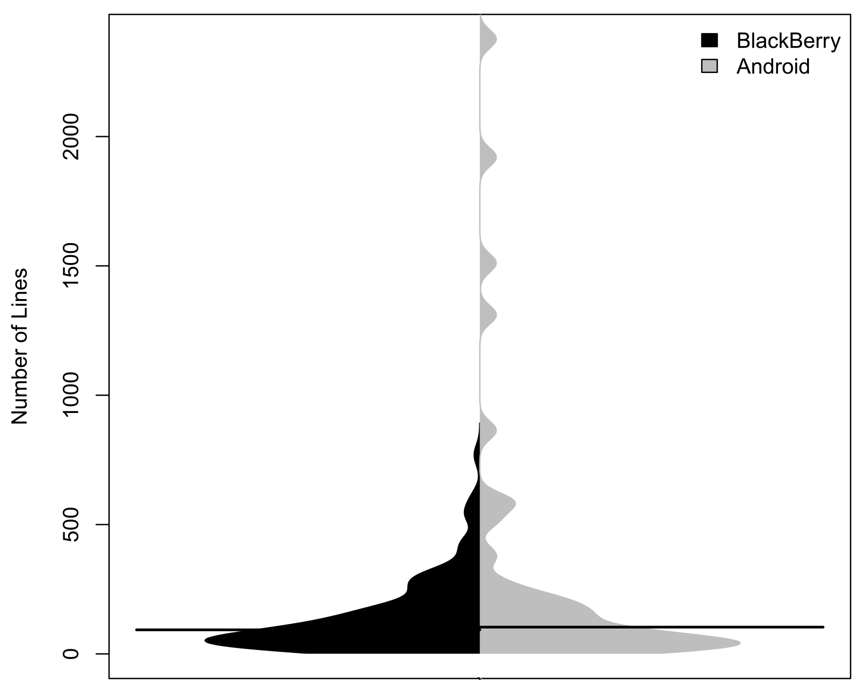


Figure 4.1: Distribution of file sizes across the WordPress mobile app project-specific files.

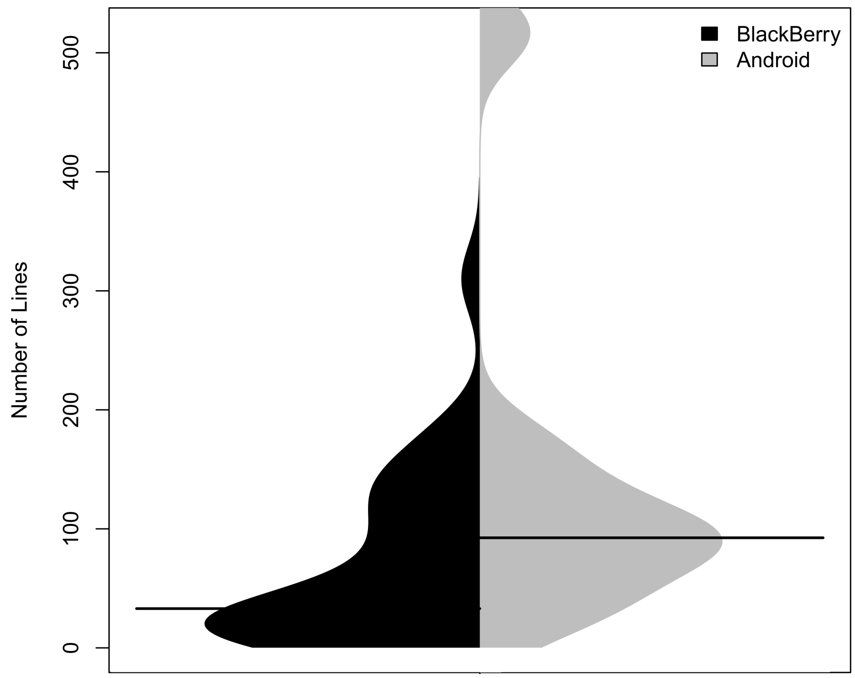


Figure 4.2: Distribution of file sizes across the Google Authenticator mobile app project-specific files.

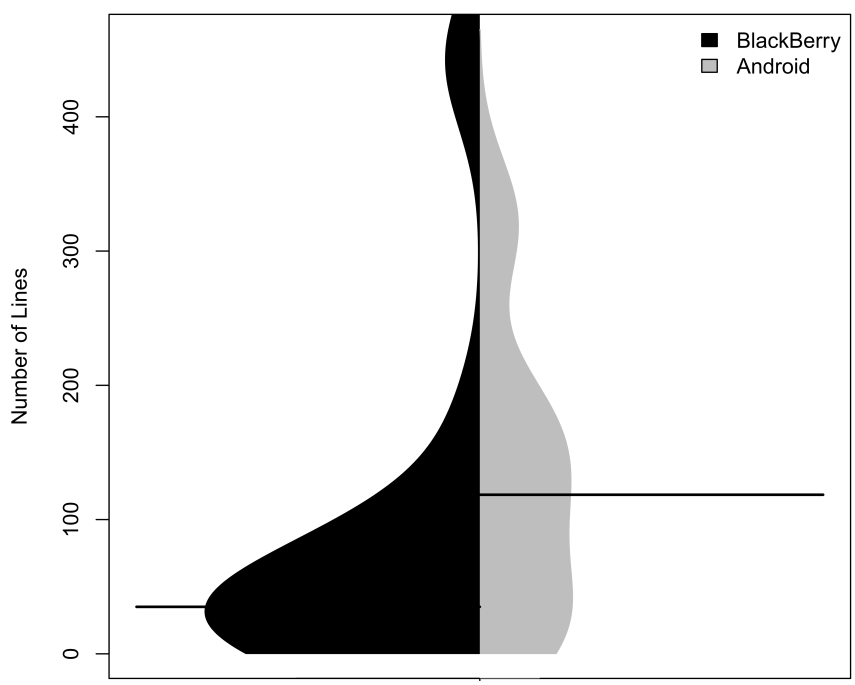


Figure 4.3: Distribution of file sizes across the Facebook SDK project-specific files.

This difference is not merely due to differences in coding style of the developers developing the Android and BlackBerry apps, since the Google Authenticator mobile apps (developed by the same company) also show these differences. If source code volume is a good indicator of development effort, more effort seems to be needed on the BlackBerry platform. However, since most BlackBerry mobile apps contain both their own source code, as well as the source code for third party libraries, it is necessary to break down the volume metrics across mobile app and third party code (Table 4.8).

Table 4.8 shows that in two of the three mobile app pairs the BlackBerry mobile apps contain more lines of code in third party libraries than in project-specific source code (57% in Google Authenticator and 66% in the Facebook SDK). On the other hand, third party libraries are either not included (Facebook SDK and Google Authenticator) or make up approximately 19% of the lines of code (WordPress) of Android mobile apps. When looking only at the project-specific code, these BlackBerry mobile apps are still up to two times as large as the corresponding Android mobile app.

From Figure 4.1, 4.2 and 4.3, Android mobile app developers tend to write much larger files (with respect to lines of code) than BlackBerry mobile app developers. First, from Figure 4.1 and Figure 4.2, Android mobile apps have more outliers, i.e., more files that are significantly larger than the median file size. This is particularly true in the WordPress for Android mobile app, where a significant amount of WordPress code is concentrated in a few files. Second, from Figure 4.2 and Figure 4.3, the median file size of an Android mobile app is more than twice the median file size of the feature-equivalent BlackBerry mobile app.

Note, however, that the area of bean plots is standardized to 1 (i.e., the bean plots cannot be used to compare the total code size of the mobile apps in each pair).

> Less source code is required for Android mobile apps than feature equivalent BlackBerry mobile apps. Android mobile app developers typically write larger source code files and tend to concentrate more code into fewer files. BlackBerry mobile apps include more third party libraries in their code base.

RQ2: How different are the number and type of dependencies between platforms?

Motivation

We study the API usage properties of each mobile app to uncover how mobile apps depend on language, platform, user interface and third party APIs, and on their own classes. Since mobile app developers need to port their mobile apps to multiple platforms, a high platform dependency ratio (defined in Section 4.1.3) negatively impacts the porting process by increasing the amount of code that needs to be rewritten).

Approach

We use the methodology presented in Section 4.1.3 to extract the number of dependencies for each class in the mobile app. Table 4.9 presents our measurements for these key metrics and the percentage of the total number of dependencies (in parentheses) for each category of dependencies and each mobile app pair. Table 4.9 also presents the percentage of increase in the metrics for the BlackBerry version relative to the Android version of each mobile app. Finally, Table 4.10 presents the platform dependency ratio, defined in Equation 4.1 in Section 4.1.3, for each mobile app.

Table 4.9: Source Code Dependency Metrics.

WordPress			
Dependency	Android	BlackBerry	Difference
Language	5860 (43%)	6720 (33%)	+15%
Platform	3593 (27%)	600 (3%)	-83%
User Interface	1961 (15%)	2473 (12%)	+26%
Third Party	365 (3%)	665 (3%)	+82%
Project Specific	1724 (13%)	10132 (49%)	+488%

Google Authenticator			
Dependency	Android	BlackBerry	Difference
Language	312 (33%)	949 (58%)	+204%
Platform	269 (28%)	43 (3%)	-84%
User Interface	223 (23%)	154 (9%)	-31%
Third Party	0 (0%)	192 (12%)	—
Project Specific	156 (16%)	300 (18%)	+92%

Facebook SDK			
Dependency	Android	BlackBerry	Difference
Language	252 (42%)	1176 (38%)	+367%
Platform	170 (29%)	79 (3%)	-54%
User Interface	31 (5%)	99 (3%)	+219%
Third Party	0 (0%)	1101 (36%)	—
Project Specific	140 (24%)	606 (20%)	+333%

Results

Table 4.9 and Table 4.10 expose several interesting trends. (1) Android mobile apps rely much more on platform and user interface APIs than their BlackBerry equivalents (platform dependency ratios of 41%, 51% and 34% on the Android platform

Table 4.10: Mobile app Platform Dependency Ratios.

mobile app	Android	BlackBerry
WordPress	41%	15%
Google Authenticator	51%	12%
Facebook SDK	34%	6%

compared to 15%, 12% and 6% on the BlackBerry platform). (2) BlackBerry mobile apps depend heavily on project-specific classes, far more than they rely on the BlackBerry platform (49% of WordPress dependencies, 18% of Google Authenticator dependencies and 20% of Facebook SDK dependencies compared to 3% non-User Interface dependencies). (3) BlackBerry mobile apps rely on the underlying platform primarily for the user interface libraries. In the WordPress and Google Authenticator BlackBerry mobile apps, 80% of the platform dependencies are on user interface libraries. (4) For Android, the Android and Java APIs appear to provide most of the dependencies of the mobile apps. Even excluding the user interface APIs, over 25% of the Android mobile app dependencies are on the Android platform, leading to a relatively high platform dependency ratio (41% for WordPress, 51% for Google Authenticator and 34% for Facebook SDK). (5) Finally, in all three Android mobile apps, third party dependencies account for fewer dependencies than any other dependency category.

Results

From Table 4.9, it seems that the extent to which each mobile app depends on third party libraries seems to fluctuate from 0% in the Google Authenticator mobile app and Facebook SDK for Android to 36% in the Facebook SDK for BlackBerry. Manual analysis shows that these third party libraries are especially used for implementing functionality that is missing from a language or platform API. Examples of missing functionality on the BlackBerry platform are the Java Script Object Notation protocol and regular expression support, and on both platforms the module for XML-Remote Procedure Calls.

Java Script Object Notation (JSON) is a light weight data interchange format for language-independent client-server communication(Java Script Object Notation, 2012). Within the BlackBerry version of the WordPress mobile app, JSON is used for geocoding and fetching of page statistics from the WordPress back-end. JSON is included in the Android API `org.json` (Android API, 2012), but on the BlackBerry platform, prior to version 5.0.0, mobile app developers needed to include their own implementation of the JSON format (Android API, 2012; BlackBerry API 5, 2012; BlackBerry API 6, 2012). Although JSON is included in version 6.0.0 of the BlackBerry platform, mobile app developers still need to maintain a third party implementation of JSON to preserve backwards compatibility.(BlackBerry API 6, 2012).

Regular expressions are typically used for search and replace operations in strings and extraction of substrings. Regular expression functionality is included in the standard Java library `java.util.regex` (Android API, 2012), which is not available on the J2ME platform that is supported by the BlackBerry platform. The WordPress for BlackBerry mobile app uses the Jakarta Regexp regular expression package from the Apache Jakarta Project (Jakarta Regexp, 2012) to determine the number of characters in a comment, post or page, before posting to a web site or blog.

XML-RPC is a lightweight mechanism for exchanging data and invoking web services. XML-RPC is used by both the Android and BlackBerry versions of the

WordPress mobile app. The WordPress for BlackBerry mobile app uses the kXML-RPC implementation, a J2ME XML-RPC implementation built on top of the kXML parser (android-xmlrpc, 2012). The WordPress for Android mobile app uses the android-xmlrpc implementation, a very thin XML-RPC client library for the Android platform (kXML-RPC, 2012).

Apart from missing functionality, third party libraries are also used as an alternative to poorly implemented functionality in the language or platform APIs. One example of poorly implemented functionality on the Android platform is the visualization of lists of thumbnails off the Internet. The third party Thumbnail module from the CommonsWare Android Components library allows a mobile app to load and cache thumbnail images transparently in the background to avoid tying up the user interface thread (CommonsWare Android Components, 2012). The module has been included in the WordPress for Android mobile app. Since this module requires the use of the Cache module, the latter module has also been included in the WordPress for Android mobile app (CommonsWare Android Components, 2012). In this case, including one third party library requires the inclusion of a second third party library.

Android mobile apps rely primarily on the Android APIs, whereas BlackBerry mobile apps rely on Java libraries and project-specific classes in the mobile app. Android mobile apps contain little to no third party libraries. More than half of the dependencies on the BlackBerry platform are on user interface APIs. Android mobile apps have a much higher platform dependency ratio than feature equivalent BlackBerry mobile apps.

RQ3: How different is the amount of code churn between platforms?

Motivation

Given the rapid pace and high pressure of the mobile app development market, we want to characterize the effort needed to develop each mobile app. We also explore the code churn properties of the third party libraries to determine the amount of effort needed to maintain them, i.e., do mobile app developers highly customize such libraries, or mainly clone them.

Approach

We use the methodology presented in Section 4.1.4 to extract the code churn properties for each mobile app. Table 4.11 presents the values of these metrics and the percentage of the total number of changes (in parentheses) for each class of each mobile app pair.

We also visualize the line churn using a box plot. Box plots graphically depict the smallest observation, lower quartile, median, upper quartile, and largest observation using a box. Circles correspond to outliers. Figure 4.4, 4.5 and 4.6 depict the line churn characteristics across all commits.

We also examine the growth of the mobile apps in size (lines of code) over time. Figure 4.7 shows this evolution for the WordPress BlackBerry mobile app over the project's lifetime (from May 2009 to March 2011).

Table 4.11: Code Churn Metrics.

WordPress		
Metric	Android	BlackBerry
Total # File Changes	660	2760
Average # File Changes/File	12.00	11.45
# Third Party File Changes	47 (7%)	59 (2%)
# Project File Changes	613 (93%)	2701 (98%)
Total # Line Changes	23276	46823
Average # Line Changes/Lines	1.04	0.89
# Third Party Line Changes	245 (1%)	648 (1%)
# Project Line Changes	23031 (99%)	46175 (99%)
Google Authenticator		
Metric	Android	BlackBerry
Total # File Changes	12	12
Average # File Changes/File	1.20	0.39
#Third Party File Changes	0 (0%)	0 (0%)
#Project File Changes	12 (100%)	12 (100%)
Total #Line Changes	306	94
Average # Line Changes/Lines	0.16	0.02
# Third Party Line Changes	0 (0%)	0 (0%)
# Project Line Changes	306 (100%)	94 (100%)
Facebook SDK		
Metric	Android	BlackBerry
Total # File Changes	329	38
Average # File Changes/File	54.83	0.67
# Third Party File Changes	0 (0%)	5 (13%)
# Project File Changes	329 (100%)	33 (87%)
Total # Line Changes	2979	473
Average # Line Changes/Lines	1.80	0.05
# Third Party Line Changes	0 (0%)	23 (5%)
# Project Line Changes	2979 (100%)	450 (95%)

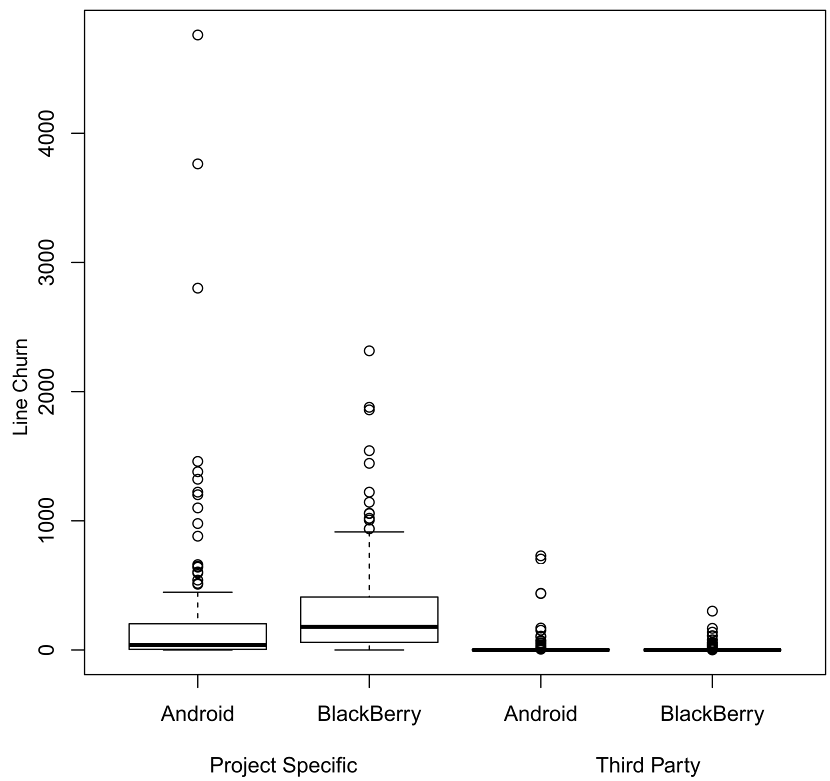


Figure 4.4: Line Churn Characteristics of the WordPress mobile app.

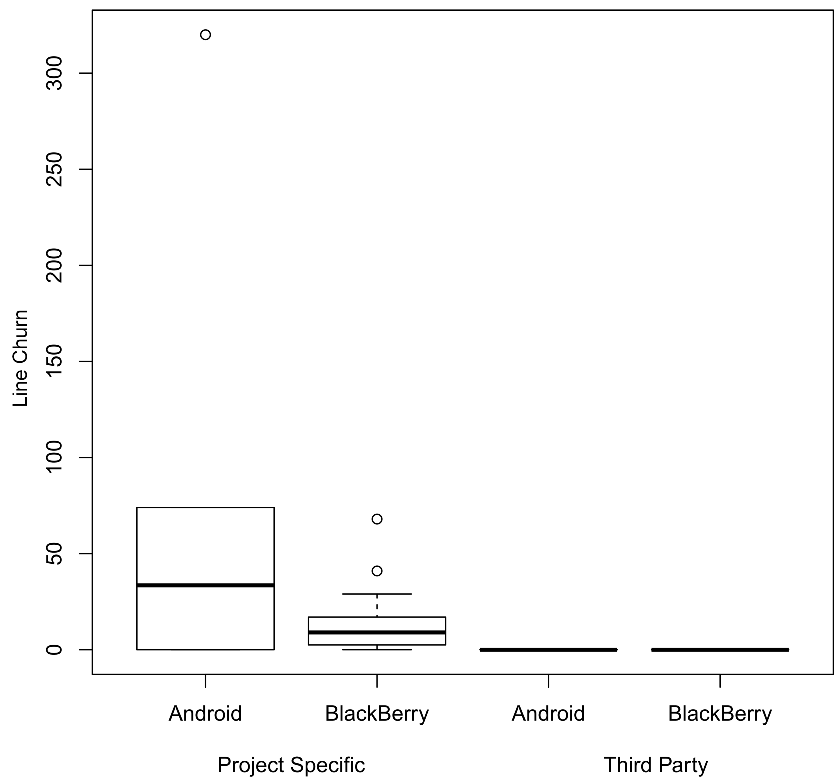


Figure 4.5: Line Churn Characteristics of the Google Authenticator mobile app.

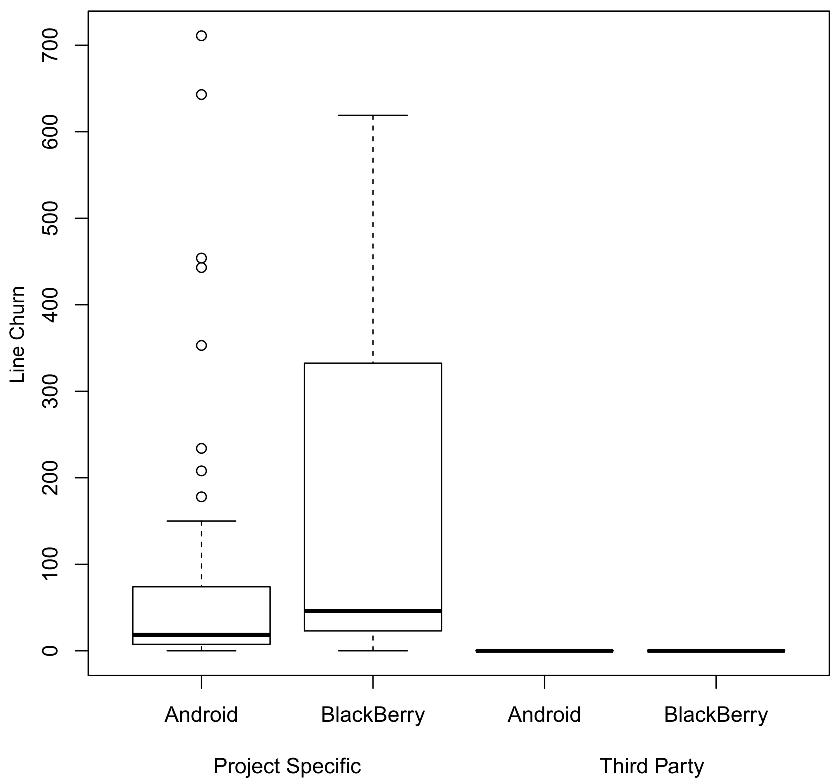


Figure 4.6: Line Churn Characteristics of the Facebook SDK.

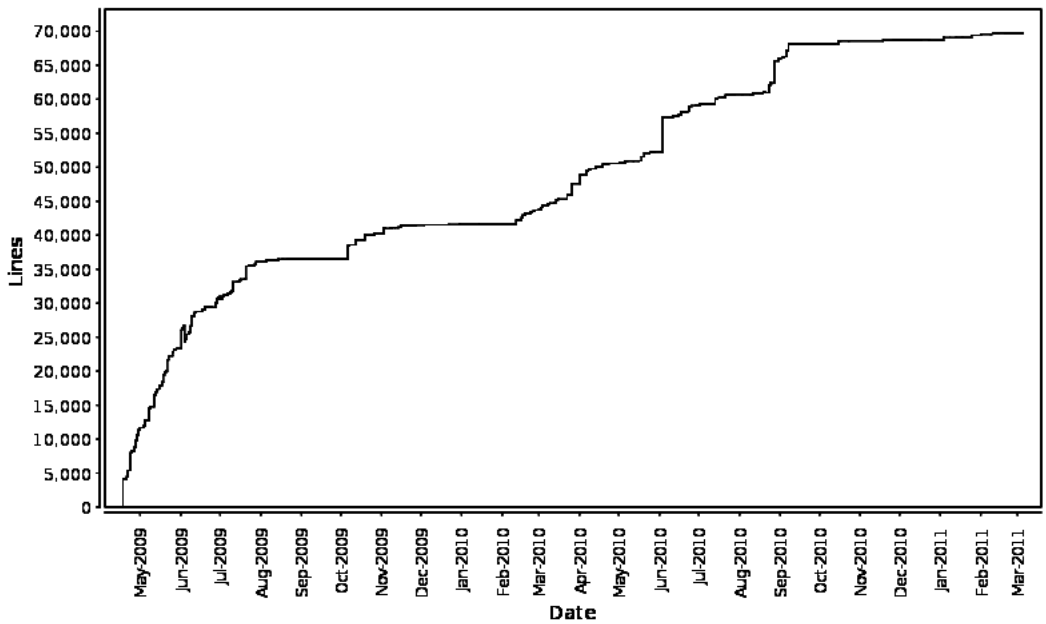


Figure 4.7: Size (lines of code) of the WordPress for BlackBerry mobile app from May 2009 to March 2011

Results

From Table 4.11, we can see that although Android mobile apps have fewer commits to their repositories, the average number of times a file is changed is much higher for Android mobile apps. For example, the average number of times a file is changed in Android Facebook SDK is 54.83 compared to 0.67 for the BlackBerry mobile app. In this case, Table 4.11 and Figure 4.6 show that Android mobile apps see many small changes, whereas BlackBerry mobile apps see fewer larger changes. On the other hand, Figure 4.4 and Figure 4.6 show that changes to BlackBerry mobile apps typically affect more lines of code, even though Android mobile apps contain more outliers. This indicates that although the size of most changes to Android mobile apps is relatively small, there are a number of relatively large changes.

From Table 4.11, Figure 4.4, 4.5 and 4.6, third party source code, on either platform, experiences very little code churn after the initial import. This suggests that these libraries are mostly just copied to make the projects self-contained, rather than to heavily customize them. Only for the WordPress mobile app, many large changes are made. We check the repository and find that 80% of these changes correspond to refactoring or fixing of defects in the kXML-RPC third party library. Without access to the third party library source code mobile app developers would not be able to fix or refactor these libraries themselves.

Figure 4.7 shows the growth of the WordPress for BlackBerry mobile app (lines of code) over time. The other five mobile apps show a similar pattern, except for the mobile apps with shorter project histories. Those mobile apps show very little growth after the initial commit.

Source code files in Android mobile apps change more frequently than source code files in BlackBerry mobile apps, but typically see smaller changes. Third party libraries typically change very little.

4.3 Threats to Validity

4.3.1 Threats to Internal Validity

Threats to interval validity describe concerns regarding alternate explanations for our results.

The Facebook SDK for BlackBerry was developed by Research In Motion (the company behind the BlackBerry) itself. This may have introduced bias into our study of platform dependencies, since the developers have intimate knowledge of the BlackBerry platform and may be biased towards relying more on BlackBerry APIs. However, the results in Table 4.9 seem to contradict this. Similarly, the Google Authenticator mobile apps for Android and BlackBerry were both developed by Google (the company behind Android). Since these developers were simultaneously developing the same mobile app for the Android and BlackBerry platforms, they may have been biased towards using Java APIs (as opposed to device-specific APIs), as well as Android APIs. Table 4.9 suggests that such a bias is possible.

Given that mobile apps like the ones that we analyzed are typically only a couple of years old, they do not have the stable project histories of long-lived, commonly studied projects like Linux and Apache. This may have biased our code churn metrics. Given this short history, it is not yet known which mobile apps are (or will be) either successful or representative of good development style.

4.3.2 Threats to Construct Validity

Threats to construct validity describe concerns regarding the measurement of our metrics.

We investigated feature equivalence between each mobile app pairs using feature lists on the mobile app webpages, change logs for each release and feature requests in the forum or issue tracking systems. However, we did not verify that the functioning versions (i.e., installed and operating on a mobile device) had these features. This may have introduced false positives into the Mobile App Selection process (i.e., two mobile apps that are thought to be feature equivalent, are, in fact, not feature equivalent). In addition, although two mobile apps may be feature equivalent, the features may have been implemented very differently (e.g., simple and straight-forward user interface compared to a more complicated interface).

The identification of third-party libraries in each mobile app was done using heuristics and manual analysis. It is possible that some third-party libraries were misidentified using this approach.

4.3.3 Threats to External Validity

Threats to external validity describe concerns regarding the generalizability our results.

The studied mobile apps represent a small subset of the total number of mobile apps available on the Android and BlackBerry platforms. In addition, we did not consider mobile app games, which are the most commonly downloaded mobile apps (Nielsen Co., 2010*a*,*b*), since we were unable to acquire a pair of feature-equivalent mobile app games. Finally, the Google Authenticator mobile app and the Facebook SDK are rather small, approximately 5,000 lines of code. However, we do not know whether these are typical sizes for a mobile app or an outlier. The results of our case study may not generalize to other mobile apps or platforms.

4.4 Conclusions

This chapter presents an exploratory study of mobile apps on two popular mobile platforms, as a first step toward understanding the development and maintenance process of mobile apps. In particular, we study the source code, dependency and code churn properties of three pairs of feature-equivalent mobile apps on the Android and BlackBerry platforms in order to address the question of how mobile app developers can target multiple platforms with limited resources.

Mobile apps written for the BlackBerry platform are more than twice the size of feature-equivalent Android mobile apps. Missing functionality in the BlackBerry and Java 2 ME APIs has forced BlackBerry mobile app developers to rely on third party libraries that have increased the size (lines of code) of the mobile app.

BlackBerry mobile apps rely less on BlackBerry-specific APIs and more on Java APIs and other classes in the mobile app. On the other hand, Android mobile app developers leverage more Android and Java SE APIs. However, heavy reliance on the Android platform has led to a greater degree of platform lock-in in these Android mobile apps. Therefore, developers who wish to target both the BlackBerry and Android platforms should write their mobile apps for the BlackBerry platform then port their mobile apps to the Android platform.

While mobile apps on both platforms experience a high degree of churn due to constant and rapid evolution, included third party libraries experience very little churn and therefore require little effort by mobile app developers to maintain.

Although Android apps and BlackBerry apps are both written in Java and intended to run on mobile devices, significant differences exist between the mobile apps of these two platforms.

Chapter 5

Platform Dependence and Source Code Quality

In the previous chapter, we performed a study of three pairs of functionally equivalent mobile apps from two popular mobile platforms (i.e., the Android and BlackBerry platforms), as a first step toward understanding the development and maintenance process of mobile apps. We found that BlackBerry apps are much larger and rely more on third party libraries. As such, they are less susceptible to platform changes since they rely less on the underlying platform. On the other hand, Android apps tended to rely heavily on the Android platform. Therefore, in this chapter, we study the relationship between source code quality and mobile platform dependency.

A mobile platform (e.g., Android) consists of numerous APIs that provide mobile apps with access to commonly required functionality or an interface to the operating system and hardware accessories. In the previous chapter, we found that developers depend heavily on platform APIs to build their mobile apps. The reason is three-fold. One, similar to web applications (Hassan and Holt, 2002), mobile apps are rapidly developed by small teams who may only have limited experience with software development (Butler, 2011; Gavalas and Economou, 2011; Lohr, 2010; Wen, 2011). Two, the rapid succession of mobile technologies and fierce competition amongst developers forces them to release new features at break-neck speed, without sacrificing quality. Three, the proliferation of mobile devices means that developers cannot make any assumptions about the environment in which their mobile apps will be operating, and hence prefer to use a standard environment. According to industry experts, leveraging the functionality provided by underlying mobile platforms is the catalyst behind the rapid development of many mobile apps (Black Duck Software Inc., 2011).

However, too much API usage of the underlying mobile platform can lock an app into that platform. This does not only have repercussions on the portability of the app to other platforms (sometimes requiring a complete rewrite), but also has a major impact on the quality of the app. For example, the rapid evolution of mobile platforms makes it hard for app developers to keep their app working on newer platform versions, leading to defects and inconsistencies that impact the end user.

The software engineering community has proposed, evaluated and noted several theories of how quality software is created and maintained. These "software engineering empirical theories" aim to tie aspects of software artifacts (e.g., size and complexity) (Chidamber and Kemerer, 1994), their development (e.g., number of changes) (Nagappan and Ball, 2005) and their developers (e.g., developer experience) (Bird et al., 2011) to definitions of quality (e.g., post-release defects). Such empirical theories are derived from a large number of empirical observations. However, to date, most observations have been made using large-scale projects such as Apache and Eclipse (Brian et al., 2010).

Dependency metrics have been used to enhance the prediction of defects in software systems. Schröter et al. showed that import dependencies can predict software defects (e.g., importing compiler packages is riskier than importing UI packages) (Schröter et al., 2006). Zimmerman and Nagappan (Zimmermann and Nagappan, 2008) performed a study of Windows Server 2003 to determine how models predicting software defects may be enhanced by using metrics based on Social Network Analysis (SNA). The authors show that SNA metrics improved the prediction of post-release failures by 10%. This study was replicated by Nguyen et al. who found similar results in the Eclipse project (Nguyen et al., 2010).

Since dependency metrics have been shown to be highly related to defects in source code files (Nguyen et al., 2010; Zimmermann and Nagappan, 2008), we are interested in analyzing whether this finding holds as well for mobile apps when dependencies are interpreted as "platform dependencies." Therefore, in this chapter, we address the following three research questions:

- *RQ1: Are source code files that depend on the Android platform more defect-prone?* – We find that source code files that rely on the Android platform tend to be more defect-prone.

- *RQ2: Does the degree of platform dependency contain unique information regarding defects?* – We find that the ratio of platform dependencies to the total number of dependencies significantly increases our ability to explain defects in source code files.

- *RQ3: Which source code metrics have the largest impact on source code quality?* – We find that increasing coupling or decreasing cohesion has a large negative impact on source code quality while increasing the platform dependency ratio has a negative impact on source code quality that is consistent across most mobile apps.

This chapter is organized as follows: Section 5.1 describes the setup of our case study and Section 5.2 discusses the results of our case study. Section 5.3 outlines the threats to validity. Finally, Section 5.4 concludes the chapter.

5.1 Case Study Setup

This section outlines our approach to understanding the relationship between source code quality and mobile platform dependency. First, we selected mobile apps for our case study. Second, we extracted source code and dependency metrics from the selected mobile apps. Finally, we calculated whether each source code file was defect-prone or defect-free.

5.1.1 Mobile App Selection

In this chapter, we studied mobile apps written for the Android platform. The Android platform is the largest (by user base) and fastest growing mobile platform. In addition, the Android platform itself is open-source and has more free and open-source mobile apps than any other major mobile platform (Black Duck Software Inc., 2010, 2011, 2012; Distimo, 2011*a*).

Mobile apps for Android devices are primarily hosted in Google Play (formerly the Android Market) (Android Market, 2012). Google Play records details such as cost, user ratings, reviews and the number of downloads in the previous 30 days for each mobile app. However, Google Play is lacking in two key metrics, i.e., the number of cumulative downloads and the development status (open-source or closed-source). Therefore, we supplement the information provided by Google Play with information from two additional sources.

First, we use FDroid, a third-party mobile app store, that exclusively contains free and open-source (FOSS) Android apps that are also listed in Google Play. As of May 1, 2012, the FDroid repository contained 236 FOSS Android apps.

Second, we use AppBrain, a third-party interface to Google Play, to get the number of cumulative downloads (App Brain, 2012).

We use the data provided by these three sources and the following criteria to select our case study subjects.

- Open-source – mobile apps must be open-source in order to access their source code repositories. This limits the number of potential case study subjects to 236, i.e., the number of mobile apps in the FDroid repository.

- Large user community – "successful" mobile apps have hundreds of thousands of downloads every month (Android Market, 2012). Therefore, in order to study what successful mobile apps are doing "right," we look at the mobile apps with at least 250,000 cumulative downloads (the highest download bracket) (Android Market, 2012; App Brain, 2012). This limits the number of potential case study subjects to 56.

- Simplicity – the code base for the mobile app must be easily identified. For example, Firefox for Android was excluded because we could not differentiate the source code of the mobile version from the desktop version because they share the same source code repository. This limits the number of potential case study subjects to 44.

- Significant code base – mobile apps must have at least 200 source code files. In regression modelling, a general rule of thumb is that at least 10 cases are required per independent variable (Harrell et al., 1984). In our experience, approximately 20% of the source code files in a mobile have are defect-prone, therefore, we need mobile apps with at least 200 source code files in order to build a regression model with four variables (see Subsection 5.1.2). This limits the number of potential case study subjects to 5.

Table 5.1 contains the final list of mobile apps that we include in our case study. We perform our case study on the entire source code history and repository as of May 1, 2012.

Table 5.1: Mobile Apps Included in Our Case Study

Project	Description	Homepage
ConnectBot	SSH Client	github.com/kruton/connectbot/
FBReader	E-Book Reader	github.com/geometer/FBReaderJ
KeePassDroid	Password Vault	github.com/bpellin/keepassdroid
Sipdroid	VOIP Client	code.google.com/p/sipdroid/
XBMCRemote	Remote Control	code.google.com/p/android-xbmcremote/

5.1.2 Source Code Metrics

We used the Understand tool by SciTools (Scitools, 2012) to extract source code metrics from each of the subject mobile apps. Understand is a static analysis toolset for measuring and analyzing the source code of small- to large-scale software projects written in a number of programming languages. We extracted the following metrics for each class in each of the subject mobile apps.

- Lines of code – total number of lines of code (LOC).
- Coupling – number of coupled classes (Coupling). Class A is said to be coupled to class B if class A uses a type, data, or member from class B.
- Cohesion – average cohesion for each class data member (Cohesion). Class A is said to be cohesive if a high percentage of class A's methods use each of class A's variables.

5.1.3 Dependency Metrics

We also used the Understand tool to extract the class dependencies for each mobile app. Class dependencies describe how each class in a mobile app depends on 1) other classes in the mobile app and 2) external libraries, e.g., the Java library, the Android library and (possibly) third-party libraries. Whereas coupling measures the total number of unique coupled classes, class dependencies measures the intensity of the coupling for each coupled class, e.g., is class A depending on class B for one method call, five method calls, or five methods calls and two data types? Therefore, we are better able to measure the degree of dependence between two classes. Further, class dependencies can be mapped to the higher level structures (e.g., packages or projects) in which the class is located (e.g., `android.app.Activity` is a class in the Android platform).

In this study, we are interested in the relationship between source code quality and mobile platform dependency (i.e., when an Android app depends on the Android platform). Therefore, for each class in a mobile app, we calculate the total number of dependencies on classes in the Android platform (Platform) in addition to the total number of dependencies (Dependencies).

We calculate each metric at the file level. We calculate lines of code, coupling, the number of platform dependencies and the total number of dependencies at the file level by summing each metric over every class in a source code file. We calculate cohesion at the file level by averaging the cohesion of each class in the file, weighted by the number of lines of code in the class.

Finally, we also calculate the platform dependency ratio, i.e., the ratio of the number of platform dependencies to the total number of dependencies (i.e., the number of platform dependencies plus the number of other dependencies). The platform dependency ratio is the dependency metric that will be used in our defect models.

5.1.4 Source Code Quality

Source code quality can be measured in a number of ways. One commonly used technique is to use the number of defects in a file as a measure of quality. In practice, this number is typically approximated by the number of defect fixing changes made to the source code file. This technique assumes that each defect fixing change corresponds to a defect in the source code file.

We find the total number of defects in a source code file by counting the number of times the file is changed by a defect fixing change. When developers contribute source code to a source code repository, they are prompted to provide an explanation (i.e., commit log message) of what they changed and why the change was made. Hence, we can find the number of defect fixing changes by mining these commit log messages for a specific set of key words (Hassan, 2008*a*; Mockus and Votta, 2000). These keywords are "fix(ed,es)", "bug(s)", "defect(s)" and "patch(s)".

Although many mobile apps record a list of tasks (e.g., defects to be fixed and features to be implemented) in an issue tracking system, we are unable to use these issues because these tend not to include information regarding how the defect was fixed (e.g., the location of the defect). Hence, heuristics based on the commit message are the only way to identify defect fixing changes due to the lack of a connection between the source code repository and issue tracking system.

5.2 Case Study Results

This section presents the results of our case study on the mobile apps selected in Subsection 5.1.1.

Preliminary Data Analysis

Prior to answering our research questions, we perform a preliminary analysis of the data by calculating descriptive statistics. Table 5.2 presents the mean, standard deviation (SD), minimum (Min) and maximum (Max) values, skew and kurtosis for each metric extracted from each project. It is necessary to study these descriptive statistics, skew and kurtosis in particular, in order to determine if transformations are required before the data can be modelled.

Skew is a measure of the amount of asymmetry in a distribution, i.e., the difference between the left and right sides of the distribution. Skew can have a positive or negative value, where a positive skew indicates that most values are concentrated to the left of the mean, with extreme values to the right and a negative skew indicates that most values are concentrated to the right of the mean, with extreme values to the left. $-0.5 \leq$ skew ≤ 0.5 indicates that the distribution is approximately symmetric.

Kurtosis (excess kurtosis) is a measure of the "peakness" of a distribution with respect to the normal distribution, i.e., the shape of the peak and tails of the distribution compared to the normal distribution. Kurtosis can have a positive or negative value, where a positive kurtosis indicates that the peak is higher and sharper and the tails are longer and thicker than the normal distribution and a negative kurtosis indicates that the peak is lower and broader and the tails are shorter and thinner than the normal distribution. $-0.5 \leq$ kurtosis ≤ 0.5 indicates that the shape of the distribution does not differ significantly from the normal distribution.

Table 5.2: Preliminary Data Analysis

Project	Metric	Mean	SD	Min	Max	Skew	Kurtosis
ConnectBot	LOC	152.10	223.34	4	1,300.00	2.54	6.98
	Coupling	13.07	25.66	0	196.00	4.59	25.00
	Cohesion	55.49	32.24	0	100.00	0.09	-1.44
	Platform	10.11	23.16	0	93.33	2.20	3.42
FBReader	LOC	85.00	126.51	2	1,199.00	3.37	17.39
	Coupling	12.38	15.28	0	114.00	2.84	10.28
	Cohesion	66.44	32.36	0	100.00	-0.31	-1.37
	Platform	9.71	18.90	0	96.43	2.08	3.68
KeePassDroid	LOC	80.61	104.73	3	799.00	3.56	17.34
	Coupling	9.54	11.97	0	79.00	3.20	12.45
	Cohesion	62.24	33.55	0	100.00	-0.22	-1.34
	Platform	13.00	27.32	0	100.00	1.97	2.47
Sipdroid	LOC	105.40	152.99	5	837.00	2.41	5.89
	Coupling	11.02	16.79	0	111.00	3.22	11.87
	Cohesion	60.10	36.60	0	100.00	-0.11	-1.63
	Platform	13.17	26.57	0	94.44	1.75	1.53
XBMCRemote	LOC	155.80	194.86	4	1,367.00	2.78	10.14
	Coupling	23.68	29.46	0	208.00	2.52	8.23
	Cohesion	49.11	31.53	0	100.00	0.39	-1.02
	Platform	15.93	25.19	0	95.83	1.33	0.54

From Table 5.2 we find that LOC, Coupling and Platform have a high (≥ 0.5)

positive skew, i.e., the metrics are concentrated to the left of the mean, with extreme values to the right, and high positive kurtosis, i.e., the peaks are higher and sharper and the tails are longer and fatter than the normal distribution. This effect can be seen in Figure 5.1. Figure 5.1 presents the distribution of LOC across the source code files of one of the studied mobile apps, i.e., ConnectBot. From Figure 5.1 and Table 5.2, we find that the source code files in the ConnectBot project vary between 4 and 1,300 LOC, but there is a peak at around 35 LOC (the left side of Figure 5.1). We find similar distributions in all of our subject mobile apps. Therefore, we log transform LOC, Coupling and Platform.

From Table 5.2 we find that Cohesion has high ($\leq$ -0.5) negative kurtosis, i.e., peaks are lower and broader and the tails are shorter and thinner than the normal distribution. We find similar distributions in all of our subject mobile apps. Therefore, we square-root transform Cohesion.

In the remainder of this chapter, whenever we refer to LOC, Coupling, Cohesion or Platform, we actually are referring to the transformed values.

RQ1: Are source code files that depend on the Android platform more defect-prone?

Motivation

Mobile apps are known to be highly dependent on both the Android and Java platforms. In the previous chapter, we defined the "platform dependency ratio" as the ratio of platform dependencies to the total number of dependencies. A low platform dependency ratio indicates that developers do not rely significantly on the platform APIs. For example, their app may be simple or self-contained, or the platform may

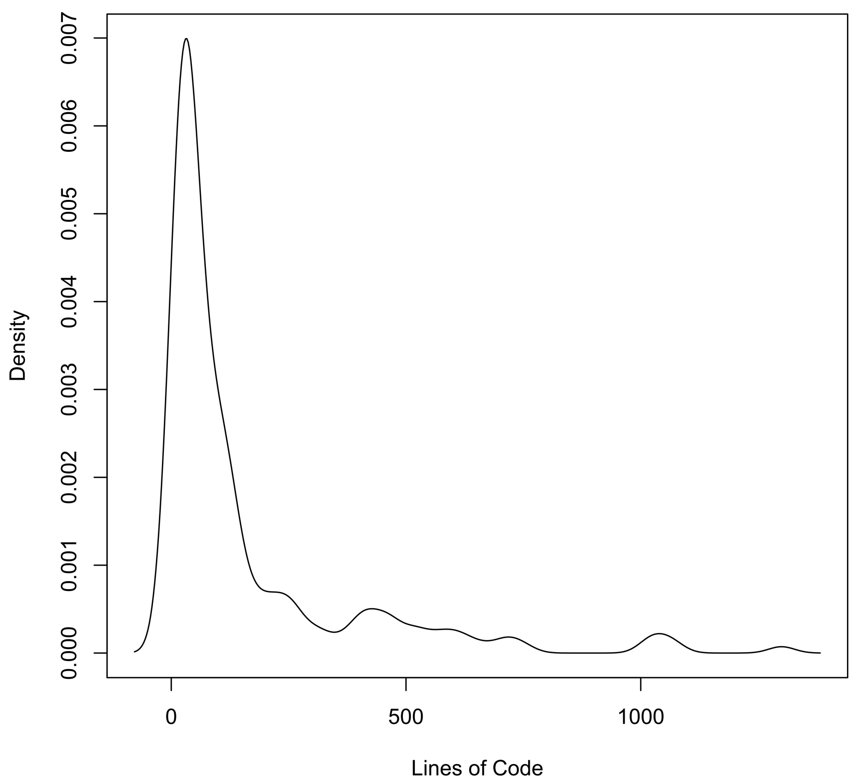


Figure 5.1: Distribution of lines of code across the source code files of ConnectBot.

be too difficult to use. Conversely, a high platform dependency ratio indicates that mobile app developers rely heavily on the platform APIs. However, this leads to platform "lock-in," which may complicate porting to other platforms and potentially introduces instability due to the rapid evolution of mobile platforms. For example, the Android platform has undergone a major release every year. Therefore, we believe that developers should be aware of the consequences of depending on these platforms.

Approach

We used three techniques to determine whether source code files that are tightly coupled to the Android platform are more defect-prone.

First, we split the source code files of each mobile app into two subsets: defect-free source code files, which have never experienced a defect, and defect-prone source code files, which have experienced at least one defect. We then visualize the distribution of platform dependency ratios across source code files with and without defects using box plots. Box plots graphically depict the smallest observation, lower quartile, median, upper quartile, and largest observation using a box. Circles correspond to outliers.

Second, use a paired t-test (parametric test) and a paired Wilcoxon signed rank test (non-parametric test) to determine if the difference between the platform dependency ratios across each subset, i.e., defect-free and defect-prone source code files, is statistically significant. The paired Wilcoxon signed rank test is resilient to strong departures from the t-test assumptions, therefore, the Wilcoxon test helps ensure that non-significant t-test results are not simply due to violations of the t-test assumptions.

Finally, we measured the spearman correlation between the platform dependency ratio and the number of defects.

Results

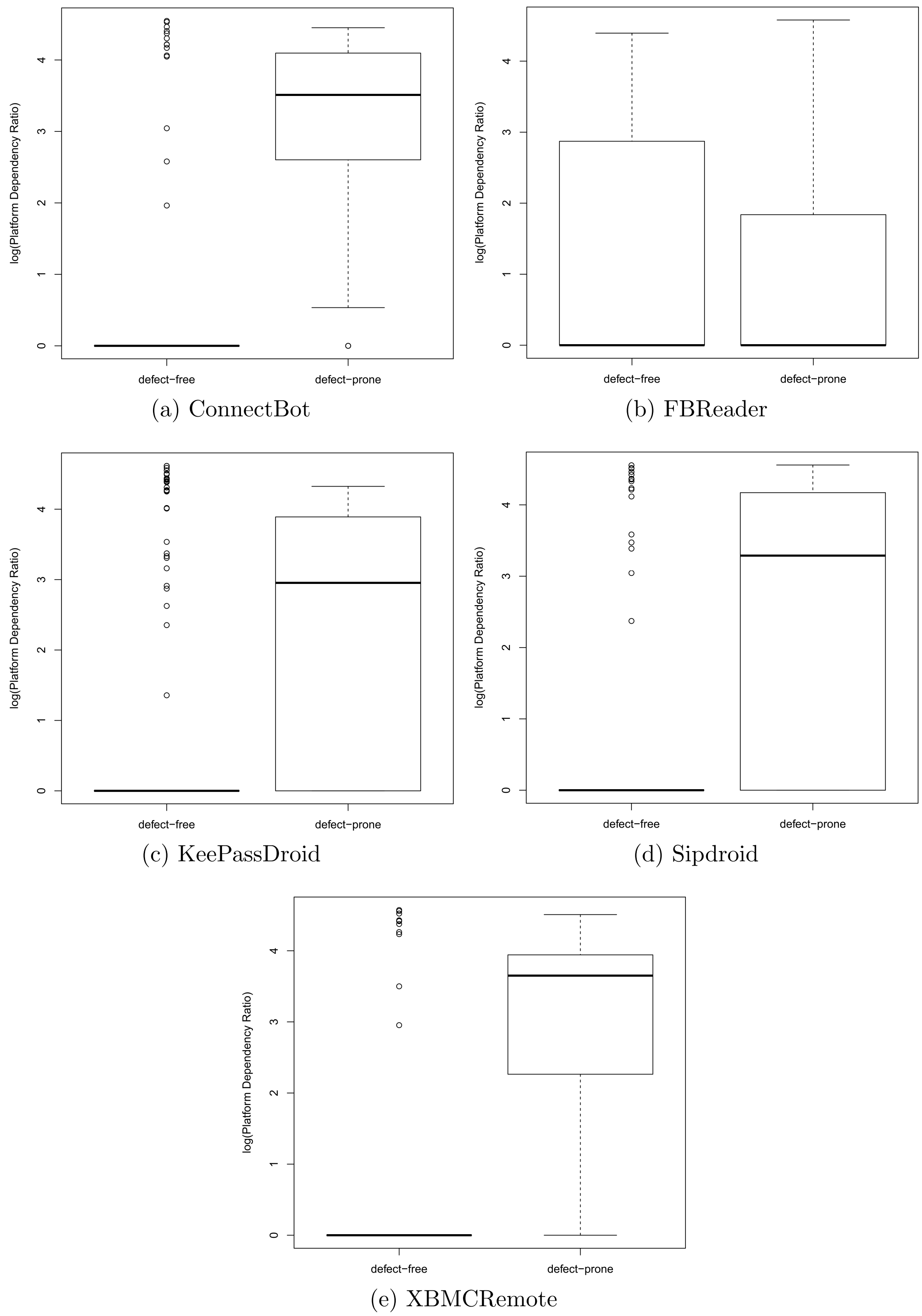
Table 5.3 shows the total number of commits to the source code repository, the number of source code files and the total number of lines of code in each mobile app. It also shows the number and percentage of source code files that are defect-prone and the the size (in lines of code) of the defect prone source code files. We can see that in ConnectBot, KeePassDroid and Sipdroid, up to 83% of the source code files are defect-free, while in FBReader and XBMCRemote around half of the source code files are defect-free.

Table 5.3: Percentage of Defect-Prone Source Code Files

Project	# Commits	# Files	# Lines of Code	% Defect-Prone Files	
				Number	Size
ConnectBot	476	201	30,572	34 (17%)	20,225 (66%)
FBReader	4685	405	34,423	220 (54%)	8,100 (24%)
KeePassDroid	492	257	20,717	49 (19%)	15,155 (73%)
Sipdroid	622	202	21,290	53 (26%)	8,989 (42%)
XBMCRemote	781	301	46,895	121 (40%)	25,179 (54%)

We then visualize the distribution of platform dependency ratios across source code files with and without defects for each mobile app using box plots. Figures 5.2a, 5.2b, 5.2c, 5.2d and 5.2e present these box plots.

From Figures 5.2a, 5.2c, 5.2d and 5.2e, we find that defect-prone source code files tend to rely on the platform libraries more than defect-free source code files. The median platform dependency ratio in defect-prone source code files across all mobile apps is 3.25 (26%), whereas the median platform dependency ratio in defect-free source code files across all mobile apps is 0. Further, in all cases except FBReader, most defect-prone source code files have at least some dependence on the platform (median



(a) ConnectBot

(b) FBReader

(c) KeePassDroid

(d) Sipdroid

(e) XBMCRemote

Figure 5.2: Distribution of platform dependency ratios across the defect-free and defect-prone source code files

$\geq$ 0), whereas most source code files that are defect-free also have no dependencies on the platform (the log transform of zero is zero).

However, from Figure 5.2b, we find that one project, i.e., FBReader, does not appear to show a significant difference between the distribution of platform dependency ratios across source code files with and without defects. It is interesting to note that FBReader is an outlier in many respects. From Table 5.3, we find that FBReader has many more commits and source code files than any other project. Further, we find that the source code files of FBReader are generally more defect-prone.

The outlier status of FBReader is also clear from Table 5.4, which presents the results of a paired t-test and a paired Wilcoxon signed rank test performed to determine if the difference between the distribution of platform dependency ratios across source code files with and without defects, seen in Figures 5.2a, 5.2b, 5.2c, 5.2d and 5.2e, are statistically significant. The values in bold indicate that the difference is statistically significant ($p \leq 0.05$). Again, the difference between the distribution of platform dependency ratios across source code files with and without defects is statistically significant in all mobile apps except FBReader.

Table 5.4: T-Tests and Wilcoxon Tests

Project	t-test	Wilcoxon test
ConnectBot	$\mathbf{2.77 * 10^{-12}}$	$\mathbf{3.41 * 10^{-21}}$
FBReader	$9.68 * 10^{-2}$	$9.12 * 10^{-2}$
KeePassDroid	$\mathbf{1.93 * 10^{-5}}$	$\mathbf{1.29 * 10^{-7}}$
Sipdroid	$\mathbf{8.87 * 10^{-9}}$	$\mathbf{1.44 * 10^{-12}}$
XBMCRemote	$\mathbf{2.59 * 10^{-35}}$	$\mathbf{2.50 * 10^{-33}}$

Finally, Table 5.5 presents the spearman correlation between the platform dependency ratio and the number of defects, as well as the spearman correlation between

LOC and the number of defects. We select LOC for comparison (i.e., a baseline) because it has been shown to be highly correlated with defects (Chidamber and Kemerer, 1994; Nagappan and Ball, 2005; Shihab et al., 2010; Zimmermann et al., 2007). Indeed, from Table 5.5, we find that LOC of a source code file has a moderately positive correlation (median of 0.38) with the number of defects in that source code file for all of our mobile apps. This correlation is similar to the observed correlation in desktop applications. For example, Zimmerman et al. (Zimmermann et al., 2007) found a correlation of 0.40 between LOC and defects in Eclipse.

Table 5.5: Correlation Between Source Code Metrics and Defects

Project	LOC	Platform
ConnectBot	0.38	0.67
FBReader	0.44	-0.06
KeePassDroid	0.16	0.33
Sipdroid	0.43	0.53
XBMCRemote	0.21	0.72
Median	0.38	0.53

From Table 5.5, we find that the spearman correlation between the platform dependency ratio and the number of defects (Platform) is higher than the spearman correlation between LOC and defects in four of the five mobile apps. The median Platform correlation is 0.53, which indicates a strong positive relationship, and is greater than the median LOC correlation.

Source code files that are more tightly coupled to the Android platform tend to be more defect-prone than source code files that are less tightly coupled to the Android platform.

RQ2: Does the degree of platform dependency contain unique information regarding defects?

Motivation

In our previous research question, we found that there is a moderate positive relationship between the platform dependency ratio and defects. In this research question, we study whether the platform dependency ratio contributes unique information to our understanding of defect-proneness. In particular, we study whether combining traditional source code metrics with the platform dependency ratio can enhance our ability to explain the defect-proneness of source code files.

Approach

We built logistic regression models and used two techniques to determine whether the platform dependency ratio can help in explaining defects. Logistic regression models allow us to determine the relationship between the platform dependency ratio and defect-proneness while controlling for other metrics, i.e., lines of code, coupling and cohesion.

In order to build logistic regression models, we first characterized each source code file as either defect-prone (at least one defect) or defect-free (no defects).

We then build two logistic regression models for each mobile app. The first model (Traditional model) was built using traditional metrics (LOC, Coupling and Cohesion (Chidamber and Kemerer, 1994; Nagappan and Ball, 2005)). The second model (Full model) was built using both traditional metrics and Platform.

We remove any overly influential observations, i.e., data points that may have a disproportionate influence on the value of one or more of the estimated coefficients in the regression models. We analyze the impact that each observation has on our model using dfbeta residuals. The dfbeta residual approximates the influence of each observation by calculating, for each coefficient, the ratio of the change in the coefficient when an individual observation is removed to the coefficient's standard error. In small data sets, overly influential observations will have dfbeta residuals with absolute values greater than 1 (Cohen et al., 2002; der Meera et al., 2010). We removed overly influential observations and rebuilt our models on the new data sets.

We then assess the statistical significance of each coefficient in the new Full model to determine which metrics are statistically significant when modelling defects.

Finally, we compare the two models by calculating the change in explanatory power from the Traditional model to the Full model. The explanatory power of a logistic regression model varies between 0-100% and quantifies the variability of the data set that is explained by the model. An explanatory power of 100% indicates that our model can perfectly explain the data set.

Results

We calculated dfbeta residuals for each coefficient in each model. Figure 5.3 presents these dfbeta residuals for the coefficient modelling LOC in ConnectBot.

From Figure 5.3, we find that there are two overly influential observations (observation 106 and 123). Therefore, we removed these observations from the ConnectBot data set (no other mobile app project had any overly influential observations). We repeated this procedure for each coefficient in each project. We then rebuilt our models.

Table 5.6 presents the coefficients in the full model, i.e., the models built with both

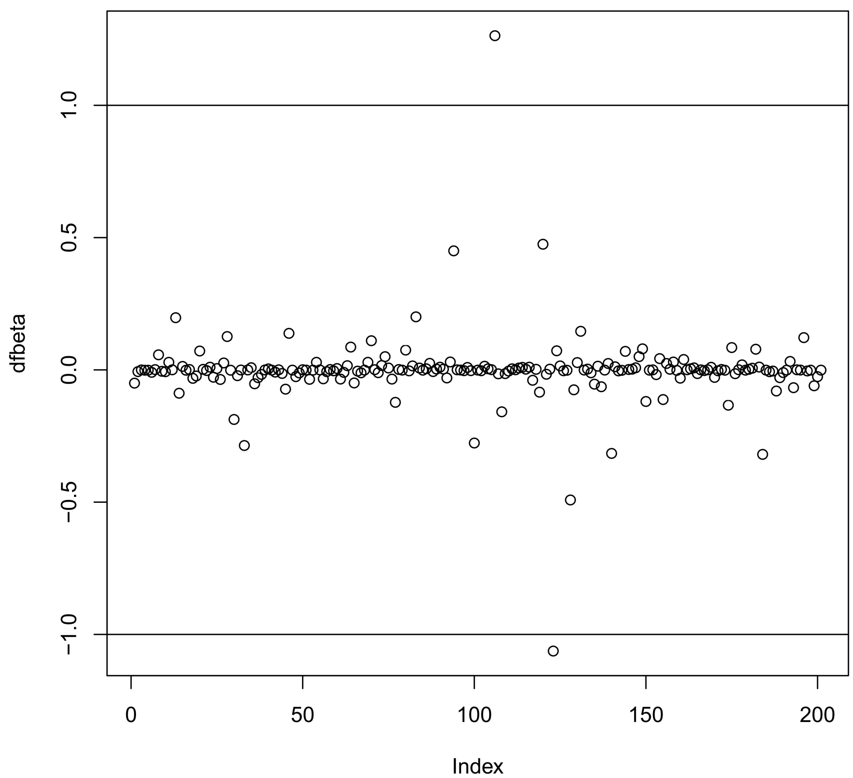


Figure 5.3: DFBeta residuals for the coefficient modelling lines of code in ConnectBot.

traditional metrics and the platform dependency ratio, for each mobile app (recall that these metrics were transformed in Subsection 5.2). The coefficients in bold are statistically significant (p $\leq$ 0.05).

Table 5.6: Coefficients in the Full Model of each Mobile App.

Project	(Intercept)	LOC	Coupling	Cohesion	Platform
ConnectBot	-0.562	-0.522	**1.461**	**-0.572**	**0.967**
FBReader	**-3.017**	**0.718**	0.247	0.033	**-0.241**
KeePassDroid	-1.120	**-0.897**	**1.902**	**-0.173**	0.212
Sipdroid	**-2.972**	0.170	**0.897**	**-0.186**	**0.483**
XBMCRemote	**-1.875**	0.288	-0.287	-0.071	**1.084**

From Table 5.6, we find that Platform is significant in four mobile apps and Coupling is significant in three mobile apps. This is strong evidence that dependency metrics can be used to explain defects in source code files. Further, LOC is statistically significant in only two mobile apps. Cohesion is statistically significant in three mobile apps. It is interesting to note that, despite being related measures, coupling and the platform dependency ratio appear to complement each other. Coupling is statistically significant in KeePassDroid whereas the platform dependency ratio is not and the platform dependency ratio is statistically significant in FBReader and XBMCRemote whereas coupling is not statistically significant.

It is interesting to note that KeePassDroid was ported from another platform, whereas ConnectBot, FBReader, Sipdroid and XBMCRemote were developed as Android apps. This may explain why all traditional metrics (i.e., LOC, Coupling and Cohesion) are statistically significant in KeePassDroid but Platform is not significant.

From Table 5.6, we find that the platform dependency ratio is statistically significant and appears to enhance traditional source code metrics. To verify this, we compare the explanatory power of a logistic regression model (Traditional) built using traditional metrics (LOC, Coupling and Cohesion) and a logistic regression model (Full) built using both traditional metrics and the platform dependency ratio. We perform ANOVA analysis to determine if the difference between the Traditional and the Full model is statistically significant. The values in bold indicate that the increase in explanatory power is statistically significant. Table 5.7 presents this data, as well as the median explanatory power across all five Traditional models (one for each mobile app) and all five Full models (one for each mobile app). Values in bold (in the Full column) indicate that the platform dependency ratio is statistically significant in this model ($p \leq 0.05$).

Table 5.7: Deviance Explained by Traditional and Full Models

Project	Traditional	Full	Difference
ConnectBot	42.33	**59.15**	+40%
FBReader	11.58	**13.09**	+13%
KeePassDroid	21.89	23.31	+7%
Sipdroid	29.98	**35.79**	+19%
XBMCRemote	5.45	**40.42**	+641%
Median	21.89	35.79	+63%

From Table 5.7, we find that adding the platform dependency ratio to our models increases the explanatory power. The median explanatory power using traditional source code metrics is 21.89 and the median explanatory power using traditional source code metrics combined with Platform is 35.79 (a 63% increase).

The smallest increase in deviance explained (7%) is in KeePassDroid, where the difference between the Full model and the Traditional model is not statistically significant. As previously mentioned, KeePassDroid was ported from another platform and the platform dependency ratio is not a statistically significant predictor. Conversely, the largest increase in deviance explained (641%) is in XBMCRemote. This may because XBMCRemote has a greater portion of its source code files depending on the Android platform. Table 5.8 presents the percentage of source code files that depend on the Android Platform.

Table 5.8: Percentage of Source Code Files Depending on the Android Platform.

Project	% Source Code Files
ConnectBot	21%
FBReader	30%
KeePassDroid	24%
Sipdroid	24%
XBMCRemote	36%
Median	24%

From Table 5.8, we find that a greater portion of the source code files in XBMCRemote depend on the Android platform compared to any other mobile apps.

> The platform dependency ratio can help in explaining defects in source code files.

RQ3: Which source code metrics have the largest impact on source code quality?

Motivation

In our previous research question, we found that the platform dependency ratio can help in explaining defects in source code files. In this research question we study which source code metrics have the largest impact on source code quality. In particular, we study the effects of a proportional increase in each source code metric. For example, does doubling LOC introduce more defects relative to doubling the number of coupled classes?

Approach

In the previous research question, we built a logistic regression model for each mobile using both traditional metrics (LOC, Coupling and Cohesion) and Platform. Here, we calculate the change in defect-proneness due to a proportional increase of each source code metric to determine which source code metric has the largest impact on source code quality.

To do that, we first calculate the average value of each source code metric, i.e., LOC, Coupling, Cohesion and Platform. Similar to Shihab et al. (Shihab et al., 2011), a baseline hypothetical source code file is built using the average value for each source code metric. Four hypothetical files are constructed by increasing each source code metric in the baseline by 10%, one at a time (keeping the other metrics constant at their average value). Table 5.9 shows the hypothetical source code files for ConnectBot.

We use the logistic models built in the previous question to predict the defect-proneness for each hypothetical source code file. The defect-proneness is the probability that a source code file is defect-prone. Finally, we calculate the change in defect-proneness of each hypothetical source code file compared to the baseline.

Table 5.9: Hypothetical Source Code Files Used to Assess the Impact of Each Source Code Metric on Defect-Proneness.

File	LOC	Coupling	Cohesion	Platform
Baseline	4.23	1.92	7.10	0.73
File1	**4.65**	1.92	7.10	0.73
File2	4.23	**2.11**	7.10	0.73
File3	4.23	1.92	**7.80**	0.73
File4	4.23	1.92	7.10	**0.81**

Results

Table 5.10 presents the change in defect-proneness from a 10% increase in each metric over the baseline (average) values. For example, the first value in the first row, -19.26%, indicates that increasing LOC by 10%, decreases the probability that a source code file is defect-prone by 19.26%. The values in bold in Table 5.10 correspond to a 10% increase in a metric that was found to be statistically significant in Table 5.6.

Table 5.10: Impact of an Increase in Each Source Code Metric on Defect-Proneness.

Project	LOC	Coupling	Cohesion	Platform
ConnectBot	-19.26%	**30.89%**	**-32.56%**	**7.10%**
FBReader	**11.71%**	2.33%	1.14%	**-1.04%**
KeePassDroid	**-26.63%**	**37.36%**	**-10.78%**	1.69%
Sipdroid	5.44%	**15.01%**	**-10.49%**	**3.66%**
XBMCRemote	7.80%	-4.48%	-2.79%	**8.51%**
Median	5.44%	15.01%	-10.49%	3.66%

From Table 5.10, we find that Coupling and Cohesion have the greatest impact on defects. “High cohesion and low coupling leads to high quality” is a classic software engineering concept (Chidamber and Kemerer, 1994).

Although the platform dependency ratio does not have the greatest effect on defect-proneness, it is the most consistent. The range (i.e., the difference between the maximum and minimum values) for LOC, Coupling and Cohesion are 38.34%, 41.84% and 33.7% respectively, whereas the range for Platform is only 9.55%.

From Table 5.10, we see that the platform dependency ratio has the smallest impact on source code quality, despite its ability to significantly increase the explanatory power of our models. This may because the average platform dependency ratio is low and only a subset of the source code files (20%-36%) actually depend on the platform. This can be seen in Tables 5.2, 5.8 and 5.9. Therefore, knowing that a source code file has *any* dependence on the platform may be enough to identify defect-prone source code files.

Finally, LOC has an inconsistent impact on source code quality. A 10% increase in LOC in FBReader, Sipdroid and XBMCRemote increases defect-proneness by, on average, 8%, whereas a 10% increase in LOC in ConnectBot and KeePassDroid decreases defect-proneness by, on average, 23%.

Coupling has the largest impact on source code quality while Platform has the most consistent impact on source code quality.

5.3 Threats to Validity

5.3.1 Threats to Construct Validity

Threats to construct validity describe concerns regarding the measurement of our metrics.

The number of defects in each source code file was measured by identifying the source code files that were changed in a defect fixing change. Although this technique has been found to be effective (Hassan, 2008*a*; Mockus and Votta, 2000), it is not without flaws. We identified defect fixing changes by mining the commit logs for a set of keywords. Therefore, we are unable to identify defect fixing changes (and therefore defects) if we failed to find a specific keyword, if the committer misspelled the keyword or if the committer failed to include any commit message. We are also unable to determine which source code files have defects when defect fixing modifications and non-defect fixing modifications are made in the same commit.

5.3.2 Threats to External Validity

Threats to external validity describe concerns regarding the generalizability our results.

We have limited our study to a very small subset of open-source mobile apps. In addition, we have only studied the mobile apps of a single mobile platform (i.e., the Android Platform). Finally, we did not consider mobile app games, which are the most commonly downloaded mobile apps, because we were unable to find mobile app games that met our requirements (Nielsen Co., 2010*a*,*b*). Therefore, it is unclear how our results will generalize to 1) other mobile apps, 2) close-source mobile apps and 3) other mobile platforms.

In addition to the aforementioned threats to validity, our selection of mobile apps excluded, by necessity, mobile apps with small code bases, short histories and poor documentation. Therefore, it is unclear how our results will generalize to these types of mobile apps.

The dependency metrics used in this study are very simple. For example, we do not consider the functionality provided by the dependency, the complexity of setting up the dependency or the source code quality of the source or target of the dependency. Therefore, our results may not apply to other types of dependency metrics.

Other more complex dependency metrics should be explored, especially given that our results indicate that the platform dependency ratio plays a significant role in the defect-proneness of a source code file.

5.4 Conclusions

This chapter presented a study of the relationship between source code quality and mobile platform dependency. In particular, we studied whether source code files that are tightly coupled to the Android platform are more defect-prone, whether the platform dependency ratio can help in explaining defects and which source code metrics have the largest impact on source code quality. We answered these questions by studying five open-source mobile apps written for the Android platform.

We find that source code files that are more tightly coupled to the Android platform tend to be more defect-prone. However, the underlying reasons remain unclear, are Android APIs hard to use? Are they more buggy? Do developers avoid relying on a rapidly evolving platform? We intend to address these questions in future studies. In the mean time, developers looking to prioritize their testing efforts should consider testing source code files with the highest platform dependency ratios.

We also find that mobile apps do exhibit some of the classical relationships between source code metrics and quality, e.g., "high cohesion and low coupling lead to high quality" (Chidamber and Kemerer, 1994), but not necessarily others, e.g., "larger source code files are more defect prone" holds in three of the five studied mobile apps. Developers should focus on creating cohesive classes coupled to the minimum number of classes.

In the future we intend to extend our analysis to additional mobile apps and mobile platforms. We also intend to divide the dependencies into finer categories. For example, instead of treating the entire Android platform as one category, it could be split into User Interface APIs, Networking APIs, Persistent Data APIs, etc.

Chapter 6

Conclusions and Future Work

This chapter summarizes our findings from the previous chapters. We also present the limitations of our results and outline directions for future work.

6.1 Summary

Despite the ubiquity of mobile devices and the popularity of mobile apps, few researchers have studied mobile apps from a software engineering perspective. Software engineering researchers have proposed and evaluated several theories of how high quality, successful software is developed and maintained. However, such software engineering concepts have primarily been evaluated against large-scale projects. The relationship between these large-scale projects and mobile apps and the applicability of these software engineering concepts to mobile apps is unclear.

Hence, we perform three quantitative studies to validate our research hypothesis:

The mobile app sector is rapidly becoming the largest sector of software today. Yet there is very limited research done to understand the development practices and the quality of such mobile apps. We believe that these mobile apps bring a unique set of challenges to software engineering practice and research.

We find that the scale of mobile apps is much smaller than traditionally studied desktop/server applications. The development of a mobile app tends to be driven by only one of two developers. These developers tend to rely heavily on functionality provided by the underlying mobile platform through platform-specific APIs. However, the degree to which developers rely on these platform APIs varies between platforms. Developers of BlackBerry apps are less dependent on BlackBerry APIs than Android developers are on Android APIs. Finally, defects in Android apps tend to be concentrated in a small number of source code files that are tightly coupled to the Android platform.

In this thesis, we presented evidence to support our hypothesis that mobile apps differ from traditionally studied desktop/server applications. Researchers should begin to study mobile apps alongside these traditionally studied desktop/server applications given the increasing popularity of mobile apps. We also presented differences between Android apps and Blackberry apps. Mobile app developers who wish to target both the BlackBerry and Android platforms should write their mobile apps for the BlackBerry platform then port their mobile apps to the Android platform. Finally, we believe that the developers of mobile apps and mobile platforms should be aware of the positive relationship between platform dependence and defects. However, further investigation is required to understand why this relationship exists in order to facilitate the development and maintenance of high quality of mobile apps.

6.2 Limitations and Future Work

The limitations of each our of three studies were presented in their respective chapters. However, there are some limitations that crosscut all three of our studies and threaten the validity of our thesis.

The mobile apps selected for our case studies represent a small subset of the total number of mobile apps available. In addition, we have limited our studies to open source mobile apps. Further, the studies in Chapter 3 and Chapter 5 were limited to mobile apps from a single mobile platform (i.e., the Android Platform). Therefore, it is unclear how our results will generalize to 1) closed source mobile apps and 2) other mobile platforms. In the future, we can extend our studies to these types of mobile apps. Despite this limitation, we believe that the mobile apps in our case study are a good representation of the larger mobile app ecosystem. The Android platform is the largest and fastest growing mobile platform. In addition, there are more open source mobile apps on the Android platform than any other mobile platform (Black Duck Software Inc., 2010, 2011, 2012).

The metrics selected to compare mobile apps to desktop/server applications in Chapter 3 and Android apps to Blackberry apps Chapter 4 represent a small subset of the total number of dimensions along which software projects can be compared. Therefore, it is unclear how our results may change when comparing along other dimensions. In the future, we can extend our studies to include comparisons along additional dimensions. Despite this limitation, we believe that our comparisons were sufficient to address our research questions.

Finally, our results indicate that there is a statistically significant relationship between source code quality and platform dependence. However cause of this relationship is unclear. Are Android APIs poorly documented? Are they difficult to use? Are they buggy? Should developers avoid relying on specific APIs? Future work should address these questions by contacting developers for their input. Despite this limitation, we believe that the existence of a statistically significant relationship between source code quality and platform dependence is cause for concern given the high degree of platform dependence.

Bibliography

All Facebook (2010), '10 surprising app platform facts', http://allfacebook.com/app-platform-facts_b18514. Last viewed: 20-Oct-2012.

Android API (2012), 'Android package index', http://developer.android.com/reference/packages.html. Last viewed: 20-Oct-2012.

Android Developers (2012), 'Hello, World - Android Developers', http://developer.android.com/training/basics/firstapp/index.html. Last viewed: 20-Oct-2012.

Android Market (2012), 'Android Market', https://play.google.com/store. Last viewed: 20-Oct-2012.

android-xmlrpc (2012), 'android-xmlrpc', http://code.google.com/p/android-xmlrpc/. Last viewed: 20-Oct-2012.

App Brain (2012), 'App brain', http://www.appbrain.com/. Last viewed: 20-Oct-2012.

Apple App Store (2012), 'Apple App Store', http://www.apple.com/iphone/from-the-app-store/. Last viewed: 20-Oct-2012.

Bird, C., Nagappan, N., Murphy, B., Gall, H. and Devanbu, P. (2011), Don't touch my code! examining the effects of ownership on software quality, *in* 'Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering', pp. 4–14.

Black Duck Software Inc. (2010), 'Android wins over open source mobile developers, growing 3x faster than iphone', `http://blackducksoftware.com/news/releases/2010-03-16`. Last viewed: 20-Oct-2012.

Black Duck Software Inc. (2011), 'Mobile innovation, growth driven by open source', `http://blackducksoftware.com/news/releases/2011-03-02`. Last viewed: 20-Oct-2012.

Black Duck Software Inc. (2012), 'Android and enterprise benefit from mobile open source development', `http://blackducksoftware.com/news/releases/2012-05-15`. Last viewed: 20-Oct-2012.

BlackBerry API 5 (2012), 'BlackBerry JDE 5.0.0 API Reference', `http://www.blackberry.com/developers/docs/5.0.0api/index.html`. Last viewed: 20-Oct-2012.

BlackBerry API 6 (2012), 'BlackBerry JDE 6.0.0 API Reference', `http://www.blackberry.com/developers/docs/6.0.0api/index.html`. Last viewed: 20-Oct-2012.

BlackBerry App World (2012), 'BlackBerry App World', `http://appworld.blackberry.com/webstore/`. Last viewed: 20-Oct-2012.

Brian, Robinson and Francis, P. (2010), Improving industrial adoption of software engineering research: a comparison of open and closed source software, *in* 'Proceedings of the International Symposium on Empirical Software Engineering and Measurement', pp. 197–206.

Brooks, F. P. (1975), *The mythical man-month – Essays on Software-Engineering*, Addison-Wesley.

Butler, M. (2011), "Android: Changing the mobile landscape", *IEEE Pervasive Computing* .

Charland, A. and LeRoux, B. (2011), "Mobile application development: Web vs. native", *Queue* , Vol. 9, pp. 20–28.

Chetan Sharma Consulting (2010), 'Sizing up the global apps market', `http://chetansharma.com/mobileappseconomy.htm`. Last viewed: 20-Oct-2012.

Chidamber, S. and Kemerer, C. (1994), "A metrics suite for object oriented design", *Transactions on Software Engineering* , Vol. 20, pp. 476–493.

ChurnExtension (2012), 'ChurnExtension - Mercurial', `http://mercurial.selenic.com/wiki/ChurnExtension`. Last viewed: 20-Oct-2012.

Cohen, J., Cohen, P., West, S. G. and Aiken, L. S. (2002), *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*, third edn, Routledge Academic.

CommonsWare Android Components (2012), 'CommonsWare Android Components', `http://commonsware.com/cwac`. Last viewed: 20-Oct-2012.

ComScore Inc. (2010), 'May 2010 U.S. Mobile Subscriber Market Share', `http://www.comscore.com/Press\_Events/Press\_Releases`. Last viewed: 20-Oct-2012.

der Meera, T. V., Grotenhuisb, M. T. and Pelzerb, B. (2010), "Influential cases in multilevel modeling: A methodological comment", *American Sociological Review* , Vol. 75, pp. 173–178.

Dinh-Trong, T. T. and Bieman, J. M. (2005), "The freebsd project: A replication case study of open source development", *Transactions on Software Engineering* , Vol. 31, pp. 481–494.

Distimo (2011*a*), 'Comparisons and contrasts: Windows phone 7 marketplace and google android market', `http://www.distimo.com/publications`. Last viewed: 20-Oct-2012.

Distimo (2011*b*), 'In-depth view on download volumes in the google android market', `http://www.distimo.com/publications`. Last viewed: 20-Oct-2012.

Distimo (2012), 'The need for cross app store publishing and the best strategies to pursue', `http://www.distimo.com/publications`. Last viewed: 20-Oct-2012.

Enck, W., Ongtang, M. and McDaniel, P. (2009), "Understanding Android Security", *Security and Privacy Magazine* , Vol. 7, pp. 50–57.

F-Droid (2012), 'F-droid', `http://f-droid.org/`. Last viewed: 20-Oct-2012.

Facebook SDK for Android (2012), 'Facebook SDK for Android', `https://github.com/facebook/facebook-android-sdk`. Last viewed: 20-Oct-2012.

Facebook SDK for BlackBerry (2012), 'Facebook SDK for BlackBerry', `http://sourceforge.net/projects/facebook-bb-sdk/`. Last viewed: 20-Oct-2012.

Gartner Inc. (2011), 'Gartner says worldwide mobile application store revenue forecast to surpass $15 billion in 2011', `http://www.gartner.com/it/page.jsp?id=1529214`. Last viewed: 20-Oct-2012.

Gasimov, A., Tan, C.-H., Phang, C. W. and Sutanto, J. (2010), Visiting mobile application development:what, how, and where, *in* 'Proceedings of the International Conference on Mobile Business and Global Mobility Roundtable', pp. 74–81.

Gavalas, D. and Economou, D. (2011), "Development platforms for mobile applications: Status and trends", *IEEE Software* , Vol. 28, pp. 77–86.

Geldenhuys, J. (2010), Finding the core developers, *in* 'EUROMICRO Conference on Software Engineering and Advanced Applications', pp. 447–450.

git-log (2012), 'git-log', `http://kernel.org/pub/software/scm/git/docs/git-log.html`. Last viewed: 20-Oct-2012.

Gittens, M., Kim, Y. and Godwin, D. (2005), The vital few versus the trivial many: examining the pareto principle for software, *in* 'International Computer Software and Applications Conference', pp. 179–185.

Google Authenticator (2012), 'Google Authenticator', `http://code.google.com/p/google-authenticator/`. Last viewed: 20-Oct-2012.

Grace, M. C., Zhou, W., Jiang, X. and Sadeghi, A.-R. (2012), Unsafe exposure analysis of mobile in-app advertisements, *in* 'Proceedings of the conference on Security and Privacy in Wireless and Mobile Networks', pp. 101–112.

Grace, M. C., Zhou, Y., Zhang, Q., Zou, S. and Jiang, X. (2012), Riskranker: scalable and accurate zero-day android malware detection, *in* 'Proceedings of the international conference on Mobile systems, applications, and services', pp. 281–294.

Hammershoj, A., Sapuppo, A. and Tadayoni, R. (2010), Challenges for mobile application development, *in* 'International Conference on Intelligence in Next Generation Networks', pp. 1–8.

Han, D., Zhang, C., Fan, X., Hindle, A., Wong, K. and Stroulia, E. (2012), Understanding android fragmentation with topic analysis of vendor-specic bugs, *in* 'Proceedings of the Working Conference on Reverse Engineering'.

Harman, M., Yue, J. and Yuanyuan, Z. (2012), App store mining and analysis: Msr for app stores, *in* 'Working Conference on Mining Software Repositories', pp. 108–111.

Harrell, F. E., Lee, K. L., Califf, R. M., Pryor, D. B. and Rosati, R. A. (1984), "Regression modelling strategies for improved prognostic prediction.", *Statistics in Medicine* , Vol. 3, pp. 143–152.

Hassan, A. E. (2008*a*), Automated classification of change messages in open source projects, *in* 'Proceedings of the Symposium on Applied Computing', pp. 837–841.

Hassan, A. E. (2008*b*), The road ahead for Mining Software Repositories, *in* 'Frontiers of Software Maintenance', pp. 48–57.

Hassan, A. E. and Holt, R. C. (2002), Architecture recovery of web applications, *in* 'Proceedings of the International Conference on Software Engineering', pp. 349–359.

Herraiz, I., Gonzalez-Barahona, J. M. and Robles, G. (2007), Towards a theoretical model for software growth, *in* 'Proceedings of the International Workshop on Mining Software Repositories', pp. 21–28.

Hu, W., Chen, T., Shi, Q. and Lou, X. (2010), Smartphone software development course design based on android, *in* 'Proceedings of the International Conference on Computer and Information Technology', pp. 2180–2184.

International Data Corp. (2011), 'Idc forecasts nearly 183 billion annual mobile app downloads by 2015: Monetization challenges driving business model evolution', `http://www.idc.com/getdoc.jsp?containerId=prUS22917111`. Last viewed: 20-Oct-2012.

Israel, M. R. J., Nagappan, M., Adams, B. and Hassan, A. E. (2012), Understanding reuse in the android market, *in* 'Proceedings of the International Conference on Program Comprehension', p. to appear.

Jakarta Regexp (2012), 'Jakarta Regexp', `http://jakarta.apache.org/regexp/`. Last viewed: 20-Oct-2012.

Java Script Object Notation (2012), 'Java Script Object Notation', `http://json.org/`. Last viewed: 20-Oct-2012.

Kampstra, P. (2008), "Beanplot: A boxplot alternative for visual comparison of distributions", *Journal of Statistical Software, Code Snippets* , Vol. 28, pp. 1–9.

Khomh, F., Dhaliwal, T., Zou, Y. and Adams, B. (2012), Do faster releases improve software quality? an empirical case study of mozilla firefox, *in* 'Proceedings of the International Working Conference on Mining Software Repositories'.

Kumar Maji, A., Hao, K., Sultana, S. and Bagchi, S. (2010), Characterizing failures in mobile oses: A case study with android and symbian, *in* 'Proceedings of the International Symposium on Software Reliability Engineering', pp. 249–258.

kXML-RPC (2012), 'kXML-RPC', `http://kxmlrpc.objectweb.org/`. Last viewed: 20-Oct-2012.

Lind, R. and Vairavan, K. (1989), "An experimental investigation of software metrics and their relationship to software development effort", *Transactions on Software Engineering* , Vol. 15, pp. 649–653.

Localytics (2011), 'First impressions matter! 26% of apps downloaded in 2010 were used just once', `http://www.localytics.com/blog/2011/first-impressions-matter-26-percent-of-apps-downloaded-used-just-once/`. Last viewed: 20-Oct-2012.

Lohr, S. (2010), 'Google's Do-It-Yourself App Creation Software', `http://www.nytimes.com/2010/07/12/technology/12google.html`. Last viewed: 20-Oct-2012.

Loukas, A., Woehrle, M. and Langendoen, K. (2011), On mining sensor network software repositories, *in* 'Proceedings of the Workshop on Software Engineering for Sensor Network Applications', pp. 25–30.

Mockus, A., Fielding, R. T. and Herbsleb, J. (2000), A case study of open source software development: the Apache server, *in* 'Proceedings of the International Conference on Software Engineering', pp. 263–272.

Mockus, A., Fielding, R. T. and Herbsleb, J. (2002), "Two case studies of open source software development: Apache and mozilla", *Transactions on Software Engineering and Methodolog* , Vol. 11, pp. 309–346.

Mockus, A. and Votta, L. G. (2000), Identifying reasons for software changes using historic databases, *in* 'Proceedings of the International Conference on Software Maintenance', pp. 120–130.

Murai, J. (2011), 'You Win RIM! (An Open Letter To RIM's Developer Relations)', `http://blog.jamiemurai.com/2011/02/you-win-rim/`. Last viewed: 20-Oct-2012.

Nagappan, N. and Ball, T. (2005), Use of relative code churn measures to predict system defect density, *in* 'Proceedings of the International Conference on Software Engineering', pp. 284–292.

Nguyen, T. N. D., Adams, B. and Hassan, A. E. (2010), Studying the impact of dependency network measures on software quality, *in* 'Proceedings of the International Conference on Software Maintenance', pp. 1–10.

Nielsen Co. (2010*a*), 'Games Dominate America's Growing Appetite for Mobile Apps', `http://blog.nielsen.com/nielsenwire/online/\_\%20mobile/games-dominate-americas-growing-appetite-for-mobile-apps`. Last viewed: 20-Oct-2012.

Nielsen Co. (2010*b*), 'The State of Mobile Apps', `http://blog.nielsen.com/nielsenwire/online/\_mobile/the-state-of-mobile-apps`. Last viewed: 20-Oct-2012.

Nielsen Co. (2010*c*), 'U.S. Smartphone Battle Heats Up: Which is the Most Desired Operating System?', `http://blog.nielsen.com/nielsenwire/online/\_mobile/us-smartphone-battle-heats-up`. Last viewed: 20-Oct-2012.

Nielsen Co. (2011*a*), 'Apple Leads Smartphone Race, while Android Attracts Most Recent Customers', `http://blog.nielsen.com/nielsenwire/online/\_mobile/apple-leads-smartphone-race-while-android-attracts-most-recent-customers`. Last viewed: 20-Oct-2012.

Nielsen Co. (2011*b*), 'Who is Winning the U.S. Smartphone Battle?', `http://blog.nielsen.com/nielsenwire/online/\_mobile/who-is-winning-the-u-s-smartphone-battle`. Last viewed: 20-Oct-2012.

Ostrand, T. J., Weyuker, E. J. and Bell, R. M. (2005), "Predicting the location and number of faults in large software systems", *Transactions on Software Engineering* , Vol. 31, pp. 340–355.

research2guidace (2011), 'The market for mobile application development services', `http://research2guidance.com/shop/index.php/application-developer-market-2010-2015`. Last viewed: 20-Oct-2012.

Schröter, A., Zimmermann, T. and Zeller, A. (2006), Predicting component failures at design time, *in* 'Proceedings of the International Symposium on Empirical Software Engineering', pp. 18–27.

Scitools (2012), 'Understand Your Code', `http://scitools.com/`. Last viewed: 20-Oct-2012.

Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S. and Glezer, C. (2010), "Google Android: A Comprehensive Security Assessment", *Security and Privacy Magazine* , Vol. 8, pp. 35–44.

Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B. and Hassan, A. E. (2010), Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project, *in* 'Proceedings of the International Symposium on Empirical Software Engineering and Measurement', pp. 29–39.

Shihab, E., Kamei, Y. and Bhattacharya, P. (2012), Mining challenge 2012: The android platform, *in* 'Proceedings of the International Working Conference on Mining Software Repositories', p. to appear.

Shihab, E., Mockus, A., Kamei, Y., Adams, B. and Hassan, A. E. (2011), High-impact defects: a study of breakage and surprise defects, *in* 'Proceedings of the ACM SIGSOFT symposium and the European Conference on Foundations of Software Engineering', pp. 300–310.

StatSVN (2012), 'StatSVN - Repository Statistics', `http://statsvn.org/`. Last viewed: 20-Oct-2012.

Teng, C.-C. and Helps, R. (2010), Mobile application development: Essential new directions for IT, *in* 'Proceedings of the International Conference on Information Technology: New Generations', pp. 471–475.

Tracy, K. W. (2012), "Mobile application development experiences on apple's ios and android os", *Potentials* , Vol. 31, pp. 30–34.

W3Techs - Web Technology Surveys (2012), 'Usage Statistics and Market Share of Content Management Systems for Websites', `http://w3techs.com/technologies/overview/content_management/all`. Last viewed: 20-Oct-2012.

Wen, H. (2011), `http://radar.oreilly.com/2011/06/google-app-inventor-programmers-mobile-apps.html`. Last viewed: 20-Oct-2012.

WordPress for Android (2012), 'WordPress for Android', `http://android.wordpress.org/`. Last viewed: 20-Oct-2012.

WordPress for BlackBerry (2012), 'WordPress for BlackBerry', `http://blackberry.wordpress.org/`. Last viewed: 20-Oct-2012.

Workshop on Mobile Software Engineering (2011), 'Workshop on mobile software engineering', `http://mobileseworkshop.org/`. Last viewed: 20-Oct-2012.

Wu, Y., Luo, J. and Luo, L. (2010), Porting mobile web application engine to the android platform, *in* 'Proceedings of the International Conference on Computer and Information Technology', pp. 2157–2161.

Xin, C. (2009), Cross-platform mobile phone game development environment, *in* 'Proceedings of the International Conference on Industrial and Information Systems', pp. 182–184.

Zimmermann, T. and Nagappan, N. (2008), Predicting defects using network analysis on dependency graphs, *in* 'International Conference on Software Engineering', pp. 531–540.

Zimmermann, T., Premraj, R. and Zeller, A. (2007), Predicting defects for eclipse, *in* 'International Workshop on Predictor Models in Software Engineering', p. 9.