Revisiting Prior Empirical Findings For Mobile Apps: An Empirical Case Study on the 15 Most Popular Open-Source Android Apps

Mark D. Syer, Meiyappan Nagappan, Ahmed E. Hassan
Software Analysis and Intelligence Lab
School of Computing, Queens University
{mdsyer, mei, ahmed}@cs.queensu.ca

Bram Adams
Lab on Maintenance, Construction and Intelligence of Software
Génie Informatique et Génie Logiciel, École Polytechnique de Montréal
bram.adams@polymtl.ca

Abstract

Our increasing reliance on mobile devices has led to the explosive development of millions of mobile apps across multiple platforms that are used by millions of people around the world every day. However, most software engineering research is performed on large desktop or server-side software applications (e.g., Eclipse and Apache). Unlike the software applications that we typically study, mobile apps are 1) designed to run on devices with limited, but diverse, resources (e.g., limited screen space and touch interfaces with diverse gestures) and 2) distributed through centralized "app stores," where there is a low barrier to entry and heavy competition. Hence, mobile apps may differ from traditionally studied desktop or server side applications, the extent that existing software development "best practices" may not apply to mobile apps. Therefore, we perform an exploratory study, comparing mobile apps to commonly studied large applications and smaller applications along two dimensions: the size of the code base and the time to fix defects. Finally, we discuss the impact of our findings by identifying a set of unique software engineering challenges posed by mobile apps.

1 Introduction

App stores (e.g., Google Play and Apple App Store) have changed the software development world by providing a platform for the rapid growth of mobile apps (i.e., software applications for smartphones and tablets). Since 2007, mobile apps have exploded into a multi-billion dollar market [1, 2] and become hugely popular among consumers and developers. Mobile app downloads have risen from 7 billion in 2009 to 15 billion in 2010 [2] and are projected to reach 74 billion in 2016 [1]. Simultaneously, the number of mobile apps has also increased: Google Play now hosts over 1 million mobile apps [3].

Despite the ubiquity of mobile devices and the popularity of mobile apps, few software engineering researchers have studied them [4, 5, 6, 7]. During the thirty-five years following Fred Brooks' seminal text, The Mythical Man Month [8], the software engineering community has proposed and evaluated several ideas of how high quality, successful software is developed and maintained. These "software engineering concepts" aim to tie aspects of software artifacts (e.g., size and complexity) [9], their development (e.g., number of changes) [10] and their developers (e.g., developer experience) [11] to definitions of quality (e.g., post-release defects). However, such software engineering concepts have primarily been evaluated against large-scale projects [12].

Understanding how established software engineering concepts apply to mobile apps is an important research question. Mobile apps distinguish themselves from desktop and server applications by their limited functionality, low memory and CPU footprint, limited screen sizes, touch interfaces and access to diverse hardware (e.g., GPS and accelerometers). Hence, mobile apps are expected to become a major challenge for software developers and maintainers in the near future because the underlying hardware and software is constantly and rapidly evolving. In addition, centralized "app stores" (e.g., Google Play) provide consumers with easy access to tens of thousands of apps, and developers with access to millions of consumers. This ecosystem has become characterized by low barriers to entry for prospective developers, but with heavy competition.

As a first step towards understanding how established software engineering concepts apply to mobile apps, we perform an exploratory study to compare mobile apps and desktop/server applications. 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 the following two research questions:

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

- *RQ2: How does the defect fix time compare?* – mobile app developers typically fix defects faster than desktop/server developers, regardless of the size of the project. Defects in mobile apps tend to be concentrated in fewer source code files.

The paper is organized as follows: Section 2 motivates our case study and presents related work. Section 3 describes the setup of our case study and Section 4 describes the results. In Section 5, we present potential directions for future research of mobile apps. Section 6 outlines the threats to the validity of our case study. Finally, Section 7 concludes the paper.

2 Motivation and Related Work

Mobile app downloads have seen explosive growth in the past few years, which is expected to continue well into the future [1]. Despite this growth, few researchers have studied mobile apps from a software engineering perspective. Software engineering concepts like "high churn leads to poor quality" [10] and "high cohesion and low coupling leads to high quality" [9], have primarily been evaluated against large-scale projects such as Apache and Eclipse [12]. Determining how these concepts apply to mobile apps may reduce the effort in developing and maintaining high quality mobile apps.

Our existing software engineering knowledge may not apply to 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.

2.1 Hardware Limitations and Diversity

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 fails to 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 and server systems to mobile devices (mobile devices are intended to be used "on-the-go") may also limit 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 permanent. This is particularly true of the screen size and resolution, which may limit 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.

Therefore, the diversity and limitations of mobile device hardware may limit the scope of mobile apps. Although all software faces hardware limitations, the limitations of mobile devices are generally much greater than those of desktop or server systems.

2.2 Distribution Channel

Major mobile platforms (e.g., the Android platform) 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 [1, 2]. In 2010, more than 50,000 developers were creating mobile apps for the Android or iOS platforms [13]. Further, development tools have been released to the general public so that anyone can develop a mobile app, even without prior development experience. Nearly half of mobile app developers have less than two years of experience [14]. The large and constantly expanding development community leads to high competition between developers. For example, Google Play contains hundreds of competing weather, media and instant messaging apps.

The high competition between developers may affect the quality of mobile apps for fear of losing users to competing apps. Therefore, the emphasis on defect fixing may be high and hence the defect fix times may be low.

2.3 Related Work

Software engineering researchers are now beginning to explore the challenges and issues surrounding mobile apps and platforms [15]. Researchers are also studying mobile apps from other perspectives, including app ecosystems [4, 16], cross platform development [17, 18] and security [19, 20, 21]. However, there are only a few studies of mobile apps from a software engineering perspective.

Mojica et al. studied code reuse in 4,323 Android apps and found that, on average, 61% in the classes in a mobile app are reused by other apps in the same domain [6] (e.g., social networking). The authors also found that 23% of the 534,057 classes in their case study inherit from a class in the Android platform. In this paper, we are interested in how such measures compare across mobile apps and desktop and server applications.

Maji et al. studied defect reports in the Android and Symbian platforms to understand where defects occur in these platforms and how defects are fixed [22]. The authors determine that development tools, web browsers and multimedia modules are the most defect-prone and that most defects require 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 paper, we study Android apps, not the platform itself.

Minelli and Lanza have developed SAMOA, a new tool that can gather and visualize basic source code metrics (e.g., size and complexity) from mobile apps [5]. SAMOA is intended to help developers better understand the development and evolution of their app, whereas the purpose of our work is to empirically establish typical properties of mobile apps and to contrast these properties with traditionally-studied desktop and server applications. Further, we extract and analyze metrics from both the source code and issue tracking systems of both mobile apps and desktop and server applications.

In our previous work, 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 towards understanding the development and maintenance process of mobile apps [7]. We found that BlackBerry apps are much larger and rely more on third party libraries. However, they are less susceptible to platform changes since they rely less on the underlying platform. On the other hand, Android apps tend to concentrate code into fewer large files and rely heavily on the Android platform. On both platforms, we found code churn to be high. However, we are unaware of how these findings on mobile apps compare to desktop or server applications.

3 Case Study Setup

3.1 Mobile App Selection

In this paper, 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 has more free mobile apps than any other major mobile platform [23].

Mobile apps for Android devices are primarily hosted in Google Play [3]. Google Play 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. We used Google Play to list the top 2,000 free apps and the top 2,000 paid apps (2,000 is the maximum number of apps that Google Play ranks). However, we limit our study to free Android mobile apps, because 1) the majority (63%) of Android apps are free [23], 2) free apps are downloaded significantly more than paid mobile apps [24] and 3) the source code repositories and issue tracking systems are not available for paid apps.

Google Play uses the term "free" to describe Android apps that are may be 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 Twitter). 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 demo version with a limited feature set 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.

We selected 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. The FDroid repository is an alternative app store that exclusively lists free and open-source (FOSS) Android apps that are also listed in Google Play [25]. It contains links to the homepage, source code repository and issue tracking system (if available). Since it does not contain any information regarding the user base (e.g., the number of downloads), we must retrieve this information from Google Play.

Mobile apps in the resulting list of filtered apps are 1) popular amongst users (allowing us to study "successful" apps) and 2) open source (allowing us to access source code repositories and issue tracking systems). This list contains twenty mobile apps. However, five were excluded from our case study: three were excluded because they did not have an issue tracking system, one used a different source control system (other than SVN/GIT) and one was a mobile port of a desktop application. We excluded the latter app because the same development team produced the mobile and desktop versions within the same source code repository, meaning that the app was developed largely following a desktop mind set, with minor changes towards the mobile end. This is not typical for a mobile app. Therefore, our case study includes fifteen mobile apps.

Table 1 contains the list of mobile apps that were included in our case study. We have assigned an ID to each mobile app for brevity. Table 1 contains mobile apps from a number of different domains, including utilities, networking, multimedia and games.

Table 1: Selected Mobile Apps

ID	Name	Description
M1	3	Music Player
M2	Apps Organizer	Utility
M3	Barcode Scanner	Utility
M4	ConnectBot	SSH Client
M5	Cool Reader	E-Book Reader
M6	Frozen Bubble	Game
M7	K-9 Mail	Email Client
M8	KeePassDroid	Password Vault
M9	Quick Settings	Utility
M10	Reddit is fun	Social Networking
M11	Scrambled Net	Game
M12	Sipdroid	VOIP client
M13	Solitaire	Game
M14	Tricorder	Game
M15	Wordpress	CMS Client

Table 2: Selected Desktop and Server Applications

ID	Name	Description	First Commit	Last Commit
D1	Apache HTTP Server	HTTP Server	07/03/1996	07/02/2011
D2	Eclipse UI Component	User Interface	05/23/2001	09/12/2011
D3	aspell	Spell Checker	01/01/2000	01/01/2004
D4	joe	Text Editor	04/19/2001	04/19/2005
D5	wget	File Retriever	02/12/1999	01/11/2003

3.2 Desktop/Server Application Selection

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

First, we study two large, commonly studied desktop/server 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 [12], used by many researchers to derive general findings about the software engineering process. We do not claim that these projects are the baseline against which all software engineering research 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 concepts developed from studying desktop/server applications apply to mobile apps.

Second, we also study three 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 and hence available to millions of users). 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 2 contains the list of desktop/server applications that were included in our case study. We have assigned an ID to each desktop/server application for brevity. For each of the selected desktop/server applications, Table 2 contains 1) a brief description and 2) the date of the first and last commits included within our analysis.

3.3 Research Questions

As a first step toward determining how existing software engineering knowledge may be applied to 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?

The purpose of our research questions is to empirically establish properties of mobile apps and to contrast these properties with traditionally-studied software.

4 Case Study Results

RQ1: How does the size of the code base compare?

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 (e.g., lines of code) has been shown to be highly correlated to the complexity of a software application [26, 27]. 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 and of mobile apps and compare it to 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 used the Understand tools by Scitools [28] to extract these metrics. Understand is a toolset of static source code analysis tools for measuring and analyzing software projects. Table 3 presents the total number of source code files and lines of code (LOC) 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 or 2) the date of the last commit for the desktop/server applications in Table 2.

Results

Size of the Code Base

From Table 3, we find that the size of our mobile apps ranges between 2,332 and 47,927 LOC with a median value of 14,014. This is the same order of magnitude as aspell, joe and wget. 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 Eclipse project).

From Table 3, we find that many games tend to be small. For example, M6, M11, M13 and M14 are less than 6,000 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, multiple identities, customizable viewing preferences and alternate themes.

Table 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
D1	386	123,293
D2	2,360	276,980
D3	129	12,311
D4	91	27,419
D5	61	17,376

Since mobile apps tend to have small code bases, we further explore two factors that may influence the size of the code base: 1) the size of the development team and 2) platform usage.

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 constrained by the developers combined effort.

We explored 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 extracted the commit logs from the source code repository and isolated the committer field for each source code commit (i.e.,

a commit that included 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 extracted list of developers contains only unique entries and we merge any uncaught cases (e.g., john_doe@gmail.com and doe.john@google.com). Similar to Mockus et al. [29], 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. Although this definition of core developer and may miss developers that contribute substantially through other means, it is commonly used in practice [29]. Table 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 4, we find that the number of core developers participating in the development and maintenance of a 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 mobile app developers, from 1 to 22 developers, the number of core developers is one in three quarters of the mobile apps. Therefore, the development of mobile apps tends to be driven by a single developer.

The size of the development team was further explored by comparing the size of the code base and the size of the development team. Figure 1 shows the number of developers plotted against the number of lines of code in the code base. Figure 1 also shows a trend line generated by fitting a straight line to the data from our mobile apps.

From Figure 1, we find that the number of developers 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 4, where we find that the average number

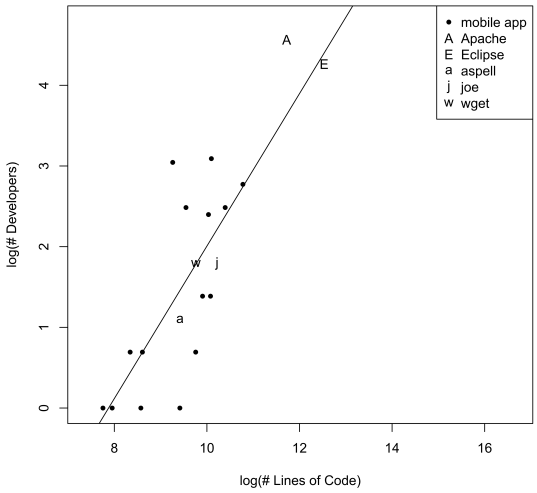

Figure 1: Size of the code base and development team.

of lines of code per developer varies significantly, from 501 to 12,282. Interestingly, the median number of lines of code per developer across the mobile apps (10,524) and the Apache HTTP server and Eclipse UI component (9,977) are very similar.

There are many reasons why mobile apps have varying numbers 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 to 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 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.

Table 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
D1	96	1,284	27	4,566
D2	71	3,901	18	15,388
D3	3	4,104	1	12,311
D4	6	4,570	2	13,710
D5	6	2,896	1	17,376

Table 5: Java and Android Dependencies

ID	Java	Android
M1	20%	39%
M2	13%	16%
M3	19%	13%
M4	21%	14%
M5	33%	11%
M6	19%	20%
M7	14%	17%
M8	38%	7%
M9	8%	42%
M10	64%	17%
M11	10%	30%
M12	13%	11%
M13	4%	12%
M14	7%	25%
M15	68%	18%

We explore the extent of platform usage by extracting the number of references (e.g., 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. These dependencies were classified into one of the following categories based on the class name:

- Android - dependency on a class that is part of the Android platform (e.g., `android.app.Activity`).
- Java - dependency on a class that is part of the Java platform (e.g., `java.io.IOException`).

We then determined platform usage (i.e., the ratio of the number of platform dependencies to the total number of dependencies). Table 5 presents the Platform and Java dependencies as a percentage of the total dependencies.

From Table 5, we find that Java and Android dependence is relatively high in most mobile apps. Further, the median value of the Java and Android dependencies combined is 39% and exceeds 80% in M10 and M15.

From Table 5, we also find that Android dependence is particularly 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 dependence 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 on other systems and consumer usage trends (are consumer demands satisfied by mobile apps with limited functionality?).

We find that mobile apps and unix utilities tend to have smaller code bases than larger desktop/server applications. The development of mobile apps and unix utilities tends to be driven by a one or two core developers. Further, mobile apps tend to rely heavily on an underlying platform.

RQ2: How does the defect fix time compare?

Motivation

Software quality is a major draw for users, and with access to thousands of mobile apps through app stores, 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 explored 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 did not include issues that were classified 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 2 presents these measures for each mobile app and desktop/server application and Table 6 presents the median value of these measures across all 1) mobile apps, 2) large desktop/server applications and 3) unix utilities.

Table 6: Median Percentage of Defects Fixed in One Week/Month/Year

	Week	Month	Year
Mobile Apps	36	68	100
Apache HTTP & Eclipse	33	69	92
Unix Utilities	21	36	80

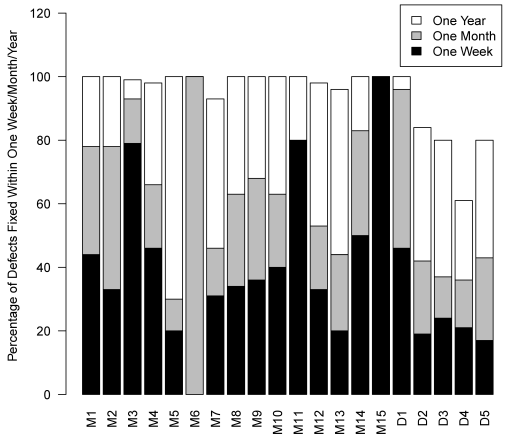

Figure 2: Percentage of Defects Fixed in One Week, One Month and One Year.

Results

Defect Fix Times

From Figure 2, we find that developers of four mobile apps fix 50% of reported defects in one week and developers of eleven 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 is zero because only a single defect was reported in total (it was fixed in 26 days).

From Table 6, we find that the defect fix times in mobile apps are more similar to large desktop/server applications than to 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 Table 6 and Figure 2, we also find that 20% of defects in unix utilities take longer than one year to fix, whereas 100% of defects in mobile apps are fixed within one year.

From Figure 2, 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.

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

Number of Defects Reported

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, 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 did not include issues that were classified 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 7 presents the number of defects reported and the number of users reporting defects in each 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 structure as issue tracking systems. The developers of M7 have specifically asked users to report defects in their issue tracking system.

Table 7: 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
D1	3,425	5,104
D2	1,206	6,287
D3	35	268
D4	26	64
D5	25	55

From Table 7, 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 (media 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 [30]. 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 set of keywords [31, 32]. The keywords (i.e., "bug(s)," "fix(es,ed,ing)," "issue(s)," "defect(s)" and "patch(s)") were developed based on manual analysis 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 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.

Once we have extracted 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 presents the percentage of defects in the top 20% most defect-prone files.

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

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

From Figure 3, 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.

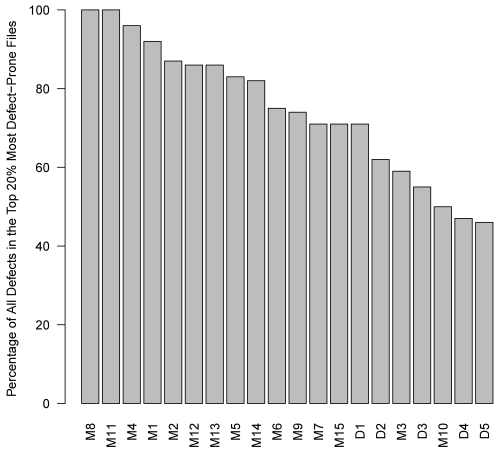

Figure 3: Percentage of All Defects in the Top 20% Most Defect-Prone Files.

Finding the rationale for the relatively quick defect fix times in mobile apps requires more analysis on other systems.

> We find that mobile app developers typically fix defects faster than desktop/server developers, regardless of the size of the project. Defects in mobile apps tend to be concentrated in fewer source code files.

5 Discussion

We identified several potential areas for future research in mobile apps during our current and previous case studies [7]. We believe that the potential impact of such research may be significant given the ubiquity of mobile devices and growing importance of mobile apps. We present two such areas based on observations made during our manual analysis of the source code and development history of mobile apps.

5.1 Platform Usage

We observed that many mobile apps depend highly on their underlying platform (i.e., the Android platform). In our previous work, we defined the "platform dependency ratio" as the ratio of platform API calls to all calls [7]. A low platform dependency ratio indicates that developers do not rely significantly on the platform APIs. For exam-

ple, 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 high platform dependency ratio 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.

5.2 Development Processes

We observed that many mobile apps have a very high frequency of releases. For example, K9Mail typically has two internal releases every week and one release to Google Play every month. Quick release cycles may be required to remain competitive within the marketplace.

Currently, there is evidence 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 market as quickly as possible. For example, as we discussed in Subsection 3.1, 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 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 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 [33]. However, several open questions remain. Does such a high frequency of releases mitigate the lack of testing? If there are frequent releases for the mobile app, then does quality matter as much? Is the project in a constant beta testing state? Does the platform provide 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?

6 Threats to Validity

This section outlines the threats to the validity of our case study.

6.1 Construct Validity

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), however, it is the best measure currently available [4]. Therefore, it is possible that we have mistakenly included mobile apps with small user bases or excluded mobile apps with large user bases from our analysis.

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 and James).

The number of defects in each source code file was measured by identifying the files that were changed in a defect fixing change. Although this technique has been found to be effective [31, 32], 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 [34].

6.2 Internal Validity

The time to fix a defect was calculated by counting the number of days between the date an issue was opened and the date it was closed. It is possible that the defect was fixed quickly, but the issue tracking system was not immediately updated to reflect the fix. In addition, this analysis does not consider defects that were not reported within the issue tracking system.

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).

6.3 External Validity

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). Further, some mobile apps (acting merely as a client) rely on data or services from backend servers, however our results are limited to the apps themselves. Therefore, it is unclear how our results will generalize to 1) closed source mobile apps and desktop/server applications and 2) other mobile platforms.

6.4 Conclusion Validity

Our results may have been affected by confounding factors. One such confounding factor is that the defect reporting mechanisms of Android apps affects the defect fixing process [35]. This may have affected the time to fix defects and the number of defects reported in each mobile app. For example, the authors found that Google Code's bug tracker, which is used by most open-source Android apps, offers less support for management (i.e., triaging) than other widely-used issue tracking systems (e.g, Bugzilla or Jira). This may increase the time it takes to fix defects in mobile apps.

The conclusions of empirical software engineering research may be mistaken as "obvious." However, the goal of empirical research is to scientifically validate whether the "obvious" conclusions are true. We would like to point out that there have been no empirical studies to prove/disprove the claims in this study. Mobile app developers cannot make decisions based on "gut-feeling" instincts. Data-driven empirical studies such as this will provide the necessary scientific foundation for developers to make informed decisions regarding their software.

7 Conclusions

This paper presented an exploratory study to compare mobile apps and desktop/server applications, as a first step toward understanding how the software engineering concepts developed by studying desktop/server will apply to mobile apps. We studied 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 and server applications. Mobile apps and unix utilities are smaller than traditionally studied desktop and server applications (e.g., Apache HTTP server and Eclipse). This is true in terms of the size of the code base and the development team. Further, 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 of reported defects within one month. This is greater for the Eclipse UI component and the aspell, joe and wget utilities, which typically fix only 20% of reported defects in one week and 40% of reported 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 the most defect-prone files is the highest in mobile apps, followed by large desktop and 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 and server applications and unix utilities respectively.

In conclusion, our findings suggest that mobile apps may be facing unique challenges. In order to support the 50,000 developers creating mobile apps [13], researchers should begin to study mobile apps alongside traditionally studied desktop and server applications.

References

[1] L. Columbus, "Roundup of mobile apps and app store forecasts," Jun 2013. [Online]. Available: www.forbes.com /sites/louiscolumbus/2013/06/09/roundup-of-mobile-apps-app-store-forecasts-2013/

[2] C. Sharma, "Sizing up the global apps market," www.chetansharma.com/ mobileappseconomy.htm, Chetan Sharma Consulting, Mar 2013.

[3] "Google play," play.google.com.

[4] M. Harman, Y. Jia, and Y. Z. Test, "App Store Mining and Analysis: MSR for App Stores," in *Proceedings of the International Working Conference on Mining Software Repositories*, Jun 2012.

[5] R. Minelli and M. Lanza, "Software analytics for mobile applications - insights & lessons learned," Mar 2013, pp. 144–153.

[6] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan, "Understanding reuse in the android market," in *Proceedings of the International Conference on Program Comprehension*, Jun 2012, pp. 113–122.

[7] M. D. Syer, B. Adams, A. E. Hassan, and Y. Zou, "Exploring the development of micro-apps: A case study on the blackberry and android platforms," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, Sep 2011.

[8] F. P. Brooks, *The mythical man-month – Essays on Software-Engineering*. Addison-Wesley, 1975.

[9] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Transactions on Software Engineering*, vol. 20, no. 6, pp. 476 –493, Jun 1994.

[10] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the International Conference on Software Engineering*, 2005, pp. 284–292.

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

[12] B. Robinson and P. Francis, "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*, Sep 2010, pp. 21:1–21:10.

[13] N. O'Neill, "10 surprising app platform facts," http://allfacebook.com/app-platform-facts_b18514, All Facebook, Sep 2010.

[14] A. Cravens, "A demographic and business model analysis of today's app developer," http://pro.gigaom.com/archives/research-briefings/, GigaOM, Sep 2012.

[15] "Mobile software engineering," mobileseworkshop.org, Apr 2013.

[16] H.-W. Kim, H. L. Lee, and J. E. Son, "An exploratory study on the determinants of smartphone app purchase," in *Proceedings of the International DSI and the APDSI Joint Meeting*, Jul 2011.

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

[18] C. Xin, "Cross-platform mobile phone game development environment," in *Proceedings of the International Conference on Industrial and Information Systems*, Apr 2009, pp. 182–184.

[19] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *Security and Privacy*, vol. 7, no. 1, pp. 50–57, Jan 2009.

[20] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google android: A comprehensive security assessment," *Security and Privacy*, vol. 8, no. 2, pp. 35–44, Mar 2010.

[21] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, Jun 2012, pp. 281–294.

[22] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *Proceedings of the International Conference on Software Reliability Engineering*, Nov 2010, pp. 249–258.

[23] "Comparisons and Contrasts: Windows Phone 7 Marketplace and Google Android Market," www.distimo.com/publications, Distimo, Jan 2011.

[24] "In-depth view on download volumes in the google android market," www.distimo.com/publications, Distimo, May 2011.

[25] "F-droid," https://f-droid.org/.

[26] R. Lind and K. Vairavan, "An experimental investigation of software metrics and their relationship to software development effort," *Transactions on Software Engineering*, vol. 15, no. 5, pp. 649–653, May 1989.

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

[28] "Understand Your Code," scitools.com.

[29] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the apache server," in *Proceedings of the International Conference on Software Engineering*, Jun 2000, pp. 263–272.

[30] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, apr 2005.

[31] A. E. Hassan, "Automated classification of change messages in open source projects," in *Proceedings of the Symposium on Applied Computing*, Mar 2008, pp. 837–841.

[32] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance*, Oct 2000, pp. 120–130.

[33] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? an empirical case study of mozilla firefox," in *Proceedings of the International Working Conference on Mining Software Repositories*, Jun 2012.

[34] A. E. Hassan, "The road ahead for Mining Software Repositories," in *Frontiers of Software Maintenance*, Oct 2008, pp. 48–57.

[35] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source android apps," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, Mar 2013, pp. 133–143.