Software Qual J (2015) 23:485–508
DOI 10.1007/s11219-014-9238-2

Studying the relationship between source code quality and mobile platform dependence

Mark D. Syer · Meiyappan Nagappan · Bram Adams · Ahmed E. Hassan

Published online: 20 May 2014
© Springer Science+Business Media New York 2014

Abstract The recent meteoric rise in the use of smartphones and other mobile devices has led to a new class of software applications (i.e., mobile apps). One reason for this success is the extensive support available to mobile app developers through the APIs provided by mobile platforms (e.g., Android). In our previous research, we found that mobile apps tend to depend highly on these platform-specific APIs. High dependence on a particular mobile platform may introduce instability and defects, as these mobile platforms are rapidly evolving. Therefore, the extent of platform dependence may be an indicator of software quality. In this paper, we examine the relationship between platform dependence and defect proneness of the source code files of an Android app to determine whether software metrics based on platform dependence can be used to prioritize software quality assurance efforts. We find that (1) source code files that are defect prone have a higher dependence on the platform than defect-free files and (2) increasing the platform dependence increases the likelihood of a defect being present in a source code file. Thus, platform dependence may be used to prioritize the most defect-prone source code files for code reviews and unit testing by the software quality assurance team.

M. D. Syer (✉) · M. Nagappan · A. E. Hassan
Software Analysis and Intelligence Lab (SAIL) School of Computing, Queen's University, Kingston, Canada
e-mail: mdsyer@cs.queensu.ca

M. Nagappan
e-mail: mei@cs.queensu.ca

A. E. Hassan
e-mail: ahmed@cs.queensu.ca

B. Adams
Lab on Maintenance, Construction and Intelligence of Software (MCIS), Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montréal, Montreal, Canada
e-mail: bram.adams@polymtl.ca

1 Introduction

Mobile apps are software applications that run on mobile devices (e.g., smartphones and tablets). Although applications for mobile devices have existed for decades, the development of mobile apps exploded in 2008 when Apple first opened its App Store. Since then, mobile apps have rapidly grown into a multi-billion dollar market (Sharma 2010). The revenues from mobile apps have risen from $4.1 billion in 2009 to $6.5 billion in 2010 (International Data Corp 2011) and are projected to reach $74 billion in 2016 (Sharma 2010).

Mobile devices (e.g., smartphones and tablets) have a diverse set of hardware specifications, such as touch screens, GPS, cameras and accelerometers in comparison with laptops, desktops or servers. These hardware accessories are accessed through APIs provided by the platform (e.g., Android). The platform also provides APIs (1) to access commonly required functionality and (2) interface to the operating system. Such APIs are used by the developers to quickly build mobile apps that exploit the platform and device features.

In our previous work, we found that developers depend heavily on platform APIs to build their mobile apps (Syer et al. 2011). 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; Lohr 2010; Wen 2011; Gavalas and Economou 2011). Two, the rapid succession of mobile technologies and fierce competition among developers forces them to release new features at break-neck speed, without sacrificing quality. Platform APIs provide commonly required functionality that developers may reuse. 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 that will also provide a standard "look and feel" to their mobile apps. 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 dependence on the APIs from 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 (potentially 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.

As dependency metrics have been shown to be highly correlated to defects in source code files (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". Linares-Vásquez et al. (2013) have shown that dependence on fault-prone APIs is significantly lower in highly rated mobile apps (i.e., mobile apps with generally positive customer feedback). However, app ratings are influenced by many factors, such as software quality, cost and privacy concerns (Khalid 2013). Hence, we are interested in determining the specific relationship between software quality (i.e., a software engineering concern that influences app ratings), as measured by the number of source code defects, and platform dependence.

In this paper, we conduct a case study on five Android mobile apps to address the following three research questions:

- *RQ1: Are defect-prone source code files more dependant on the Android platform?* Yes. We find that defect-prone files in all five apps have a statistically significant higher dependence on the Android platform compared to defect-free files.
- *RQ2: Does the extent of platform dependence help explain why some source code files are more defect prone than others?* Yes. We find that the ratio of platform dependencies to the total number of dependencies (i.e., the platform dependency ratio) significantly increases our ability to statistically explain defects in source code files.
- *RQ3: What is the impact of platform dependence on source code quality?* We find that in four out of the five mobile apps, increasing the platform dependency ratio increases the statistical likelihood of defects in source code files.

The findings from this paper indicate that the platform dependency ratio may be used to prioritize the most defect-prone source code files for code reviews and unit testing. This does not necessarily imply that platform dependencies introduce defects, although dependence on defect-prone platform APIs has been shown to reduce app quality (Linares-Vásquez et al. 2013). However, the purpose of this paper is to empirically determine whether the platform dependency ratio may be used to help prioritize software quality assurance efforts.

This paper is organized as follows: Section 2 motivates our case study of the source code quality implications of mobile platform dependence and presents related work in mobile apps and dependency analysis. Section 3 describes the setup of our case study, and Sect. 4 discusses the results of our case study. Section 5 outlines the threats to validity. Finally, Sect. 6 concludes the paper.

2 Motivation and related work

The rise of mobile apps is a relatively recent trend in software engineering. However, software engineering researchers are now beginning to explore the challenges and issues surrounding mobile apps and platforms (Workshop on Mobile Software Engineering 2011). Researchers are also studying mobile apps from other perspectives, including app ecosystems (Harman et al. 2012; Kim et al. 2011), cross platform development and development tools (Wu et al. 2010; Xin 2009; Gasimov et al. 2010; Charland and LeRoux 2011; Tracy 2012) and security (Enck et al. 2009; Shabtai et al. 2010; Grace et al. 2012a, b). However, there are only a few studies of mobile apps from a software engineering perspective.

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

Maji et al. (2010) studied defect reports in the Android and Symbian platforms to understand where defects occur in these platforms and how defects are fixed. 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 (2013) have developed SAMOA, a tool that can gather and visualize basic source code metrics (e.g., size and complexity) from mobile apps. 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 the relationship between static source code metrics and source code quality.

Linares-Vásquez et al. studied app ratings (i.e., customer feedback) in Android apps and found that highly rated mobile apps depend on significantly fewer fault-prone and change-prone APIs than lower-rated apps. However, app ratings are influenced by many factors, such as software quality, cost and privacy concerns (Khalid 2013). Therefore, we are interested in determining the specific relationship between software quality (i.e., source code defects) and platform dependence.

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 toward understanding the development and maintenance process of mobile apps (Syer et al. 2011). 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 the implications of our findings (e.g., high platform dependence) on source code quality.

Software engineering researchers have proposed and evaluated several models of how high-quality, successful software is developed and maintained. These models aim to tie aspects of software artifacts (e.g., size and complexity) (Zimmermann et al. 2007; Chidamber and Kemerer 1994; Shihab et al. 2010), their development (e.g., number of changes) (Nagappan and Ball 2005; Shihab et al. 2010) and their developers (e.g., developer experience) (Bird et al. 2011; Weyuker et al. 2008) to definitions of quality (e.g., post-release defects). However, such models have primarily been evaluated against large-scale projects (Robinson and Francis 2010).

One such model aims to use dependency metrics to enhance the prediction of defects in software systems. Binkley and Schach (1998) proposed a new dependency metric (i.e., the coupling dependency metric) and demonstrated that their metric outperforms existing metrics (e.g., lines of code and complexity) at predicting run-time failures and maintenance effort. Schröter et al. (2006) showed that import dependencies can predict software defects (e.g., importing compiler packages is riskier than importing UI packages). 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. (2010) who found similar results in the Eclipse project. Similarly, we are trying to improve the prediction of defects; however, our focus is on mobile apps rather than large-scale software systems [mobile apps have been shown to differ from such large-scale systems (Syer et al. 2013)].

3 Case study setup

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

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 itself is open-source and has more free and open-source mobile apps than any other major mobile platform (Distimo 2011; Black Duck Software Inc. 2010, 2011, 2012).

Mobile apps for Android devices are primarily hosted in Google Play (formerly the Android Market) (Android Market 2014). 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 does not publish two key metrics (1) the number of cumulative downloads (only very broad ranges are displayed in the store) and (2) the development status (i.e., open source or closed source) with links to the source code repositories. Therefore, we supplement the information provided by Google Play with information from two additional sources:

- *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.
- *AppBrain* A third-party interface to Google Play, to get the number of cumulative downloads (App Brain 2014).

We used 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 2014). 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 2014; App Brain 2014). This limits the number of potential case study subjects to 56.
- *Simplicity* The code base for the mobile app must be easily identified (i.e., contained within its own source code repository). For example, Firefox for Android was excluded because, at the time of our data extraction in on May 1, 2012, we could not differentiate the source code of the mobile version from the desktop version because they share the same source code repository [after performing our analysis, the independent Frennec mobile version of Firefox was released (MozillaWiki 2014)]. 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 modeling, 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 app are defect prone; therefore, we need mobile apps with at least 50 source code files for each source code metric in our regression models (i.e., $20\,\% \times 50 = 10$). As we are including four source code metrics in our regression models (see Sect. 3.2), mobile apps must have at least 200 source code files. This limits the number of potential case study subjects to 5.

Table 1 contains the final list of mobile apps that were included in our case study. Our case study was performed on the source code repository as of May 1, 2012.

Table 1 Mobile apps included in our case study

Project	Description	Homepage
ConnectBot	SSH client	https://github.com/kruton/connectbot/
FBReader	E-book reader	https://github.com/geometer/FBReaderJ
KeePassDroid	Password vault	https://github.com/bpellin/keepassdroid
Sipdroid	VOIP client	http://code.google.com/p/sipdroid/
XBMCRemote	Remote control	http://code.google.com/p/android-xbmcremote/

Table 2 Android SDK version dependencies of the Mobile apps included in our case study

Project	Minimum version	Target version	Maximum version
ConnectBot	3	11	NA
FBReader	5	NA	10
KeePassDroid	3	10	NA
Sipdroid	3	4	NA
XBMCRemote	3	9	NA

NA indicates that the value has not been specified `AndroidManifest.xml` file

Table 2 presents the minimum, target and/or maximum version of the Android SDK (when specified) that is compatible with the app. This information is available in the `AndroidManifest.xml` file that is required in the root directory of every Android app.

3.2 Source code metrics

We used the Understand tool by SciTools (2014) to extract static 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* The total number of lines of code (LOC).
- *Coupling* The number of coupled classes. Class A is said to be coupled to class B if class A uses a type, data, or member from class B.
- *Cohesion* The average cohesion across each class data member. The cohesion of a class data member is defined as the percentage of methods in the class that use that data member. Class A is said to be cohesive if a high percentage of class A's methods use each of class A's variables.

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 extent of dependence between two classes.

In this paper, we studied source code defects in mobile apps that depend on an underlying mobile platform (i.e., the Android platform). Therefore, for each class in a mobile app, we calculated the total number of dependencies on classes in the Android platform (Platform) in addition to the total number of dependencies. These values are used to 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).

We calculated each metric at the file level. We calculated 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 (89 % of the source code files contain only a single-named class.). We calculated 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.

We have selected lines of code, coupling and cohesion for two reasons. First, these metrics have been shown to be good predictors of source code defects (Zimmermann et al. 2007; Nagappan and Ball 2005; Chidamber and Kemerer 1994; Shihab et al. 2010). Second, similar to the platform dependency ratio, these metrics are product metrics (i.e., static source code metrics). We do not include process metrics (e.g., code churn), as process metrics have been shown to be better predictors of source code defects than product metrics (Nagappan and Ball 2005; Shihab et al. 2010). Hence, comparing the platform dependency ratio (a product metric) to process metrics would be an unfair comparison.

> Therefore, four metrics are associated with each source code file (i.e., lines of code, coupling, cohesion and platform dependency ratio).

3.3 Source code quality

Source code quality can be defined and 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 and ignores (1) reported, but not yet fixed, and (2) unreported bugs.

We measure 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. To identify such changes, it is important to realize that 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 2008a; 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 were unable to use these issues because these tend not to include information regarding how the defect was fixed (e.g., the patch or location of the defect). Further, very few commit log messages contain a reference to the issue tracking system (e.g., a defect ID). 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.

4 Case study results

This section presents the results of our case study on the mobile apps selected in Sect. 3.1.

4.1 Preliminary data analysis

Prior to answering our research questions, we perform a preliminary analysis of the data by calculating descriptive statistics. Such a preliminary analysis helps us choose the right pre-processing steps and statistical methods for the analysis in the research questions. Table 3 presents the mean, standard deviation (SD), minimum value (Min) and maximum value (Max), 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 whether transformations are required before the data can be modeled.

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). A positive skew indicates that most values are concentrated to the left of the mean, with extreme values to the right. 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). A positive kurtosis indicates that the peak is higher and sharper and the tails are longer and thicker than the normal distribution. A negative kurtosis

Table 3 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

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 considerably from the normal distribution.

From Table 3 we find that LOC, coupling and Platform have a high ($\geq$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 Fig. 1. Figure 1 presents the distribution of LOC across the source code files of one of the studied mobile apps (i.e., ConnectBot). From Fig. 1 and Table 3, we find that the source code files in the ConnectBot project vary between 4 and 1300 LOC, but there is a peak at around 35 LOC (the left side of Fig. 1). Similar distributions were found in all of our subject mobile apps. Therefore, we log transform LOC, coupling and Platform (i.e., each value X is transformed as $\log(X + 1)$).

From Table 3 we find that cohesion has high (≤ -0.5) negative kurtosis (i.e., peaks are lower and broader and the tails are shorter and thinner than the normal distribution). Similar distributions were found in all of our subject mobile apps. Therefore, we square-root transform cohesion.

In the remainder of this paper, whenever we refer to LOC, coupling, cohesion or Platform, we actually are referring to the transformed values.

In addition to our analysis of the skew and kurtosis of each metric extracted from each project, we also assess the multicollinearity. Multicollinearity may be caused by high correlation between supposedly independent variables. As independent variables become highly correlated, it becomes difficult to distinguish the effect of each independent variable on the dependent variable. Therefore, we use the variance inflation factor (VIF) measure to capture the multicollinearity of each metric extracted from each project. For each project, we iteratively calculate the VIF measure for each metric and remove the metric with the

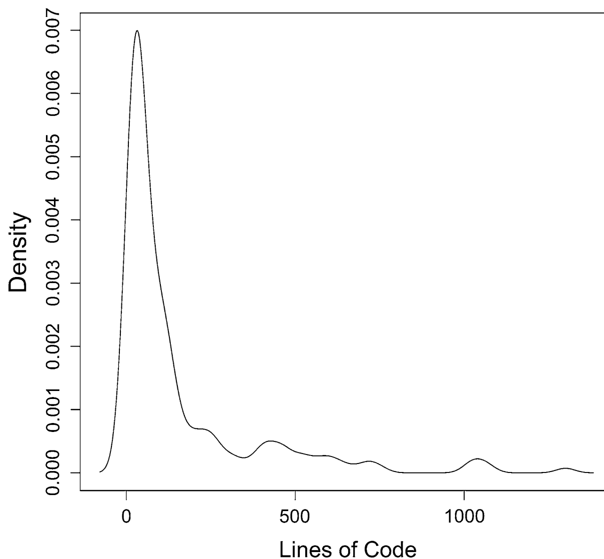


Fig. 1 Distribution of lines of code across the source code files of ConnectBot

Table 4 Multicollinearity analysis

Project	LOC	Coupling	Cohesion	Platform
ConnectBot	6.05	3.08	3.17	1.67
FBReader	2.50	2.55	1.38	1.41
KeePassDroid	4.37	3.87	1.24	1.40
Sipdroid	3.25	2.68	1.52	1.44
XBMCRemote	3.02	3.30	1.14	1.37

highest VIF measure, until no metric has a VIF measure higher than 5 (Fox 2008). Table 4 presents the VIF measure for each metric extracted from each project.

From Table 4, we find that the LOC variable in the ConnectBot project exhibits a VIF measure greater than 5. Therefore, this LOC is excluded from our analysis of the ConnectBot project.

4.2 RQ1: Are defect-prone source code files more dependant on the Android platform?

4.2.1 Motivation

Mobile apps are known to be highly dependent on both the Android and Java platforms (Syer et al. 2011). In our previous work, we defined the "platform dependency ratio" as the ratio of dependencies on the platform to the total number of dependencies (Syer et al. 2011). 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. 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 introduce instability due to the rapid evolution of mobile platforms. For example, the Android platform has undergone a major release every year. If such lock-in does occur, we believe that developers should be aware of the consequences of depending on these platforms.

4.2.2 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 visualized 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, we used a two-sided unpaired *t* test (parametric test) to determine whether the difference between the platform dependency ratios across the defect-free and defect-prone source code files is statistically significant. The two-sided unpaired *t* test is used to compare the population means of two independent populations (e.g., defect-free and defect-prone source code files). *T* tests are one of the most frequently performed statistical

tests (Elliott 2006). We also use a two-sided unpaired Wilcoxon signed-rank test (non-parametric test) to determine whether the difference between the platform dependency ratios across the defect-free and defect-prone source code files is statistically significant. The unpaired 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 (Rice 1995).

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

4.2.3 Results

We visualize the distribution of platform dependency ratios across source code files with and without defects for each mobile app using box plots. Figure 2a–e present these box plots.

From Fig. 2a, c–e, 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 $\geq$ 0), whereas most source code files that are defect-free also have no dependencies on the platform (the log transform of the value 0 is $\log(\text{zero} + 1)$ and, hence, 0).

However, from Fig. 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.

Table 5 presents the results of a two-sided unpaired t test and a two-sided unpaired Wilcoxon signed-rank test performed to determine whether the difference between the distribution of platform dependency ratios across source code files with and without defects, seen in Fig. 2a–e, is 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.

Finally, Table 6 presents the Spearman correlation between (1) the platform dependency ratio and the number of defects and (2) LOC and the number of defects. We have selected LOC for comparison (i.e., a baseline) because it has been shown to be highly correlated with defects and a good predictor of defects (Zimmermann et al. 2007; Nagappan and Ball 2005; Chidamber and Kemerer 1994; Shihab et al. 2010). Indeed, from Table 6, 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, Zimmermann et al. (2007) found a correlation of 0.40 between LOC and defects in the Eclipse project.

From Table 6, 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.

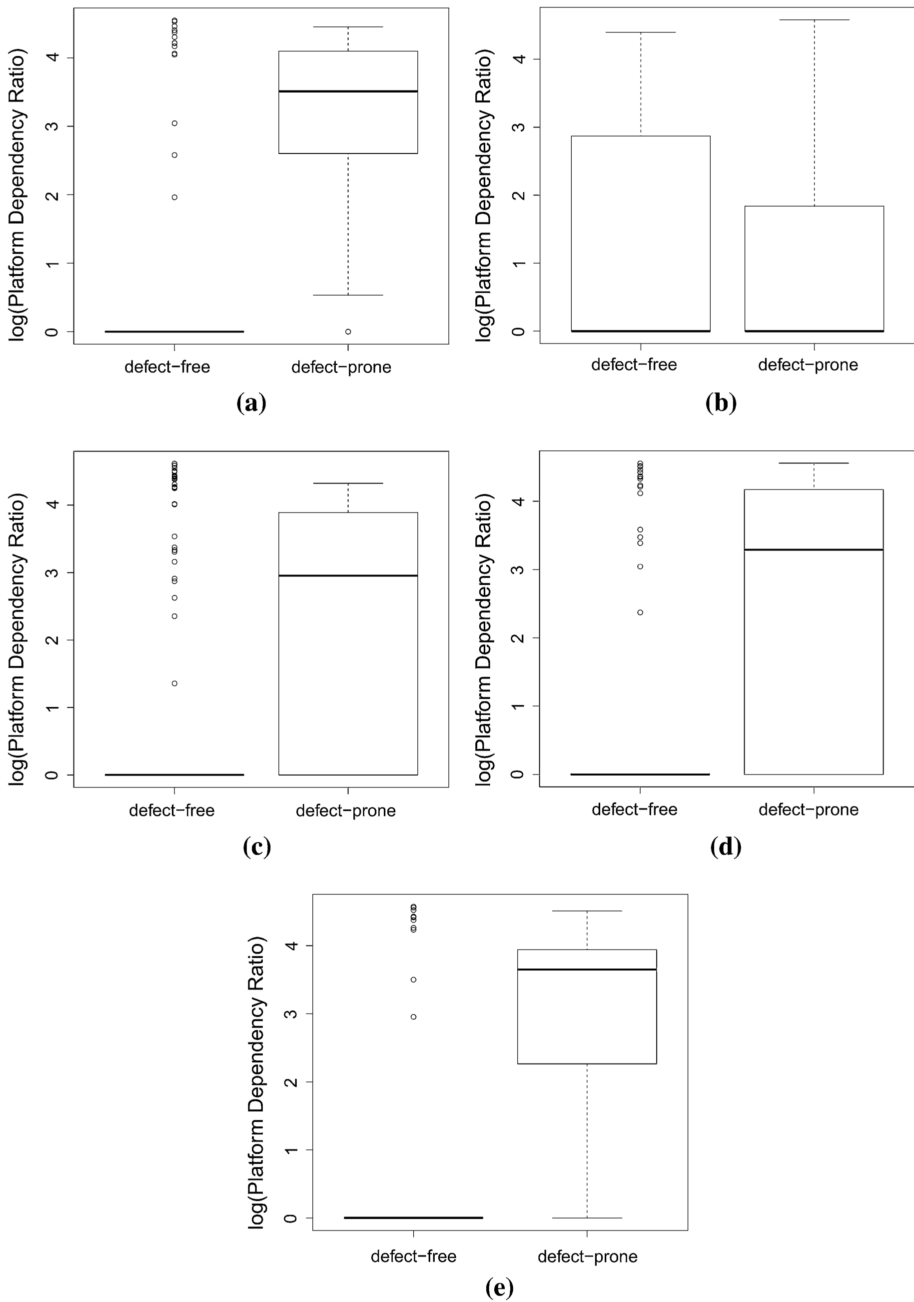


Fig. 2 Distribution of platform dependency ratios across the defect-free and defect-prone source code files. **a** ConnectBot, **b** FBReader, **c** KeePassDroid, **d** Sipdroid, **e** XBMCRemote

The results presented indicate that defect-prone source code files tend to be more dependant on the Android platform than defect-free source code files.

Table 5 *T* tests and Wilcoxon tests

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

Table 6 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

4.2.4 Discussion

It is interesting to note that FBReader is an extreme project in the results above (i.e., unlike ConnectBot, KeePassDroid, Sipdroid and XBMCRemote, the difference between the platform dependency ratios across the defect-free and defect-prone source code files is not statistically significant). In order to examine why this might be the case, we calculate the percentage of source code files that are defect prone in each mobile app and presented in Table 7. From Table 7, 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. Since more files have defects in them, it is likely that even files with no (or low) platform dependency can have defects in them.

4.3 RQ2: Does the extent of platform dependence help explain why some source code files are more defect prone than others?

4.3.1 Motivation

In our previous research question, we found that there is a more than 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

Table 7 Percentage of defect-prone source code files

Project	# Commits	# Source code Files	% Defect prone Source code files
ConnectBot	476	201	20
FBReader	4,685	405	54
KeePassDroid	492	257	19
Sipdroid	622	202	26
XBMCRemote	781	301	40

understanding of defect proneness. In particular, we study whether combining platform dependency ratio with the traditional source code metrics can enhance our ability to explain the defect proneness of source code files, or whether the impact of the dependency ratio can be explained by the other metrics.

4.3.2 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 built two logistic regression models for each mobile app. The first model (Traditional model) was built using traditional metrics [LOC, coupling and cohesion (Nagappan and Ball 2005; Chidamber and Kemerer 1994)]. The second model (Full model) was built using both traditional metrics and the platform dependency ratio.

Finally, we validated these models in two ways.

First, we analyzed 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 >1 (der Meera et al. 2010; Cohen et al. 2002). We removed overly influential observations and rebuilt our models on the new data sets.

Second, we assessed the statistical significance of each coefficient in the new full model (built after removing overly influential observations) to determine which metrics are statistically significant when modeling defects.

We compared 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 and 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 dependent variable in the data set.

4.3.3 Results

We calculated dfbeta residuals for each coefficient in each model. Figure 3 presents these dfbeta residuals for the coefficient modeling coupling in ConnectBot.

From Fig. 3, we find that there are no overly influential observations. Therefore, no observations were removed from the ConnectBot data set. We repeated this procedure for each coefficient in each project. ConnectBot was the only project where we identified any overly influential observations. We then rebuilt our models when one or more observations were removed (i.e., we rebuild on models using a data set with no overly influential observations)

Table 8 presents the coefficients in the full model (i.e., the models built with both traditional metrics and the platform dependency ratio) for each mobile app (recall that these metrics were transformed in Sect. 4). The coefficients in bold are statistically significant ($p \leq 0.05$).

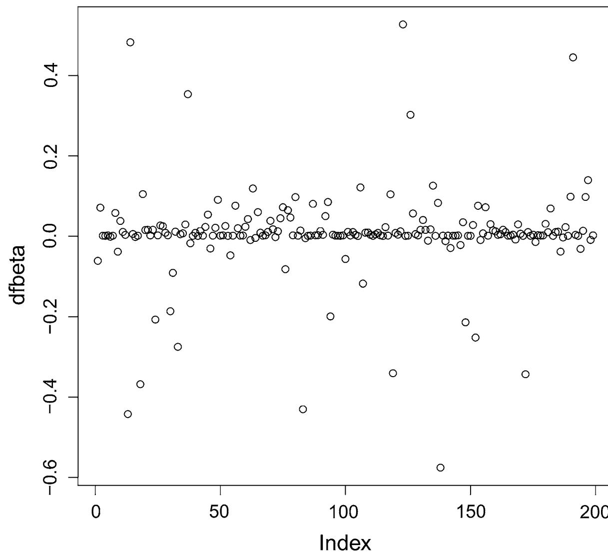


Fig. 3 DFBeta residuals for the coefficient modeling coupling in ConnectBot

Table 8 Coefficients in the full model of each mobile app

Project	(Intercept)	LOC	Coupling	Cohesion	Platform
ConnectBot	**−2.874**	NA	**1.074**	**−0.449**	**1.026**
FBReader	**−3.017**	**0.718**	0.247	0.0330	**−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	−0.0714	**1.084**

From Table 8, 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 (i.e., these metrics tend to be statistically significantly correlated with the presence of defects in mobile apps). Further, LOC is statistically significant in only two mobile apps.

Despite being related measures, coupling and Platform appear to complement each other (i.e., in some projects coupling, but not platform, is statistically significant and vice versa). 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.

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 traditional metrics (i.e., LOC, coupling and cohesion) are statistically significant in KeePassDroid but Platform is not.

From Table 8, we found 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

Table 9 Deviance explained by traditional and full models

Project	Traditional	Full	Difference (%)
ConnectBot	36.45	**59.84**	+64
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	+19

platform dependency ratio. We perform an ANOVA analysis to determine whether 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 9 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).

From Table 9, 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 with respect to the median values). The median increase in the full model over the traditional model is 19 %.

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 be because XBMCRemote has a greater portion of its source code files depending on the Android platform. Table 10 presents the percentage of source code files that depend on the Android platform.

From Table 10, 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 results presented indicate that the platform dependency ratio can help in statistically explaining defects in source code files. Hence, the platform dependency ratio could be used to help prioritize software quality efforts (e.g., code reviews and unit testing).

4.3.4 Discussion

This results of this research question indicate that that the platform dependency ratio can help in statistically explaining defects in source code files. However, the underlying reasons for this relationship remain unclear.

Table 10 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

Platform dependencies may cause defects when the underlying platform is defect prone or the APIs are difficult to use (i.e., the APIs are prone to be called incorrectly). For example, one defect in ConnectBot was caused by a defect in the Android `android.widget.ViewFlipper` API. In a defect fix with the commit log message “Workaround for ViewAnimator bug,” (ViewFlipper inherits from ViewAnimator) the developer implemented his own `ViewFlipper` with the comment “REMOVE THIS CLASS WHEN ViewAnimator IN ANDROID IS FIXED.”

It is also possible that platform dependence is higher in parts of the code base that contain more complex application logic. However, our multicollinearity analysis in Sect. 4 did not identify a high correlation between the platform dependency ratio and coupling, cohesion or lines of code, metrics shown to be highly correlated to the complexity of a software application (Lind and Vairavan 1989; Herraiz et al. 2007).

Regardless of the underlying cause, the platform dependency ratio can help in explaining defects in source code files. Hence, the platform dependency ratio could be used to prioritize software quality efforts (e.g., code reviews and unit testing).

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

4.4.1 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, relative to a baseline, are source code files with twice as many lines of code than expected more defect prone than source code files with twice the number of coupled classes than expected?

4.4.2 Approach

In the previous research question, we built a logistic regression model for each mobile app using both traditional metrics (LOC, coupling and cohesion) and the Platform metric. 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.

We first calculate the average value of each source code metric (i.e., LOC, coupling, cohesion and Platform). Similar to 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 11 shows the hypothetical source code files for FBReader.

Table 11 Hypothetical source code files for FBReader

File	LOC	Coupling	Cohesion	Platform
Baseline	85.00	12.38	66.44	9.71
File1	**93.49**	12.38	66.44	9.71
File2	85.00	**13.61**	66.44	9.71
File3	85.00	12.38	**73.09**	9.71
File4	85.00	12.38	66.44	**10.68**

Table 12 Impact of an increase in each source code metric on defect proneness

Project	LOC (%)	Coupling (%)	Cohesion (%)	Platform (%)
ConnectBot	NA	**23.22**	**−17.61**	**6.141**
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	6.62	15.01	−10.49	3.66

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.

4.4.3 Results

Table 12 presents the change in defect proneness from a 10 % increase in each metric over the baseline (average) values for each mobile app. For example, the second value in the first row, 23.22 %, indicates that increasing coupling by 10 % increases the probability that a source code file is defect prone by 23.22 %. The values in bold in Table 12 correspond to a 10 % increase in a metric that was found to be statistically significant in Table 8.

From Table 12, we find that coupling and cohesion have the greatest impact on defects. "High cohesion and low coupling lead 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 18.75 % respectively, whereas the range for Platform is only 9.55 %.

From Table 12, 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 be 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 3, 10 and 11. Therefore, knowing that a source code file has *any* dependence on the platform may be enough to identify defect-prone source code files.

The results presented indicate that the Platform metric has the most consistent impact on source code quality.

5 Threats to validity

5.1 Threats to construct validity

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

We have performed our analysis on metrics collected at the file level as opposed to the class level. However, in practice, each source code files contains a single-named class. For

example, 89 % of the source code files across the five mobile apps in our case study contain a single-named class.

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 2008a; 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. However, this is a common problem when mining software repositories (Hassan 2008b).

5.2 Threats to internal validity

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

Our results indicate that (1) defect-prone source code files tend to be more dependant on the Android platform than defect-free source code files and (2) increasing the platform dependence increases the likelihood of finding a defect in a source code file. However, the underlying reasons remain unclear because this *correlation* does not necessarily imply *causation*. Regardless of the underlying reason, the platform dependency ratio may be used to prioritize the most defect-prone source code files for code reviews and unit testing by the software quality assurance team.

5.3 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. 2010a, 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, few source code commits and poor documentation. Several open-source mobile apps were excluded based on our selection criteria. 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 [Frennec did not yet exist (MozillaWiki 2014)], and WordPress for Android was excluded because it did not have 200 source code files (we were unable to build a model with any statistically significant coefficients). 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.

6 Conclusions and future work

This paper presented a study of the relationship between platform dependence and defect proneness of the source code files of an Android app. Our study was performed to determine whether software metrics based on platform dependence can be used to prioritize software quality assurance efforts. In particular, we studied (1) whether defect-prone source code files are more dependant on the Android platform than defect-free source code files, (2) whether the platform dependency ratio can help in statistically explaining defects and (3) which source code metrics have the largest impact on source code defects. We addressed these questions by studying five open-source mobile apps written for the Android platform.

We found that (1) defect-prone source code files tend to be more dependant on the Android platform than defect-free source code files and (2) increasing the platform dependence increases the likelihood of finding a defect in a source code file. 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? Is platform dependence coincidentally greater in more complex files? We intend to address these questions in future studies. In the mean time, *developers looking to prioritize their software quality assurance efforts should first examine source code files with the highest platform dependency ratios*.

We also found 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. For example, “larger source code files are more defect prone” was found to hold in only three of our five mobile apps. Hence, focusing on reducing coupling and increasing cohesion seems to be more important from the perspective of software quality of mobile apps than reducing the size.

In the future, we intend to extend our analysis to additional mobile apps and mobile platforms. We 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. We also intend to divide the dependencies based on the type of dependency. For example, inheriting from the API, implementing an API interface or instantiating an API object. Our future studies should help shed light on the nature of the relationship between a mobile app’s dependence on the Android platform and the defect proneness of the mobile app’s source code files.

References

Android Market. (2014). *Android Market*. https://play.google.com/store. Last viewed March 14, 2014.

App Brain. (2014). *App brain*. http://www.appbrain.com/. Last viewed March 14, 2014.

Binkley, A. B., & Schach, S. R. (1998). Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the international conference on software engineering* (pp. 452–455).

Bird, C., Nagappan, N., Murphy, B., Gall, H., & 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 March 14, 2014.

Black Duck Software Inc. (2011). *Mobile innovation, growth driven by open source*. http://blackducksoftware.com/news/releases/2011-03-02. Last viewed March 14, 2014.

Black Duck Software Inc. (2012). Android and enterprise benefit from mobile open source development. http://blackducksoftware.com/news/releases/2012-05-15. Last viewed March 14, 2014.

Butler, M. (2011). Android: Changing the mobile landscape. *IEEE Pervasive Computing*, *10*(1), 4–7.

Charland, A., & LeRoux, B. (2011). Mobile application development: Web vs. native. *Queue*, *9*(4), 20–28.

Chidamber, S., & Kemerer, C. (1994). A metrics suite for object oriented design. *Transactions on Software Engineering*, *20*(6), 476–493.

Cohen, J., Cohen, P., West, S. G., & Aiken, L. S. (2002). *Applied multiple regression/correlation analysis for the behavioral sciences*, 3rd edn. Routledge Academic.

Distimo (2011). *Comparisons and contrasts: Windows phone 7 marketplace and google android market*. http://www.distimo.com/publications. Last viewed March 14, 2014.

Elliott, A. C. (2006). *Statistical analysis quick reference guidebook* (1st ed.). Thousand Oaks, CA: Sage.

Enck, W., Ongtang, M., & McDaniel, P. (2009). Understanding Android Security. *Security and Privacy Magazine*, *7*(1), 50–57.

Fox, J. (2008). *Applied regression analysis and generalized linear models* (2nd ed.). Thousand Oaks, CA: Sage.

Gasimov, A., Tan, C. H., Phang, C. W., & 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., & Economou, D. (2011). Development platforms for mobile applications: Status and trends. *IEEE Software*, *28*(1), 77–86.

Grace, M. C., Zhou, W., Jiang, X., & Sadeghi, A. R. (2012a). 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., & Jiang, X. (2012b). Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the international conference on mobile systems, applications, and services* (pp. 281–294).

Harman, M., Jia, Y., & Test, Y. Z. (2012). App store mining and analysis: MSR for App stores. In *Proceedings of the international working conference on mining software repositories*.

Harrell, F. E., Lee, K. L., Califf, R. M., Pryor, D. B., & Rosati, R. A. (1984). Regression modelling strategies for improved prognostic prediction. *Statistics in Medicine*, *3*(2), 143–152.

Hassan, A. E. (2008a). Automated classification of change messages in open source projects. In *Proceedings of the symposium on applied computing* (pp. 837–841).

Hassan, A. E. (2008b). The road ahead for mining software repositories. In *Frontiers of Software Maintenance* (pp. 48–57).

Hassan, A.E., & 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., & Robles, G. (2007). Towards a theoretical model for software growth. In *Proceedings of the international workshop on mining software repositories* (pp. 21–28).

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 March 14, 2014.

Israel, M. R. J., Nagappan, M., Adams, B., & Hassan, A. E. (2012). Understanding reuse in the android market. In *Proceedings of the international conference on program comprehension* (pp. 113–122).

Khalid, H. (2013). On identifying user complaints of ios apps. In: *Proceedings of the international conference on software engineering* (pp. 1474–1476).

Kim, H. W., Lee, H. L., & Son, J. E. (2011). An exploratory study on the determinants of smartphone app purchase. In: *Proceedings of the international DSI and the APDSI joint meeting*.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Penta, M. D., Oliveto, R., & Poshyvanyk, D. (2013). API change and fault proneness: A threat to the success of Android apps. In *Proceedings of the joint meeting on foundations of software engineering* (pp. 477–487).

Lind, R., & Vairavan, K. (1989). An experimental investigation of software metrics and their relationship to software development effort. *Transactions on Software Engineering*, *15*(5), 649–653.

Lohr, S. (2010). *Google's do-it-yourself app creation software*. http://www.nytimes.com/2010/07/12/technology/12google.html. Last viewed March 14, 2014.

Maji, A. K., Hao, K., Sultana, S., 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).

der Meera, T. V., Grotenhuisb, M. T., & Pelzerb, B. (2010). Influential cases in multilevel modeling: A methodological comment. *American Sociological Review*, *75*(1), 173–178.

Minelli, R., & Lanza, M. (2013). Software analytics for mobile applications—Insights & lessons learned. In *Proceedings of the European conference on software maintenance and reengineering* (pp. 144–153).

Mockus, A., & Votta, L.G. (2000). Identifying reasons for software changes using historic databases. In *Proceedings of the international conference on software maintenance* (pp. 120–130).

MozillaWiki. (2014). *Mobile/fennec/android*. http://wiki.mozilla.org/Mobile/Fennec/Android. Last viewed March 14, 2014.

Nagappan, N., & 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., & 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. (2010a). *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 March 14, 2014.

Nielsen Co. (2010b). *The state of mobile apps*. http://blog.nielsen.com/nielsenwire/online/_mobile/the-state-of-mobile-apps. Last viewed March 14, 2014.

Rice, J. A. (1995). *Mathematical statistics and data analysis* (2nd ed.). North Scituate: Duxbury Press.

Robinson, B., & 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).

Schröter, A., Zimmermann, T., & Zeller, A. (2006). Predicting component failures at design time. In *Proceedings of the international symposium on empirical software engineering* (pp. 18–27).

Scitools. (2014). *Understand your code*. http://scitools.com/. Last viewed March 14, 2014.

Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., & Glezer, C. (2010). Google Android: A comprehensive security assessment. *Security and Privacy Magazine*, *8*(2), 35–44.

Sharma, C. (2010). *Sizing up the global apps market*. http://chetansharma.com/mobileappseconomy.htm. Last viewed March 14, 2014.

Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., & 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., Mockus, A., Kamei, Y., Adams, B., & 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).

Syer, M. D., Adams, B., Hassan, A. E., & Zou, Y. (2011). 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* (pp. 55–64).

Syer, M. D., Nagappan, M., Hassan, A. E., & Adams, B. (2013). Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *Proceedings of the conference of the center for advanced studies on collaborative research* (pp. 283–297).

Tracy, K. W. (2012). Mobile application development experiences on Apple's iOS and Android OS. *Potentials*, *31*(4), 30–34.

Wen, H. (2011). http://radar.oreilly.com/2011/06/google-app-inventor-programmers-mobile-apps.html. Last viewed March 14, 2014.

Weyuker, E., Ostrand, T., & Bell, R. (2008). Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, *13*, 539–559.

Workshop on Mobile Software Engineering. (2011). *Workshop on mobile software engineering*. http://mobileseworkshop.org/. Last viewed March 14, 2014.

Wu, Y., Luo, J., & 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).

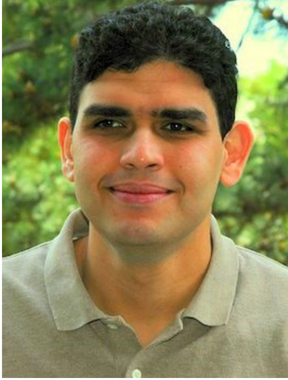
Zimmermann, T, & Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In *International conference on software engineering* (pp. 531–540).

Zimmermann, T., Premraj, R., & Zeller, A. (2007). Predicting defects for eclipse. In *International workshop on predictor models in software engineering* (p. 9).

Mark D. Syer is a PhD student at Queen's University in Canada. He received both his BSc (Engineering Physics) and MSc (Computing) degrees from Queen's University. His research interests include software performance engineering, mobile app development and maintenance, defect prediction and mining software repositories (MSR).

Meiyappan Nagappan received his PhD from NCSU in 2011. He is currently a Post Doctoral Fellow at the SAIL lab in Queen's University. He believes that as SE researchers we should look at deriving solutions that encompass the various stakeholders of software systems, and not only software developers. Hence, for the past 7 years he has been working on SE research that goes beyond just impacting S/W developers and testers. He has worked on using SE research to also address the concerns of S/W operators, build engineers, and project managers.

Bram Adams is an assistant professor at the École Polytechnique de Montréal, Canada, where he heads the MCIS lab on Maintenance, Construction and Intelligence of Software. He obtained his PhD at Ghent University (Belgium), and was a postdoctoral fellow at Queens University (Canada) from October 2008 to December 2011. His research interests include software release engineering in general, and software integration, software build systems, software modularity and software maintenance in particular. His work has been published at premier venues like ICSE, FSE, ASE, ESEM, MSR and ICSM, as well as in major journals like EMSE, JSS and SCP. Bram has coorganized four international workshops, has been tool demo and workshop chair at ICSM and WCRE, and was program chair for the 3rd International Workshop on Empirical Software Engineering in Practice (IWE- SEP 2011) in Nara, Japan. He currently is program co-chair of the ERA-track at the 2013 IEEE International Conference on Software Maintenance (ICSM), as well as of the 2013 InternationalWorking Conference on Source Code Analysis and Manipulation (SCAM), both taking place in Eindhoven, The Netherlands.



Ahmed E. Hassan is the NSERC/RIM Industrial Research Chair in Software Engineering for Ultra Large Scale systems at the School of Computing, Queens University. Dr. Hassan spearheaded the organization and creation of the Mining Software Repositories (MSR) conference and its research community. He co-edited special issues of the IEEE Transactions on Software Engineering and the Journal of Empirical Software Engineering on the MSR topic. Early tools and techniques developed by Dr. Hassans team are already integrated into products used by millions of users worldwide. Dr. Hassan industrial experience includes helping architect the Blackberry wireless platform at RIM, and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. Dr. Hassan is the named inventor of patents at several jurisdictions around the world including the United States, Europe, India, Canada and Japan.