Supporting Evolution and Maintenance of Android Apps

Mario Linares-Vásquez
Computer Science Department - The College of William and Mary
Williamsburg, VA, USA, 23185
http://www.cs.wm.edu/~mlinarev - mlinarev@cs.wm.edu

ABSTRACT

In recent years, the market of mobile software applications (apps) has maintained an impressive upward trajectory. As of today, the market for such devices features over 850K+ apps for Android, and 19 versions of the Android API have been released in 4 years. There is evidence that Android apps are highly dependent on the underlying APIs, and APIs instability (change proneness) and fault-proneness are a threat to the success of those apps. Therefore, the goal of this research is to create an approach that helps developers of Android apps to be better prepared for Android platform updates as well as the updates from third-party libraries that can potentially (and inadvertently) impact their apps with breaking changes and bugs. Thus, we hypothesize that the proposed approach will help developers not only deal with platform and library updates opportunely, but also keep (and increase) the user base by avoiding many of these potential API "update" bugs.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement

Keywords

Mining Software Repositories, Empirical Studies, Android, API changes

1. INTRODUCTION

Since developers often assume correctness behind underlying APIs, bugs in APIs can drastically impact the client code quality as perceived by the end-users. For example, Zibran *et al.* [18] found that among 1,513 bug reports related to various components of Eclipse, GNOME, MySQL, Python 3.1, and Android projects, 562 bug-reports were related to API usability issues; and about 175 (31.1%) of those issues were related to API correctness. Also Businge *et al.* [2] found that 44% of 512 Eclipse third-party plug-ins depends on "bad" (*i.e.*, unstable, discouraged and unsupported) APIs and that developers continue using those APIs. In addition, APIs that do not ensure backward compatibility support are typically hard to use because of their instability (change proneness) [17]. APIs instability requires the adaptation of clients (apps in our case) to avoid the introduction of bugs because of breaking changes[1] (*e.g.*, deletion of a method or a class, introduction of new arguments in a method, changes on the set of exceptions thrown by a method).

Stability and fault-proneness in the Android API is a sensitive and timely topic, given the frequent releases and the number of applications that use these APIs. For example, the change- and fault-proneness of the Android API are a threat to the success of Android apps [8], in the sense that APIs used by successful apps are significantly less change- and fault- prone than APIs used by unsuccessful apps. Moreover, the impact of breaking changes or bugs in the APIs could be a major factor for the development of Android apps because significant versions of Android are released as rapidly as every one to six months. In addition, previous studies by Mojica-Ruiz *et al.* [13, 12] and Syer *et al.* [16] showed that Android apps heavily rely on the Android APIs by using inheritance or API calls.

Regarding the analysis of bug data in the Android platform, few studies have been published. Martie *et al.* [9] analyzed discussions in the Android open source project issue tracker and the main findings are that (i) Android runtime error was a problematic feature of the Android platform and (ii) the new garbage collector in Android Gingerbread may have resolved issues with the Android runtime and graphics applications that use heavy-weight graphics libraries. Assaduzzaman *et al.* [1] mined changes and bug reports in Android to understand the circumstances behind changes that introduced bugs. The links between bugs and changes were identified by looking for keywords in commit messages, and by comparing the textual similarity between the reports and the commit messages. One of the findings is that changes that introduce bugs are larger than average change commits.

Given that release periods of Android APIs are short, it is possible that breaking changes appear in packages and classes that are particularly change-prone; the fast evolution of the APIs could also be explained due to a sheer number of

[1]Changes causing an application built with an older version of the component to fail under a newer version.

needed hot-fixes. Thus, breaking changes and bugs in APIs could impact the quality of Android apps as perceived by the consumers.

We believe that our research program can help Android apps developers to overcome these difficulties. Our main goal is to develop a research program that can provide developers with insights for replicating successful apps recipes or avoiding unsuccessful apps recipes, and create an approach supported by a recommendation system that is able to: (i) warn developers when platform or library changes could impact the quality of the apps because of the impact of breaking changes and bugs; and (ii) synthesize examples of changes that should be applied to adapt the apps to the changes on the underlying APIs.

In this paper, we describe the preliminary research (Section 2) that provided us with some evidence about the nature of Android apps and the impact of Android APIs change- and fault-proneness. For example, based on the results of our preliminary study [8], we believe that identifying changes and bugs on the APIs not only can help solving many API issues in the app, but also help maintain their user ratings. In addition, we describe the proposed research approach (Section 3) and the expected contributions (Section 4).

2. PRELIMINARY RESEARCH

In order to understand and validate whether the APIs can impact the quality of the apps as perceived by the end-users, we conducted two preliminary empirical studies on the reuse in Android apps and the relationship between the success of Android apps and the change/fault proneness of the APIs. Although the amount of reuse in Android apps was studied by Mojica Ruiz *et al.* [13, 12] and Syer *et al.* [16], we wanted to investigate deeply the dependency of Android apps on the platform. In addition, because there was no previous work on the relationship between success of apps and the underlying platform, we analyzed that relationship and published the results [8]. The study by Harman *et al.* [7] is closer to ours, however, they analyzed BlackBerry apps and showed a strong correlation between user ratings and the rank of app downloads, and lack of correlation between price and downloads.

2.1 Reuse in Android Apps

To keep the competitive edge in the apps market, developers have to write apps while keeping track of the entire ecosystem that is constantly changing at a rapid pace (*e.g.*, new hardware capabilities, evolving services and APIs). One of the techniques developers can use to keep the competitive edge is software reuse. However, for mobile apps there were only few studies [13, 12, 16, 7, 10, 11] reporting evidence on how mobile apps are designed or implemented, the amount of reuse in mobile apps, and its implications. Moreover, according to Minelli and Lanza [11], Android apps rely heavily on third-party APIs, and some apps reuse third-party APIs by copying the entire source code of the libraries. This type of reuse, called file-cloning or class cloning, has several implications for software licensing and it is less evident when applications are distributed in compiled packages (*e.g.*, bytecode), as in the case of APK files for Android apps.

To explore deeply the dependency of Android apps on the APIs, we analyzed multiple forms of code reuse across different types of mobiles apps that use Java as the underlying platform. In particular, we analyzed reuse in 28,562 Android apps, Android ported to BlackBerry (Android-BB) apps, and Java Mobile Edition (JME) apps[2]. In addition to the types of reuse analyzed in the studies by Mojica Ruiz *et al.* [13, 12] (*e.g.*, class cloning and inheritance), we measured the amount of reuse by implementation of interfaces, calls to API classes and methods as in [16]. In the case of reuse by class cloning, we relied on the Software Bertillonage technique [5, 4] to identify when a class is cloned across several apps, by comparing the classes' signatures. In the case of reuse by inheritance/implementation, we analyzed bytecodes to identify which API classes/interfaces are extended/ implemented most frequently. We measured API reuse by analyzing which API calls (classes and methods) are being used by each application's files. Finally, we included in the study a reuse mechanism unique to the Android API, namely the Intent object. In other words, we also identified which intents are used and reused most frequently in Android.

Our findings present a whole new perspective on reuse in Java mobile apps, and could be used as a foundation for developing recommendation systems to support developers. We conclude that Java mobile apps depend more on mobile specific APIs (Android and JME) and third-party libraries than on the Java SDK. Regarding the third-party libraries used by Android apps, we found that they play a significant role in the development of mobile applications, especially Google Ads and the JSON standard.

On average, 50% of classes in the analyzed Java mobile apps reuse classes or methods from one of the Java-based APIs (*i.e.*, Android, Java SDK, JME, Third-Party Libraries) for mobile development. Also, on average 15% of the classes in Java mobile apps reuse base classes and design contracts (*i.e.*, interfaces) from one of the Java-based APIs for mobile development; and the average proportion of classes reusing class signatures in Java mobile apps is higher than 50% in the three analyzed platforms.

We also found that the VIEW intent in Android is preferred as a generic multipurpose intent for launching actions than intents designed with a specific purpose (*e.g.*, DIAL). From the official list of possible intents [3], we found that 62 were instantiated and that 44 different action intents were not even used once.

2.2 Fault- and Change-Proneness of Android API

We empirically analyzed and provided a solid empirical evidence on the relationship between the success of 7,097 free Android apps (in terms of user ratings) and the stability and fault-proneness of the used Android API [8]. We focused only on the official Android API, which is predominantly used by Android apps; however, as part of our research plan, we will also consider the third-party APIs used by the apps. We exploited the apps average ratings in Google Play[4] as a measure of their success. To measure fault-proneness, we used the total number of bugs fixed in the used API; for stability (change-proneness), we used the number of changes at method level along three categories: (i) generic changes (including all kinds of changes), (ii) changes applied to method

[2]This study was submitted to the Journal of Empirical Software Engineering and it is still in the reviewing process.
[3]`http://developer.android.com/reference/android/content/Intent.html`
[4]`http://play.google.com`

signatures, and (iii) changes applied to the exceptions thrown by methods.

As of today, we extracted a total of 4,816 API classes used in APK files (*i.e.*, file format used to distribute Android maps), and the bug and change history of those API classes in a period going from September 2009 to January 2013 for a total of 4,781 bug-fixing activities and 370,180 method's changes. To prune out unreliable ratings, we only considered apps having at least n (*e.g.*, n=10) votes. We identified bug-fixing commits activities by mining regular expressions containing issue IDs and the keyword "fix" in the Git commit notes (*e.g.*, "fixed issue #ID" or "issue ID") as in Fischer *et al.* [6]. For identifying the changes, we used a code analyzer developed in the context of the Markos European project (`http://markosproject.berlios.de`) to compare the APIs before and after each commit at a fine-grain level. The code analyzer parses source code and categorizes changes occurring in methods into three types: (i) generic change (including all kinds of changes); (ii) changes applied to the method signature (*i.e.*, visibility change, return type change, parameter added, parameter re- moved, parameter type change, method rename); and (iii) changes applied to the set of exceptions thrown by the methods.

Our findings show that APIs used by successful apps are significantly less fault-prone than APIs used by unsuccessful apps [8]. The results of our study demonstrate that Android apps having higher success generally use APIs that are less fault- and change-prone than apps having lower success. For instance, among 7,097 apps that we analyzed, the 50 least successful apps use APIs that are 500% more fault-prone and 333% more change-prone on average than APIs used by the 50 most successful apps. APIs used by successful apps are also significantly less change-prone than APIs used by unsuccessful apps, including when changes affected method signatures and especially public methods. Instead, changes to the set of exceptions thrown by methods did not significantly relate with the app success.

Finally, a manual analysis of users' comments and API change logs allowed us to find examples providing a qualitative support to such empirical findings. In summary, although it must be clear that the lack of success of an app can depend on several factors, developers should carefully choose the APIs to be used in their apps because fault-prone APIs can in turn cause malfunctions or crashes in apps. Also, API changes may trigger the need for frequent app updates that can in turn introduce new bugs and in general affect the apps' functionality.

3. RESEARCH APPROACH

Our research plan focuses on supporting the evolution and maintenance of Android apps, from the point of view of managing and mitigating the impact of breaking changes and bugs on apps' quality. Therefore, we initially are interested in detecting when a new release of the Android API and third-party libraries represent a risk for the client code (*i.e.*, apps) from the point of view of breaking changes or changes introducing bugs (risky changes). Also, we will identify when a new release has new features that could increase the success of the apps. Subsequently, we are interested in recommending solutions that help developers to mitigate the impact of the risky changes, and recommending examples on how to implement new features provided by the APIs. Both approaches (APIs change analysis and solution recommendation) will be validated by implementing a recommendation system that will be used by Android developers.

3.1 Detecting Changes on the APIs

We consider the following types of API changes as the ones that should be managed and mitigated by Android developers as part of the evolution/maintenance of apps:

Breaking changes: changes to code element such as class/ method deletions, method parameter modifications, and modifications to the set of exceptions thrown by methods. This type of changes will be detected by using the same approach as in our preliminary work [8].

Buggy changes: this includes bugs on the APIs that have not been solved, and bugs that have been introduced by new changes but have not been detected yet. The former set of bugs will be detected by using our approach in [8]; the latter set will be detected by using bug prediction techniques.

Smelly code: although APIs with occurrences of code smells do not represent a direct risk for the apps, warning developers about the existence of smelly code could help them to understand the risk. Code smells will be detected by using a recent approach proposed by Palomba *et al.* [15].

New features: changes represented by new classes or methods added to the APIs (*e.g.*, new Intents). These changes will be detected by using an approach to be built on top of our previous work [8].

3.2 Recommending Actions and Solutions

Given a target APK, the set of APIs (classes and methods from the Android SDK and Third-Party Libraries) used by app will be extracted to detect breaking changes, buggy changes and smelly code on those APIs. In the case of breaking changes, the actions will be notified using predefined patterns for each type of breaking change. For example, if an exception is added to an API method used by the app, a template solution could be "Add a catch block for handling the exception <Exception> to the try-catch block in the method <package.Class.method>".

For buggy changes and smelly code, the recommendations will be presented as a notification describing the possibility of bugs on the APIs used by the app, and the occurrences of code smells.

For new features provided by the APIs, our goal is to provide developers with an approach for visualizing examples of API usages as it was also done on our preliminary work [14]. Moritz *et al.* [14] propose an approach for detecting and visualizing API usage examples by (i) finding similar APIs using Relational Topic Models [3], and (ii) allowing the developer to explore a software space defined by apps using the target API. The latter could be a limitation because the lack of examples of real open source apps using the recently released apps. Therefore, in addition to open source apps in repositories such as F-droid (`https://f-droid.org/`), we will index the apps that are released as examples within the corresponding SDK release, and the APIs test cases.

The recommendations described here present an initial version of our proposed research. More templates and ways for describing the actions/solutions will be explored and defined as part of the research program.

4. CONTRIBUTIONS

Our research program focuses on supporting Android developers to mitigate the negative impact of API changes. Our work has already contributed with solid empirical evidence on the relationship between change- and fault-proneness of the Android API and the success of the apps using that API. We plan to extend this research [8] by analyzing third-party libraries, and other factors that could impact applications success. Second, we will design an approach for identifying different types of changes on the APIs that could introduce bugs on the apps, or could be considered as breaking changes. This approach could be used by the API designers to validate the APIs quality before releasing a new version.

Finally, we will design techniques to automatically identify actions and solutions that could be applied to Android apps as a reaction to API changes. Although our goal is not to automatically adapt apps to the API changes, we believe that our work will contribute to the state-of-the-art of automatic code adaptation by defining templates for the actions that could drive the adaptation process.Therefore, using our approach Android developers will be able to analyze the impact of API changes and bugs, and then react timely in such a way that new releases of the APIs will not affect the quality of the apps and the user ratings.

5. ACKNOWLEDGEMENTS

We would like to thank Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, and Rocco Oliveto for their support on this research. This work is supported in part by the NSF CCF-1016868 and NSF CAREER-1253837 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

6. REFERENCES

[1] M. Assaduzzaman, M. Bullock, C. Roy, and K. Schneider. Bug introducing changes: A case study with Android. In *9th IEEE Working Conference on Mining Software Repositories (MSR'12)*, pages 116–119, 2012.

[2] J. Businge, A. Serebrenik, and M. G. J. van den Brand. Eclipse api usage: the good and the bad. *Software Quality Journal*, 2013.

[3] J. Chang and D. M. Blei. Hierarchical relational models for document networks. *Annals of Applied Statistics*, 2010.

[4] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage determining the provenance of software development artifacts. *Empirical Software Engineering*, 2012.

[5] J. Davies, D. M. German, M. W. Godfrey, and A. J. Hindle. Software bertillonage: Finding the provenance of an entity. In *IEEE Working Conference on Mining Software Repositories (MSR'11)*, 2011.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *19th International Conference on Software Maintenance (ICSM 2003), 22-26 September 2003, Amsterdam, The Netherlands*, pages 23–, 2003.

[7] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *9th IEEE Working Conference on Mining Software Repositories (MSR'12)*, pages 108–112, 2012.

[8] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *9th Joint Meeting of the European Software Engineering Conference and the 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 477–487, 2013.

[9] L. Martie, V. Palepu, H. Sajnani, and C. Lopes. Trendy bugs: Topic trends in the Android bug reports. In *9th IEEE Working Conference on Mining Software Repositories (MSR'12)*, 2012.

[10] R. Minelli. *Software Analytics for Mobile Applications.* Master's thesis, 2012.

[11] R. Minelli and M. Lanza. Software analytics for mobile applications: Insights and lessons learned. In *17th European Conference on Software Maintenance and Reengineering*, page To appear, 2013.

[12] I. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan. A large scale empirical study on software reuse in mobile apps. *IEEE Software Special Issue on Next Generation Mobile Computing*, 2013.

[13] I. Mojica Ruiz, M. Nagappan, B. Adams, and A. Hassan. Understanding reuse in the Android market. In *20th IEEE International Conference on Program Comprehension (ICPC'12)*, pages 113–122, 2012.

[14] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers. Export: Detecting and visualizing api usages in large source code repositories. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, pages 646–651, 2013.

[15] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, pages 268–278, 2013.

[16] D. Syer, B. Adams, Y. Zou, and A. Hassan. Exploring the development of micro-apps: A case study on the blackberry and android platforms. In *11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*, pages 55–64, 2011.

[17] M. Zibran. What makes APIs difficult to use? *International Journal of Computer Science and Network Security (IJCSNS)*, 8(4):255–261, 2008.

[18] M. Zibran, F. Eishita, and C. Roy. Useful, but usable? factors affecting the usability of APIs. In *18th Working Conference on Reverse Engineering (WCRE'11)*, pages 151–155, 2011.