2015 2nd ACM International Conference on Mobile Software Engineering and Systems

Detecting Antipatterns in Android Apps

Geoffrey Hecht[1,2], Romain Rouvoy[1], Naouel Moha[2], Laurence Duchien[1]
[1] University of Lille / Inria, France
[2] Université du Québec à Montréal, Canada
geoffrey.hecht@inria.fr, romain.rouvoy@inria.fr,
moha.naouel@uqam.ca, laurence.duchien@inria.fr

***Abstract*—Mobile apps are becoming complex software systems that must be developed quickly and evolve continuously to fit new user requirements and execution contexts. However, addressing these constraints may result in poor design choices, known as *antipatterns*, which may incidentally degrade software quality and performance. Thus, the automatic detection of antipatterns is an important activity that eases both maintenance and evolution tasks. Moreover, it guides developers to refactor their applications and thus, to improve their quality. While antipatterns are well-known in object-oriented applications, their study in mobile applications is still in their infancy. In this paper, we propose a tooled approach, called PAPRIKA, to analyze Android applications and to detect object-oriented and Android-specific antipatterns from binaries of mobile apps. We validate the effectiveness of our approach on a set of popular mobile apps downloaded from the Google Play Store.**

I. INTRODUCTION

Along the last decade, the development of mobile applications (apps) has reached a great success [1]. This success is partly due to the adoption of established *Object-Oriented* (OO) programming languages, such as Java, Objective-C or C#, to develop these mobile apps. However, the development of a mobile app differs from a standard one since it is necessary to consider the specificities of mobile platforms. Additionally, mobile apps tend to be smaller software, which rely more heavily on external libraries and reuse of classes [8].

In this context, the presence of common software antipatterns can be imposed by the underlying frameworks [7]. Software antipatterns are bad solutions to known design issues and they correspond to defects related to the degradation of the architectural properties of a software system [4]. Moreover, antipatterns tend to hinder the maintenance and evolution tasks, not only contributing to the technical debts, but also incurring additional costs of development. Furthermore, in the case of mobile apps, the presence of antipatterns may lead to resource leaks (CPU, memory, battery, etc.) [5], thus preventing the deployment of sustainable solutions. The automatic detection of such software anti-patterns is therefore becoming a key challenge to assess the quality, ease the maintenance and the evolution of these mobile apps, which are invading our daily lives. However, the existing tools to detect such software antipatterns are limited and are still in their infancy, at best [11].

II. BACKGROUND AND RELATED WORK

Android apps are distributed using the APK file format. They contains a `.dex` file with the compiled app classes and code in *Dex* file format. While Android apps are developed using the Java language, they use the Dalvik Virtual Machine as a runtime environment. The Dalvik Virtual Machine is register-based, in order to be memory efficient compared to the stack-based JVM [2]. The resulting bytecode is therefore different. Disassembler exists for the Dex format and tools to transform the bytecode into intermediate languages or even Java are numerous. However, there is an important loss of information during this transformation for existing approaches [3]. It is also important to note that around 30% of all the apps distributed on Google Play Store are obfuscated [12] to prevent reverse-engineering.

Verloop [11] used popular Java refactoring tools to detect code smells, like *large classes* or *long methods* in open-source software. They found that antipatterns tend to appear at different frequencies in classes that inherit from the Android framework (called core classes) compare to classes which are not (called non-core classes). They did not considered Android-specific antipatterns in both of these studies. The detection and the specification of mobile-specific antipatterns are still considered as open issues. Reimann *et al.* [9] propose a catalog of 30 quality smells dedicated to Android. These code smells are mainly originated from the good and bad practices documented online in Android documentations or by developers reporting their experience on blogs. They are reported to have a negative impact on properties, such as efficiency, user experience or security. Reimann *et al.* are also offering the detection and correction of some code smells via the REFACTORY tool [10] based on EMF models.

III. PAPRIKA APPROACH

PAPRIKA builds on a three-steps approach, which is summarized in Figure 1. As a first step, PAPRIKA parses the APK file of the application under analysis to extract some application metadata (*e.g.*, application name, package) and a representation of the code in terms of entities like `Class`, `Method` or `Attributes`. Additional metadata (*e.g.*, rating, number of downloads) are also extracted from the Google Play Store and passed as arguments. This representation is then automatically visited to compute a model of the code (including classes, methods, attributes) as a graph annotated with a set of raw quality metrics. PAPRIKA supports two kinds of metrics: *OO* such as Cyclomatic Complexity, Lack Of Cohesion In Methods; and *Android-specific* metrics like Number Of Activities or Number Of Services. PAPRIKA uses the SOOT frameworkand its DEXPLER module [3] to decompile the bytecode. This phase also works when the code is obfuscated by using some bypass strategies. As a second step, this model is stored into a graph database since we aim at providing a scalable solution to analyze mobile applications at large scale. Finally, the third step consists in querying

978-0-7695-5566-9/15 $31.00 © 2015 IEEE
DOI 10.1109/MobileSoft.2015.38

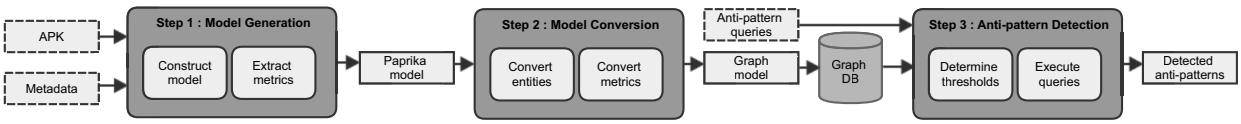

Fig. 1. Overview of the PAPRIKA approach to detect software antipatterns in mobile apps.

the graph to detect the presence of common antipatterns in the code of the analyzed applications. Currently, PAPRIKA supports 8 antipatterns, including 4 Android-specific antipatterns. The OO antipatterns are *Blob class* (blob) [4], *Swiss Army Knife* (SAK) [4], *Complex Class* (CC) [6] and *Long Method* (LM) [6]. The usage of *Internal Getter/Setter* (IGS) is an Android antipattern, their usage is not recommended because on Android fields should be accessed directly within a class to avoid virtual invoke and increase performance [5]. *No Low Memory Resolver* (NLMR) appears when the method `onLowMemory()` is not implemented by an Android activity. This method is called to trim the memory of the application when the system is running low on memory. If not implemented, the Android system may kill the process in order to free memory, which can cause an abnormal termination of programs [5]. Another Android antipattern is the *Member Ignoring Method* (MIM) because on Android, when a method does not access an object attribute, it is recommended to use a static method in order to optimize performance [5]. The last Android antipattern is the *Leaking Inner Class* (LIC) which appears when an inner class is not static. Because in Java, non-static inner and anonymous classes are holding a reference to the outer class. This could provoke a memory leak on Android [5].

IV. RESULTS

To validate and calibrate our approach, we first developed a witness mobile app implementing several instances of each type of antipatterns. We showed that we are able to detect the presence of antipatterns by analyzing the bytecode and despite code obfuscation. The results of our analysis on 15 popular applications (incl. Facebook, Skype, Twitter) are available online[1]. The antipatterns are grouped by the type of entities concerned, either classes or methods or entities. The integer value represents the number of occurrences of the antipatterns in the mobile app. The percentage is the ratio of this value with regard to the total number of classes, methods or activities in the mobile app. Each antipattern is attached to one instance of the entity type, but it should be noted that a same entity can be affected by more than one antipattern. Our results bring us three major findings. First, we found OO antipatterns in all analyzed applications. Overall, OO antipatterns are common in Android apps and they are as frequent as in non-mobile applications, except for SAK. However, a particularity exists for mobile applications: the *Activity* class of the Android framework tends to be more sensitive to Blob than other classes. Secondly, we found Android-specific antipatterns in all analyzed applications. They are really common and frequent, despite the fact that they are easy to refactor. And finally, Android-specific antipatterns are far more frequent and common than OO antipatterns. The three more frequent antipatterns (NLMR, MIM and LIC) are known to affect the efficiency of applications and thus, a refactoring focusing on the concerned classes and methods could improve the application performance without any trade-off. Also, one should note that the results on those antipatterns may greatly vary between applications, for example around 93% of Adobe Reader activities do not implement a method `onLowMemory()` whereas it is only around 15% for Twitter. Therefore, we assume that the developers practices are the root cause to their presence. These antipatterns have been recently defined for Android and thus they are not yet really well referenced by the literature and developer's documentations, which may be a cause of their strong presence. Therefore, we assume that the developers' practices are the root cause to their presence. We plan to implement the detection of more antipatterns and to perform our analysis on a larger set.

[1]Detailed results: http://goo.gl/K77H00

REFERENCES

[1] Android will account for 58commanding a market share of 75 https://www.abiresearch.com/press/android-will-account-for-58-of-smartphone-app-down. [Online; accessed November-2014].

[2] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. [Online; accessed November-2014].

[3] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[4] W. J. Brown, H. W. McCormick, T. J. Mowbray, and R. C. Malveau. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley New York, 1. auflage edition, 1998.

[5] M. Brylski. Android smells catalogue. http://www.modelrefactoring.org/smell_catalog, 2013. [Online; accessed November-2014].

[6] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[7] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *Proc. of the 22nd International Conference on Program Comprehension*, pages 232–243. ACM, 2014.

[8] R. Minelli and M. Lanza. Software analytics for mobile applications–insights and lessons learned. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 144–153. IEEE, 2013.

[9] J. Reimann, M. Brylski, and U. Amann. A Tool-Supported Quality Smell Catalogue For Android Developers. In *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung MMSM 2014*, 2014.

[10] J. Reimann, M. Seifert, and U. Aßmann. On the reuse and recommendation of model refactoring specifications. *Software & Systems Modeling*, 12(3):579–596, 2013.

[11] D. Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013.

[12] L. Xu. *Techniques and Tools for Analyzing and Understanding Android Applications*. PhD thesis, University of California Davis, 2013.