2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

An Approach to Detect Android Antipatterns

Geoffrey Hecht
University of Lille 1 / Inria, France / Université du Québec à Montréal, Canada
geoffrey.hecht@inria.fr

***Abstract*—Mobile applications are becoming complex software systems that must be developed quickly and evolve regularly to fit new user requirements and execution contexts. However, addressing these constraints may result in poor design choices, known as antipatterns, which may degrade software quality and performance. Thus, the automatic detection of antipatterns is an important activity that eases the future maintenance and evolution tasks. Moreover, it helps 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 presents a tooled approach, called PAPRIKA, to analyze Android applications and to detect object-oriented and Android-specific antipatterns from binaries of applications.**

I. Problem and motivation

Along the last decade, the development of mobile applications (apps) has reached a great success. In 2013, Google Play Store[1] reached over 50 billion app downloads [2] and is estimated to reach 200 billion by 2017 [5]. 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 [15], [18], [22].

In this context, the presence of common software antipatterns can be imposed by the underlying frameworks [13], [21]. 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 [9]. 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, battery, etc.) [10], 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 [21].

Mobile apps are mainly distributed by app stores, such as Apple Store, Google Play Store or Windows Phone Store, which do not provide access to their source code [15]. Therefore, it is necessary to analyze non open-source apps to acquire substantial and significant knowledge and data concerning the presence of antipatterns in most of the mobile apps.

[1]https://play.google.com/store

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 [3]. While Android apps are developed using the Java language, they use the Dalvik Virtual Machine as a runtime environment. The resulting bytecode is therefore different from the Java Runtime Environment [3]. Disassembler exists for the Dex format [7] and tools to transform the bytecode into intermediate languages or even Java are numerous [8], [11], [19]. However, there is an important loss of information during this transformation for existing approaches [8]. Some dependencies are also absent from the Dex files, resulting in phantom classes, which cannot be analyzed without the source code. It is also important to note that around 30% of all the apps distributed on Google Play are obfuscated [22] in order to prevent reverse-engineering.

Linares-Vásquez *et al.* [13] used the tool DECOR to perform the detection of 18 different OO antipatterns in mobile apps built using *Java Mobile Edition* (J2ME) [4]. This large-scale study shows that the presence of antipatterns negatively impacts the software quality metrics, in particular metrics related to fault-proneness. Concerning Android, Verloop [21] used popular Java refactoring tools, such as PMD [6] 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 compare to classes which are not. 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.* [16] 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 code smells via the REFACTORY tool [17]. This tool can detect the code smells from an EMF model. However, we have not been yet able to execute this tool on an Android app. Moreover, there is no evidence that all the antipatterns of the catalog are detectable using this approach.

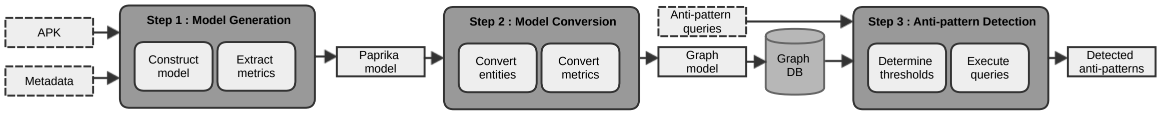

Fig. 1. Overview of the PAPRIKA approach to detect software antipatterns in mobile apps.

III. APPROACH AND UNIQUENESS

PAPRIKA builds on a three-steps approach, which is summarized in Figure 1. As a first step, PAPRIKA parses the APK file of the app under analysis to extract some app metadata (*e.g.*, app 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 as a graph annotated with a set of raw quality metrics. PAPRIKA supports two kinds of metrics: *OO* such as Cyclomatic Complexity or Depth Of Inheritance; and *Android-specific* metrics like Number Of Activities or Number Of Services. PAPRIKA uses the SOOT framework [20] and its DEXPLER module [8] to decompile the bytecode. This phase also works when the code is obfuscated by using some bypass strategies. For instance, to determine the presence of a getter/setter we do not analyze the method names, but we rather focus on the number and the types of instructions as well as the variable accessed by the method. As a second step, this model is stored into a graph database since we aim at providing a scalable solution to analyze mobile apps at large scale. Finally, the third step consists in querying the graph to detect the presence of common antipatterns in the code of the analyzed apps. Entity nodes which implement antipatterns are returned as results for analyzed apps. Currently, PAPRIKA supports 8 antipatterns, including 4 Android-specific antipatterns. The OO antipatterns are *Blob class* (blob) [9], *Swiss Army Knife* (SAK) [9], *Complex Class* (CC) [12] and *Long Method* (LM) [12]. The usage of *Internal Getter/Setter* (IGS) is an Android antipattern, their usage is not recommended because fields should be accessed directly within a class to avoid virtual invoke and increase performance [1], [10]. *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 app 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 [10]. Another Android antipattern is the *Member Ignoring Method* (MIM),on Android when a method does not access an object attribute, it is recommended to use a static method in order to optimize performance [1], [10]. 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 [10], [14].

IV. RESULTS AND CONTRIBUTIONS

We validated our approach on a witness app and showed that we are able to detect the presence of antipatterns by analyzing the bytecode with high precision (0.989) and recall (1). The analysis of bytecode is efficient even when code obfuscation is used to prevent reverse-engineering. The results of our analysis on 15 popular apps (incl. Facebook, Skype, Twitter, Adobe Reader) are presented in Table I. The integer value represents the number of occurrences of the antipatterns in the app. The percentage is the ratio of this value with regard to the total number of entities in the app. Our results bring us three major findings. First, we found OO antipatterns in all analyzed apps. Overall, the OO antipatterns are as common in Android apps as in non-mobile apps at the exception of the rarer SAK. However, a particularity exists : the *Activity* class of Android tends to be more sensitive to Blob. Secondly, we found Android-specific antipatterns in all analyzed apps. 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 apps and thus, a refactoring focusing on the concerned classes and methods could improve the app performance without any trade-off. However, the results may greatly vary between apps, for example around 93% of Adobe Reader activities do not implement a method `onLowMemory` whereas it is only around 15% for Twitter. These antipatterns have been recently defined for Android and thus they are not yet well referenced by the literature, 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 on a larger set.

TABLE I
PAPRIKA RESULTS FOR THE DETECTION OF 8 ANTIPATTERNS IN 15 APPS.

	Class			
	BLOB (OO)	SAK (OO)	CC (OO)	LIC (Android)
Total	711	157	2,367	**7,509**
Ratio	3.55%	0.78%	11.83%	**37.52%**

	Method			Activity
	LM (OO)	IGS (Android)	MIM (Android)	NLMR (Android)
Total	12,592	503	**22,997**	**122**
Ratio	10.41%	0.42%	**19.02%**	**39.23%**

References

[1] Android performance tips. http://developer.android.com/training/articles/perf-tips.html. [Online; accessed November-2014].

[2] 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].

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

[4] Java platform, micro edition (java me). http://www.oracle.com/technetwork/java/embedded/javame/index.html. [Online; accessed November-2014].

[5] Mobile applications futures 2013-2017. http://www.portioresearch.com/en/mobile-industry-reports/mobile-industry-research-reports/mobile-applications-futures-2013-2017.aspx. [Online; accessed November-2014].

[6] Pmd. http://pmd.sourceforge.net/. [Online; accessed November-2014].

[7] Smali: An assembler/disassembler for android's dex format. https://code.google.com/p/smali. [Online; accessed November-2014].

[8] 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.

[9] 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.

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

[11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

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

[13] 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.

[14] A. Lockwood. How to leak a context: Handlers and inner classes. http://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html, 2013. [Online; accessed November-2014].

[15] 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.

[16] 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.

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

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

[19] M. Schönefeld. Reconstructing dalvik applications. In *10th annual CanSecWest conference*, 2009.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proc. of the conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

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

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