Detecting Antipatterns in Android Apps

Geoffrey Hecht, Romain Rouvoy, Naouel Moha, Laurence Duchien

Detecting Antipatterns in Android Apps

Geoffrey Hecht, Romain Rouvoy, Naouel Moha, Laurence Duchien

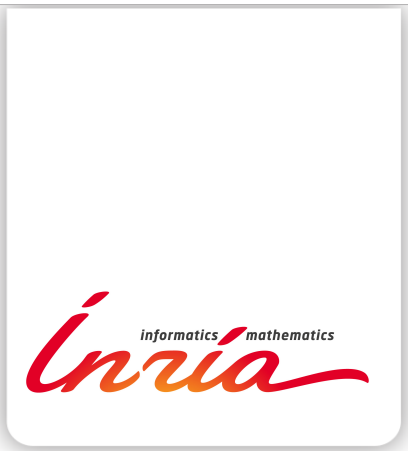
RESEARCH REPORT

N° 8693

March 2015

Project-Teams Spirals

ISSN 0249-6399 ISRN INRIA/RR--8693--FR+ENG



Detecting Antipatterns in Android Apps

Geoffrey Hecht, Romain Rouvoy, Naouel Moha, Laurence Duchien

Project-Teams Spirals

Research Report n° 8693 — March 2015 — 20 pages

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.

Key-words: Android, antipattern, mobile app, software quality

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Detecting Antipatterns in Android Apps

Résumé : 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.

Mots-clés : Android, anti-patron, application mobile, qualité logicielle

Contents

1 Introduction

Along the last years, the development of mobile applications (apps) has reached a great success. In 2013, Google Play Store[1] reached over 50 billion app downloads [3] and is estimated to reach 200 billion by 2017 [6]. 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 applications, which rely more heavily on external libraries and reuse of classes [28, 32, 40].

In this context, the presence of common software anti-patterns can be imposed by the underlying frameworks [24, 38]. 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 [16]. 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.) [17], thus preventing the deployment of sustainable solutions. The automatic detection of such software antipatterns 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 [38].

Mobile apps are mainly distributed through app stores, such as Apple Store, Google Play Store or Windows Phone Store, which do not provide access to their source code [28]. Catalogs of open-source applications are available online[2], but there is no evidence that the hosted apps are representative of the ones distributed by official app stores. 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.

We thus aim at improving the quality of mobile apps by mining legacy apps. This paper focuses on the analysis of official mobile apps to detect the presence of software antipatterns. In particular, we introduce Paprika as a tooled approach to analyze the code of Android apps and to detect both common OO and Android-specific antipatterns. Paprika innovates by analyzing the Android package of mobile apps. To the best of our knowledge, this is the first approach to detect common software antipatterns from this package. Paprika extracts quality metrics from application bytecode that are stored persistently. The resulting model can be further queried by Paprika to detect the presence of software antipatterns. One example of a detected OO antipattern is the *Blob class*, which is a class with a low cohesion between methods and a large number of attributes and operations [16]. As another example, the *Internal Getter/Setter* is an Android-specific antipattern known to degrade the performance on this system [1, 17]. It occurs when a method calls a getter or a setter of its own class. We assess our approach by reporting on the results we obtained for the detection of 8 antipatterns on 15 popular Android apps downloaded from the Google Play Store. In this study, we address the following 4 research questions:

RQ$_1$: *Can antipatterns originating from the source code be detected at the bytecode level?*
Finding: Yes, by analyzing the bytecode we are able to detect the presence of antipatterns. The analysis of bytecode is efficient even when code obfuscation is used to prevent reverse-engineering.

RQ$_2$: *To what extent mobile apps exhibit OO antipatterns?*
Finding: We found OO antipatterns in all analyzed apps. Overall, the OO antipatterns are as common in Android apps as in non-mobile applications. However, a particularity exists for mobile apps: the *Activity* class of the Android Framework tends to be more sensitive than other classes.

[1] `https://play.google.com/store`
[2] `https://f-droid.org`

RQ$_3$: *To what extent mobile apps exhibit Android-specific antipatterns?*
Finding: We found Android-specific antipatterns in all analyzed apps. They are really common and frequent, despite the fact that they are easy to refactor.
RQ$_4$: *Are OO and Android-specific antipatterns present in the same proportion?*
Finding: The Android-specific antipatterns are far more frequent and common than OO antipatterns.

The rest of the paper is organized as follows. We provide some background on Android package and bytecode in Section 2. We compare to the related works in Section 3. The details of our framework are introduced in Section 4. We validate our approach empirically on 15 applications in Section 5. Section 6 summarizes our work and outlines some avenues for future works.

2 Background on Android Package and Bytecode

This section provides a short overview of the specificities of *Android Application Package* (APK) and Dalvik bytecode.

Android apps are distributed using the APK file format. APK files are archive files in a ZIP format, which are organized as follows: 1. the file `AndroidManifest.xml` describes application metadata including *name*, *version*, *permissions* and *referenced library files* of the application, 2. the directory `META-INF` that contains meta-data and certificate information, 3. an `asset` and a `res` directory containing non-compiled resources, 4. a `lib` directory for eventual native code used as library, 5. a `resources.arsc` file for pre-compiled resources, and 6. a `.dex` file containing the compiled application classes and code in *dex* file format [4]. While Android apps are developed using the Java language, they use the Dalvik Virtual Machine as a runtime environment. The main difference between the *Java Virtual Machine* (JVM) and the Dalvik Virtual Machine is that Dalvik is register-based, in order to be memory efficient compared to the stack-based JVM [4]. The resulting bytecode compiled from Java sources and interpreted by the Dalvik Virtual Machine is therefore different.

Disassembler exists for the Dex format [8] and tools to transform the bytecode into intermediate languages or even Java are numerous [11, 14, 19, 33]. However, there is an important loss of information during this transformation for all the existing approaches. For instance, additional algorithms have to be used to infer the type of local variables or to determine the type of branches as `for`, `while` and `if` constructions are replaced by `goto` instructions in the bytecode [4, 14]. Some dependencies are also absent from the Dex files, resulting in phantom classes, which cannot be analyzed without the source code. And, of course, the native code included in the `lib` directory cannot be decompiled with these tools. It is also important to note that around 30% of all the mobile apps distributed on Google Store are obfuscated [40] in order to prevent reverse-engineering. The ProGuard tool used to obfuscated code is even pre-installed on the beta of Android Studio provided by Google to replace Eclipse ADT [2]. It is likely that code obfuscation will be even more common in the future. With obfuscation, most classes and methods are renamed, often with just one or two alphabetical characters, leading to the loss of most of lexical properties. Fortunately, the application structure is preserved and classes from the Android Framework are not renamed, thus allowing to retrieve some information from the classes that inherit them.

3 Related Work

In this section, we discuss the relevant literature about analysis and antipatterns detection in mobile apps.

Mobile apps are mostly developed using OO languages, such as Java or Objective-C. Since their definition by Chidamber and Kemerer [18], OO metrics have gained popularity to assess software quality. Numerous works validated OO metrics to be efficient quality indicators [13, 15, 23, 34]. This has lead

to the creation of tooled approaches, such as DECOR [29] or iPlasma [26], which are using OO metrics to detect code smells and antipatterns in OO applications. Most of the code smells and antipatterns, like *long method* or *blob class*, detected by these approaches are inspired by the work of Fowler [21] and Brown *et al.* [16]. These approaches are compatible with Java, but since they were mostly developed before the emergence of mobile apps they are not taking into account the specificities of Android apps and are not compatible with Dex bytecode.

With regard to mobile apps, Linares-Vásquez *et al.* [24] used DECOR to perform the detection of 18 different OO antipatterns in mobile apps built using *Java Mobile Edition* (J2ME) [5]. This large-scale study was performed on 1,343 apps and shows that the presence of antipatterns negatively impacts the software quality metrics, in particular metrics related to fault-proneness. They also found that some antipatterns are more common in certain categories of Java mobile apps. Concerning Android, Verloop [38] used popular Java refactoring tools, such as PMD [7] or Jdeodorant [35] to detect code smells, like *large class* or *long method* in open-source software. They found that antipatterns tend to appear at different frequencies in classes that inherits from the Android Framework (called core classes) compare to classes which are not (called non-core classes). For example, *long method* was detected twice as much in core classes in term of ratio. However, 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.* [30] 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 concerning various aspect like implementations, user interfaces or database usages. They are reported to have a negative impact on properties, such as efficiency, user experience or security. We chose to detect some of these code smells with our approach, which are presented in Section 4.4. We selected antipatterns that can be detected by static analysis and despite code obfuscation. Reimann *et al.* are also offering the detection and correction of code smells via the Refactory tool [31]. This tool can detect the code smells from an EMF model. The source code can be converted to EMF if necessary. 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.

Concerning the analysis of Android apps and the study of their specificities, the SAMOA [28] tool allows developers to analyze their mobile apps from the source code. The tool collects metrics, such as the number of packages, lines of code, or the cyclomatic complexity. It also provides a way to visualize external API calls as well as the evolution of metrics along versions and a comparison with other analyzed apps. They performed this analysis on 20 applications and discovered that they are significantly different from classical software systems. They are smaller, and make an intensive usage of external libraries, which leads to a more complex code to understand during maintenance activities. Ruiz *et al.* [32] analyzed Android packages to understand the reuse of classes in Android apps. They extract bytecode and then analyze classes signature for this purpose. They discovered that software reuse via inheritance, libraries, and frameworks is prevalent in mobile apps compared to regular software. Xu [40] also examined APK of 122,570 applications. He determines that developer errors are common in manifest and permissions. He also analyzed the apps' code and observed that Java reflection and code obfuscation are widely used in mobile apps, making reverse-engineering harder. He also noticed the heavy usage of external libraries in its corpus of analyzed apps. Nonetheless, antipatterns were not considered as part of these studies.

4 Paprika: A tooled Approach to detect Software Anti-Patterns

In this section, we introduce the key components of Paprika, our tooled approach for analyzing the design of mobile apps in order to detect software antipatterns.

4.1 Overview of the Approach

Paprika builds on a three-step approach, which is summarized in Figure 1. As a first step, Paprika parses the APK file of the mobile app under analysis to extract some metadata (*e.g.*, app name, package) and a representation of the code. 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 (cf. Section 4.2). As a second step, this model stored into a graph database (cf. Section 4.3). Finally, the third step consists in querying the graph to detect the presence of common antipatterns in the code of the analyzed apps (cf. Section 4.4). Paprika is built from a set of components fitting these steps in order to leverage different analyzers, databases or antipatterns detection mechanisms.

4.2 Step 1: Collecting Metrics from Application Artifacts

Input: One APK file and its corresponding metadata.
Output: A Paprika quality model including entities, metrics and properties.
Description: This steps consists in generating a model of the mobile app and extracting the raw quality metrics from an input artifact. This model is built incrementally, while analyzing the bytecode, and complemented with properties collected from the Google Play Store. From this representation, Paprika builds a model based on six entities: App, Class, Method, Attribute and Variable. The properties described in Table 1 are attached as attributes to these entities, while they are linked together by the relationships reported in Table 2.

Paprika proceeds with the extraction of metrics for each entity. The 34 metrics currently available in Paprika are reported in Table 3. Paprika supports two kinds of metrics: *OO* and *Android-specific*. Boolean metrics are used to determine different kinds of entities, whereas integers are used for counters or when the metrics are aggregated. Contrary to the properties, metrics often require computation or to process the bytecode representation. For example, it is necessary to browse the inheritance tree in order to determine if a class inherits from some Android Framework-specific fundamentals classes, which include:

- *Activity* represents a single screen on the user interface. Activity may start others activities from the same or a different application;
- *Service* is a task that runs in the background to perform long-running operations or to work for remote processes;
- *Content provider* manages shared data and transfer between apps;
- *Broadcast receiver* can listen and respond to system-wide broadcast announcements from the system or other apps;
- *Application* is used to maintain a global application state.

Some composite metrics, such as `ClassComplexity` or `LackofCohesionInMethods`, require more computation based on other raw metrics, thus they are computed at the end of the process.
Implementation: We use the Soot framework [37] and its Dexpler module [14] to analyze APK artifacts. Soot converts the Dalvik bytecode of mobile apps into a Soot internal representation, which is similar to the Java language. Soot can also be used to generate the call graph of the mobile app. This model is built incrementally by visiting the internal representation of Soot, and complemented

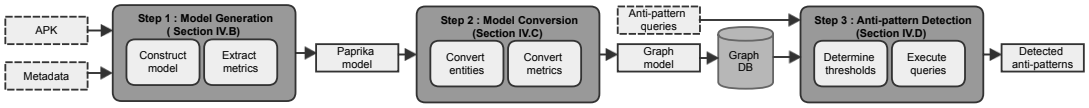


Figure 1: Overview of the Paprika approach to detect software antipatterns in mobile apps.

Table 1: List of Paprika properties.

Name	**Entities**	**Comments**
`name`	All	Name of the entity
`app_key`	All	Unique id of an application
`rating`	App	Rating on the store
`date_download`	App	APK download date
`date_analysis`	App	Date of the analysis
`package`	App	Name of the main package
`size`	App	APK size (MB)
`developer`	App	Developer name
`category`	App	Category in the store
`price`	App	Price in the store
`nb_download`	App	Number of downloads from the store
`parent_name`	Class	For inheritance
`modifier`	Class Variable Method	`public`, `protected` or `private`
`type`	Variable	Object type of the variable
`full_name`	Method	`method_name#class_name`
`return_type`	Method	Return type of the method
`position`	Argument	Position in the method signature

Table 2: List of Paprika relationships.

Name	**Entities**	**Comments**
`APP_OWNS_CLASS`	App – Class	
`CLASS_OWNS_METHOD`	Class – Method	
`CLASS_OWNS_ATTRIBUTE`	Class – Attribute	
`METHOD_OWNS_ARGUMENT`	Method – Argument	
`EXTENDS`	Class – Class	
`IMPLEMENTS`	Class – Class	
`CALLS`	Method – Method	Method call graph
`USES`	Method – Variable	Variable access

Table 3: List of Paprika Metrics

Name	Type	Entities	Comments
NumberOfClasses	OO	App	
NumberOfInterfaces	OO	App	
NumberOfAbstractClasses	OO	App	
NumberOfMethods	OO	Class	
DepthOfInheritance	OO	Class	Integer, minimum is 1.
NumberOfImplementedInterfaces	OO	Class	
NumberOfAttributes	OO	Class	
NumberOfChildren	OO	Class	
ClassComplexity	OO	Class	Sum of methods complexity, Integer
CouplingBetweenObjects	OO	Class	Chidamber and Kemerer [18], Integer
LackofCohesionInMethods	OO	Class	LCOM2 [18], Integer
IsAbstract	OO	Class, Method	
IsFinal	OO	Class, Variable, Method	
IsStatic	OO	Class, Variable, Method	
IsInnerClass	OO	Class	
IsInterface	OO	Class	
NumberOfParameters	OO	Method	
NumberOfDeclaredLocals	OO	Method	Can be different from source code
NumberOfInstructions	OO	Method	Related to number of lines in source code
NumberOfDirectCalls	OO	Method	Numbers of calls made by the method
NumberOfCallers	OO	Method	
CyclomaticComplexity	OO	Method	McCabe [27], Integer
IsGetter	OO	Method	Computed to bypass obfuscation
IsSetter	OO	Method	Computed to bypass obfuscation
IsInit	OO	Method	Constructor
IsSynchronized	OO	Method	
NumberOfActivities	Android	App	
NumberOfBroadcastReceivers	Android	App	
NumberOfContentProviders	Android	App	
NumberOfServices	Android	App	
IsActivity	Android	Class	
IsApplication	Android	Class	
IsBroadcastReceiver	Android	Class	
IsContentProvider	Android	Class	
IsService	Android	Class	

with properties collected from the Google Play Store. Then, PAPRIKA proceeds with the extraction of metrics for each entity by exploring the SOOT model. One should note that, in order to optimize performance and to reduce execution time, these steps are not executed sequentially, but rather executed in an opportunistic way while visiting the SOOT model.

Compared to traditional approaches for antipattern detection [38], using bytecode analysis instead of source code analysis raises some technical issues. For example, we cannot directly access widely-used metrics, such as the number of lines of codes or the number of declared locals of a method. Therefore, we use abstract metrics, that are approximations of the missing ones, like the number of instructions to approximate the number of lines of code.

Moreover, as evoked previously, many applications available on Android markets are obfuscated to optimize size and make reverse-engineering harder. Most methods, attributes, and classes are therefore renamed with single letters. Thus, we cannot rely on lexical data to compute some quality metrics and we have to apply some bypass strategies. For instance, to determine the presence of a getter or a setter we are not observing the method names, but we rather focus on the number and the types of instructions as well as the variable accessed by the method.

4.3 Step 2: Converting Paprika Model as a Graph Model

Input: A PAPRIKA quality model with entity, properties and metrics.
Output: A software quality graph model stored in a database.
Description: We aim at providing a scalable solution to analyze mobile apps at large.Therefore, we use a graph database as a flexible yet efficient solution to store and query the app model annotated with quality metrics extracted by PAPRIKA.

Since this kind of databases are not depending on a rigid schema, the PAPRIKA model is almost as it is described in the previous section. All PAPRIKA entities are represented by nodes, their attributes and metrics are properties attached to these nodes. The relationships between entities are represented by one-way edges.
Implementation: We selected the graph database NEO4J [10] and we are using its Java-embedded version. We chose NEO4J because, when combined with the CYPHER [9] query language, it offers good performance on large-scale datasets, especially when embedded in Java [22]. Furthermore, NEO4J is also able to contain a maximum of 2^{35} nodes and relationships, which match our scalability requirements. Finally, NEO4J offers a straightforward conversion from the PAPRIKA quality metrics model to the graph database.

4.4 Step 3: Detecting Antipatterns from Graph Queries

Input: A graph database containing a model of the applications to analyze and the antipatterns queries.
Output: Software antipatterns detected in the applications.
Description: Once the model loaded and indexed by the graph database, we use the database query language to detect common software antipatterns. Entity nodes which implements antipatterns are returned as results for all analyzed applications.
Implementation: We use the CYPHER query language [9] to detect common software antipatterns as illustrated by listings 1 and 2.

All OO antipatterns are detected using a threshold to identify abnormally high value from others commons values. To define such thresholds, we collect all the values of a specific metric and we identify outliers. We use a Tukey Box plot [36] for this task. Figure 2 illustrates this approach on the LCOM, number of methods and number of attributes, which are used in the request to detect Blob classes as presented in Listing 1. All values superior to the upper whisker are considered as very high whereas all values inferior to the lower one are very low. The upper border of the box represents the *first quartile* (Q1) whereas the lower border is the *third quartile* (Q3), the distance

between Q1 and Q3 is called the *interquartile range* (IQR). The upper whisker value is given by the formula $Q3+1.5\times IQR$, which is equal to 12 for the number of methods in our example. It means that if the number of methods exceeds 12, then it is considered as an outliers and can be tagged as a class containing a high number of methods. By combining the three thresholds, we are able to detect Blob classes. The usage of this statical method allows us to set thresholds that are specific to the input dataset, consequently results may vary depending on the mobile apps included in the analysis process. Thus, the thresholds are representative of all applications in the dataset and not only the currently analyzed application.

Currently, PAPRIKA supports 8 antipatterns, including 4 Android-specific antipatterns:

Blob Class (BLOB) - OO A Blob class, also knows as *God class*, is a class with a large number of attributes and/or operations [16]. The Blob class handles a lot of responsibilities compared to other classes. Attributes and methods of this class are related to different concepts and processes, implying a very low cohesion. Blob classes are also often associated with numerous data classes. Blob classes are hard to maintain and increase the difficulty to modify the software. In PAPRIKA, classes are identified as Blob classes whenever the metrics `numbers_of_attributes`, `number_of_methods` and `lack_of_cohesion_in_methods` are *very high*. The CYPHER query for this antipattern is described in Listing 1.

Listing 1: CYPHER query to detect a Blob class.

```
MATCH (cl:Class)
WHERE
  cl.lack_of_cohesion_in_methods > 15
  AND cl.number_of_methods > 12
  AND cl.number_of_attributes > 8
RETURN cl
```

Swiss Army Knife (SAK) - OO A *Swiss army knife* is a class with numerous interface signatures, resulting in a very complex class interface designed to handle a wide diversity of abstractions. This type of class is hard to understand and to maintain because of the resulting complexity [16]. A SAK is detected by PAPRIKA when a class implements a large number of interfaces.

Long Method (LM) - OO *Long methods* are implemented with much more lines of code than other methods. They are often very complex, and thus hard to understand and maintain. These methods can be split into smaller methods to fix the problem [21]. PAPRIKA identifies a *long method* when the number of instructions for one method is very high.

Complex Class (CC) - OO A *complex class* is a class containing complex methods. Again, these classes are hard to understand and maintain and need to be refactored [21]. The class complexity is calculated by summing the internal methods complexities. The complexity of a method can be calculated using McCabe's Cyclomatic Complexity [27].

Internal Getter/Setter (IGS) - Android On Android, fields should be accessed directly within a class to increase performance. The usage of an *internal getter or a setter* converts into a virtual invoke, which makes the operation three times slower than a direct access [1, 17]. PAPRIKA is able to identify internal getter and setter despite code obfuscation, and consequently identify such calls from the method call graph. The CYPHER query for this Android-specific antipattern is reported in Listing 2. This query matches two methods from the same class with the first one calling the second one, flagged as a getter or a setter.

Listing 2: Cypher query to detect Internal Getter/Setter.

```
MATCH (m1:Method)-[:CALLS]->(m2:Method),
      (cl:Class)
WHERE
  (m2.is_setter OR m2.is_getter)
  AND cl-[:CLASS_OWNS_METHOD]->m1
  AND cl-[:CLASS_OWNS_METHOD]->m2
RETURN m1
```

Member Ignoring Method (MIM) - Android In Android, when a method does not access an object attribute, it is recommended to use a static method. The static method invocations are about 15%–20% faster than a dynamic invocation [1, 17]. PAPRIKA can detect such methods since the access of an attribute by a method is extracted during the analysis phase. The CYPHER query for this antipattern is described in Listing 3. This request explore node properties to return non-static methods which are not constructor and relations to detect methods which are not using any class variable nor calling other methods. Such detected methods could have been made static to increase performance without any other consequences on the implementation.

Listing 3: Cypher query to detect Member Ignoring Method.

```
MATCH (m:Method)
WHERE
  NOT HAS(.`is_static`)
  AND NOT HAS(m.is_init)
  AND NOT m-[:USES]->(:Variable)
  AND NOT (m)-[:CALLS]->(:Method)
RETURN m
```

No Low Memory Resolver (NLMR) - Android When the Android system is running low on memory, the system calls the method `onLowMemory()` of running activities, which are supposed to trim their memory usage. If this method is not implemented by the activity, the Android system kills the process in order to free memory, and can cause an abnormal termination of programs [17]. As overridable methods of Android-specific classes like `Activity`, `Service`, `ContentProvider` or `Broadcast receivers` are not concerned by obfuscation, consequently their presence can be checked by PAPRIKA.

Leaking Inner Class (LIC) - Android In Java, non-static inner and anonymous classes are holding a reference to the outer class, whereas static inner classes are not. This could provoke a memory leak in Android systems [17, 25]. Given that the PAPRIKA model contains all inner classes of the application and a metric to identify static class is attached to classes, *leaking inner classes* are detected by a dedicated query in PAPRIKA.

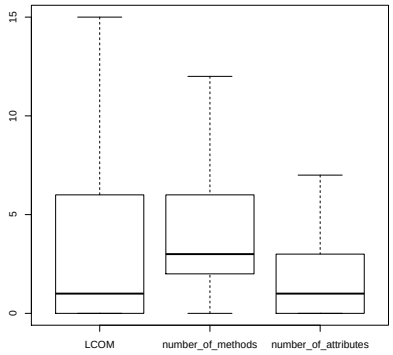


Figure 2: Box plots generated to define the Blob class detection thresholds.

5 Empirical Validation

This section describes the experimental protocol we followed to assess PAPRIKA as well as the results we obtained from the analysis of 15 mobile apps downloaded from the Google Play Store.

5.1 Research Questions

The research questions we address along these experiments are:

RQ$_1$: *Can antipatterns originating from the source code be detected at the bytecode level?*

Motivation: The compilation into bytecode and the eventual code obfuscation imply a loss or a transformation of the information available in the source. Java `.jar` files were already analyzed with success for OO antipatterns [24], but there is no evidence that the code was obfuscated and that a similar approach will be efficient when applied on Dalvik bytecode.

RQ$_2$: *To what extent mobile apps exhibit OO antipatterns?*

Motivation: As previously exposed, Android

apps are different from non-mobile applications. They are smaller and are relying more heavily on external libraries and classes reuse. Moreover, the Android Framework could have an impact on developers pratice concerning antipatterns. We want to investigate if these differences have an impact on the presence of OO antipatterns.

RQ$_3$: *To what extent mobile apps exhibit Android-specific antipatterns?*

Motivation: There is a lack of studies concerning the presence of Android-specific antipatterns in mobile apps. We want to determine if these antipatterns actually exists in the publicly available apps and how frequent they are. This knowledge can help practitioners to know where to focus their effort during maintenance tasks.

RQ$_4$: *Are OO and Android-specific antipatterns present in the same proportion?*

Motivation: In this research question, we investigate whether certain kind of antipatterns are more frequent than others. OO antipatterns are well documented both in literature and on Internet, but the resources are scarce concerning Android antipatterns. We want to investigate if this difference is justified by a lack of knowledge due to the recency of the domain or because Android antipatterns are scarcer in mobile apps.

5.2 Subjects

We apply our Paprika approach to detect 8 different antipatterns, presented in Section 4.4, they are all issued from the literature.

5.3 Objects

Witness application To validate and calibrate our approach, we first developed a witness mobile app implementing several instances of each type of antipatterns. In particular, this application implements 62 instances of the 8 antipatterns we consider in this paper. The source code of the witness mobile app, available on GitHub , represents 51 classes, including 6 activities and 205 methods. Classes generated during the compilation, such as `BuildConfig` and `R`, are included in this count.

Applications under analysis We also analyzed 15 free Android apps that are available from the Google Play Store [12]. All these mobile apps and associated metadata were downloaded in June 2014, and are summarized in Table 4. The reported size includes all the application resources, including images, data files or third-party libraries, which the latter are not analyzed by Paprika. We selected these apps because they were all ranked in the top 150 apps of their categories. It should be noted that, although this top is based on the number of downloads, this number could be rather high or low depending on the categories. In order to validate Paprika, we diversified the selected apps in terms of category, number of downloads, size and rating, but we also considered some popular apps (*e.g.*, *Facebook, Skype, Twitter*). The complexity of mobile apps therefore varies from 3 classes (*Free 5000 Movies*) to more than 9,000 classes (*Facebook*).

5.4 Evaluation Protocol

We assessed our approach with a sensitivity analysis, a comparison to other tools, and an evaluation of the impact of the obfuscation on our witness mobile app. Then, we analyzed all the mobile apps using Paprika on an Intel Core i5-2450M and with 4GB RAM. We first load all the mobile apps, which took between 5 and 15 minutes per app, depending on size and complexity. All the resulting quality models were inserted in a single Neo4J database instance. Then, as described in Section 4.4, we compute the thresholds of queries with Boxplot and we execute our 8 software antipatterns queries for each application using Cypher. Note that we excluded the witness mobile app from the computation of thresholds, to ensure that the results are representative from Google Play Store official applications.

Table 4: Description of mobile apps under analysis

App name	Size (Mb)	Downloads	Rating(/5)	Classes	Methods
Adobe Reader	8.02	100,000,000+	4.3	902	5,214
Android Temperature	9.67	500,000+	4	373	2,238
Facebook	22.14	500,000,000+	3.9	9,117	46,867
Fitnet Apps	0.43	50+	4.9	13	74
FLV HD MP4 Video Player	12.11	1,000,000+	4.3	364	2,332
Free 5000 Movies	0.04	1,000,000+	3	3	5
Opera Mini browser for Android	0.89	100,000,000+	4.4	182	1,830
Savoir Maigrir avec J-M Cohen	1.87	50,000+	2.9	449	2,190
Simulator Laser	4.92	5,000,000+	2.1	225	1,205
Skype - free IM & video calls	17.55	100,000,000+	4.1	2,364	12,901
Superbuzzer Trivia Quiz Game	2.80	500,000+	3.9	858	5,265
Tcheck'it - Gagnez de l'argent	2.92	5,000+	2.2	1,306	9,268
Twitter	9.94	100,000,000+	4.1	4,335	29,309
Video Chat Rooms - Chat.Org	4.34	100,000+	4.1	7	94
Zoom Camera Free	0.67	5,000,000+	4	415	2,131

5.5 Preliminary Assessments

Sensitivity analysis We performed a sensitivity analysis for the thresholds we compute with median, mean, Q3 and $Q3 + 1.5 \times IQR$ used as values for the detection of Blob, SAK, LM and CC. As one can observe in Table 5, the usage of $Q3 + 1.5 \times IQR$ offers the best results for threshold-based queries.

Table 5: Sensitivity analysis for the threshold

Threshold	Precision	Recall	F1
Median	0.348	1	0.5161
Mean	0.8	1	0.8889
Q3	0.7619	1	0.865
Q3+1.5*IQR	1	1	1

Globally, we expected the presence of 62 antipatterns from the source code and discovered 62 of them in the witness mobile app by analyzing the APK with Paprika, which provides a precision of 1, a recall of 1 and a F1 value of 1 for all the antipattern queries.

Comparison with other tools We compared our approach with tools dedicated to the detection of code smells and OO antipatterns form the source code of Java applications. PMD is a popular tool for Java applications, it analyzes code to detect programming flaws defined by rules. It currently contains 347 rules, including *ExcessiveMethodLength* which is similar to the *long method* antipattern. These rules are mostly define with static thresholds, which can be adjusted in the preferences of PMD. By default, a method is flagged by the *ExcessiveMethodLength* rule if it contains more than 100 lines. However, this value sounds arbitrary and inappropriate for mobile apps were 75% of methods have less than 15 instructions as showed by our statistical analysis performed on our 15 applications. In particular, only 3 instances are detected in our witness mobile app with this predefined threshold. Other values can be used as threshold, but PMD does not provide any mechanism to help in setting this value.

To address this problem, approaches like JDeodorant or Ptidej use statistical analysis on the current application to determine thresholds. In this way, JDeodorant is able to detect one instance of *Blob class* using the average ratio of cohesion over coupling for all classes [20]. Using the Paprika approach, we detect two *Blob classes*. However, when removing most of the classes to keep only these two *Blob classes*, JDeodorant is not able anymore to detect any *Blob class* even if these classes have not been modified. By computing the thresholds from a crowd of mobile apps instead

of a single instance, PAPRIKA provides a robust approach to detect antipatterns.

Impact of obfuscation on antipatterns detection To evaluate the impact of code obfuscation on our results, we performed the same detection on our witness mobile app obfuscated using the popular Proguard 5.1 and Stringer Java Obfuscator 1.6.11. Proguard is an open-source Java obfuscator compatible with Android, which is embedded in many development environments like Netbeans, EclipseME or Google's Android Studio. Stringer Java Obfuscator is a commercial tool that provides string encryption and is compatible with most IDE. Both tools support classes, methods and variables renaming as well as code optimization. As reported in Proguard FAQ, these tools are not performing any flow obfuscation in order to avoid negative effects on performance and size. However, the optimization step can restructure the code, including dead code removal. As we can observe in Table 6, Stringer has no impact on the detection, while Proguard slightly affects the recall of our detection. All detected instances are true positive as shown by the precision.

The impact of Proguard on recall can be explained by two factors. First, the Proguard optimization process removes an instance of *long method* while optimizing resources concerning application style in the Android `R` class. Then, the obfuscation process prevents us from detecting inner classes in the bytecode, hence all *leaking inner classes* are not detected anymore. The impact of obfuscation is therefore limited, especially since we were able to detect inner classes in all mobile apps studied in this paper.

5.6 Analysis Results

Table 7 summarizes the results we obtained for the detection of 8 antipatterns from our input dataset composed of 15 Android official apps and a witness app. The detailed results and statistics for all mobile apps are available online. 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. For example, a class can be a *complex class* and a *blob class* simultaneously. We further exploited these results to answer our research questions.

Table 7: PAPRIKA results for the detection of 8 antipatterns in 15 applications.

	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%**

5.6.1 RQ$_1$ Can antipatterns originating from the source code be detected at the bytecode level?

The results on our witness mobile app already proved that we were able to detect antipatterns from bytecode. This observation can be acknowledged with the results on the 15 applications. We computed the thresholds to be representative for all these applications. Antipatterns were detected in all of them regardless of their specificities and despite code obfuscation.

5.6.2 RQ$_2$ To what extent mobile apps exhibit OO antipatterns?

LM (10.41 % of methods) and CC (11.83% of classes) are frequent and common in all mobile apps with the same order of magnitude. Around 3% of all classes are Blobs. Our observation also shows that almost one third of all activities are considered as Blobs. Our hypothesis is that the design of the Android Framework tends to encourage the presence of *"blob*

Table 6: Impact of obfuscation on antipatterns detection

Tool	**Precision**	**Recall**	**F1**
Stringer obfuscation	1	1	1
Stringer obfuscation and optimization	1	1	1
Proguard optimization	1	0.9838	0.9918
Proguard optimization and obfuscation	1	0.9032	0.9491

activities". This practice was previously identified in one mobile app [28]. Our results seems to confirm that this is a frequent antipattern, but further investigation are needed to confirm this assumption on a larger dataset. Furthermore, the proportions for these three antipatterns are similar to the one observed in non-mobile applications by the DECOR approach [29]. However, we detected less than 1% SAK, whereas with the DECOR approach it is around 2.9%. The higher value of SAK is in *Facebook* (1.15% of classes), which is an application making a large usage of interfaces as we discovered when analyzing our results. In conclusion, OO antipatterns are common in Android apps and they are as frequent as in non-mobile applications, except for SAK.

5.6.3 RQ_3 To what extent mobile apps exhibit Android-specific antipatterns?

The NLMR is the most frequent antipattern in proportion as it appears in all the applications (39.23% of all activities) except in the single activity of *Video Chat Rooms*. This high presence can be explained by the fact that developers are not aware of how Android manages the memory and that the method `onLowMemory()` exists. Another reason is that small mobile apps with less than 100 classes probably do not have any cache or resources to free. Twitter, Facebook, and Skype are interesting cases because they have very few NLMR compared to the numbers of activities (around 15%). This shows that the memory management is considered when these applications are developed and therefore they are presenting a better quality for memory management.

In proportion, the LIC is the second most frequent antipattern (37.52% of all classes) and it appears in all mobile apps. Even if the ratio is really high, we are not surprised by this observation. The usage of inner and anonymous classes is common in Java and Android applications. Our assumption is that mobile developers do not consider the performance implications of inner classes, even in systems where memory leaks are more problematic due to the limited resources. Another explanation is that they cannot use static classes or weak references due to implementations constraints. An inner class may need to access the attributes of its outer class.

The MIM is another frequent antipattern (19.02 % of all methods) for all the analyzed mobile apps. Our hypothesis is that developers are using static methods only when needed by the implementation and not for efficiency purposes.

These antipatterns are detected by exploring the model, thus their detection is not impacted by thresholds. Consequently, this values are independent of the analyzed dataset and their presence is acknowledged. They are known to affect the efficiency of mobile apps and thus, a refactoring focusing on the concerned classes and methods can improve the mobile app performance without any trade-off.

5.6.4 RQ_4 Are OO and Android-specific antipatterns present in the same proportion?

Our results shows that Android-specific antipatterns are more frequent than OO ones. It is interesting to notice that the MIM, NLMR and LIC are all antipatterns, which have been recently defined for Android and thus they are not yet really well referenced by the literature and developer's documentations, that may be a cause of their strong presence. 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.

5.6.5 Other observations

One can observe that *Fitnet Apps* and *Video Chat Rooms - Chat.Org* are the applications with the biggest proportion of antipatterns. It is interesting to note that they are very small mobile apps when looking at the numbers of classes and methods. We are observing that very small apps tend to have a higher proportion of antipatterns, however our sample is too small to validate this assumption statistically. Moreover, this correlation between the mobile app complexity and the ratio of antipatterns does not seem to hold when comparing medium and big applications. In order to confirm the assumptions we have made and to look for correlations between the applications characteristics, we are planning to conduct the thorough study on a bigger sample containing thousands of mobile apps.

5.7 Threats to validity

In this section, we discuss the threats to validity of our study based on the guidelines provided by Wohlin *et al.* [39].

Construct validity threats concern the relation between theory and observations. In this study, these threats could be due to errors during the analysis process. We validate our approach on a witness mobile app developed to explicitly include 8 antipatterns and we checked manually that the stored models, metrics and antipatterns detected were corresponding to the source code even when obfuscation and optimization were used.

Internal validity concerns our selection of subject systems and analysis methods. The usage of Boxplot to define thresholds could be criticized since it depends on the analyzed dataset and could have lead to other conclusions with other applications. However, only half of the software antipatterns queries depends on these thresholds. Moreover, the usage of all the dataset to determine the thresholds is a guarantee that the thresholds are more representative than if they were arbitrary defined or calculated with only one application.

External validity threats concern the possibility to generalize our findings. We tried to collect a wide diversity of mobile apps, and we are thus thinking that they are significant and representative of what is distributed by the Google Play Store and installed on Android systems. However, future studies should consider larger sets of official Android apps. Moreover, the approach and results reported in this paper are specific to Android and cannot be generalized to other systems, such as iOS or Windows Phone.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our experiments. In addition to the results presented in this paper, the source code of our mobile app as well as our graph databases are available online.

Finally, the *conclusion validity* threats refer to the relation between the treatment and the outcome. We paid attention not to make conclusions that cannot be validated with the presented results.

6 Conclusion and future work

In this paper, we presented Paprika, an innovative tooled approach to detect antipatterns directly from Android packages. Paprika does not require the source code to perform this detection, and works even when the code has been obfuscated to prevent reverse-engineering. Thus, Paprika is allowing the analysis of any application available from app stores.

Our approach was applied on a witness application to validate its efficiency on the detection of 4 OO antipatterns and 4 Android-specific antipatterns. This validation confirmed that Paprika is able to detect such antipatterns. We also performed a study on 15 popular applica-

tions downloaded on the Google Play Store and found significant results. We discovered that antipatterns implemented in the source code can be detected at the bytecode level (RQ_1). Both types of antipatterns were found in all analyzed applications. Results also show that OO antipatterns are as present in Android applications as in non-mobile applications, with the exception of the *Swiss army knife* antipattern (RQ_2). We also discovered that Android-specific antipatterns are highly frequent and common in all applications (RQ_3), indeed they are even more frequent that OO antipatterns (RQ_4).

Concerning future work, we plan to implement the detection of more antipatterns and to perform our analysis on a larger set of applications. In this purpose, we already collected thousands of applications from the Google Play Store. We hope that this large-scale study will validate the tendencies we observed in this paper, but also that we will observe a correlation between the frequency of antipatterns and their metadata like the rating or the category. We will also study the evolution of antipatterns between different versions of an application. Finally, we are planning to use our large database of knowledge to discover new antipatterns, which are existing in the wild but have not been defined in the literature, yet.

Acknowledgements

These researches are co-funded by Université of Lille, Université du Québec à Montréal, Inria, The *Natural Sciences and Engineering Research Council of Canada* (NSERC) and *Fonds de recherche du Québec - Nature et technologies* (FQNRT).

References

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

[2] Android studio. `https://developer.android.com/sdk/installing/studio.html`. [Online; accessed November-2014].

[3] 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].

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

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

[6] 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].

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

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

[9] Cypher. `http://neo4j.com/developer/cypher-query-language`. [Online; accessed November-2014].

[10] Neo4j. `http://neo4j.com`. [Online; accessed November-2014].

[11] Tools to work with android .dex and java .class files. `https://code.google.com/p/dex2jar`. [Online; accessed November-2014].

[12] Google play store. `https://play.google.com`, 2014. [Online; accessed November-2014].

[13] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra. Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study. *Software process: Improvement and practice*, 14(1):39–62, 2009.

[14] 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.

[15] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[16] 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.

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

[18] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

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

[20] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1037–1039. ACM, 2011.

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

[22] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proc. of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM, 2013.

[23] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, 80(7):1120–1128, 2007.

[24] 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.

[25] 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].

[26] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume)*. Citeseer, 2005.

[27] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[28] 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.

[29] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A. Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010.

[30] J. Reimann, M. Brylski, and U. Aßmann. 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.

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

[32] 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.

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

[34] Y. Singh, A. Kaur, and R. Malhotra. Empirical validation of object-oriented metrics for predicting fault proneness models. *Software quality journal*, 18(1):3–35, 2010.

[35] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

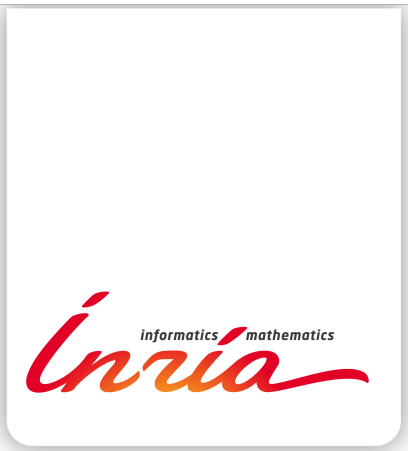
[36] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[37] 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.

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

[39] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer, 2012.

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



RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399