Tracking the Software Quality of Android Applications along their Evolution

Geoffrey Hecht[1,2], Omar Benomar[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, benomar.omar@courrier.uqam.ca, 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 requirements may result in poor design choices, also known as *antipatterns*, which may incidentally degrade software quality and performance. Thus, the automatic detection and tracking of antipatterns in this apps are important activities in order to ease both maintenance and evolution. Moreover, they guide 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 its infancy. In this paper, we analyze the evolution of mobile apps quality on $3,568$ versions of 106 popular Android applications downloaded from the Google Play Store. For this purpose, we use a tooled approach, called PAPRIKA, to identify 3 object-oriented and 4 Android-specific antipatterns from binaries of mobile apps, and to analyze their quality along evolutions.**

***Keywords*—Android, antipattern, mobile app, software quality.**

I. INTRODUCTION

Software evolve over time, inevitably, to cope with the introduction of new requirements, to adapt to new environments, to fix bugs, or to improve software design. However, regardless of the type of changes performed, the software quality may deteriorate as a result of software aging [38].

Software quality declines along evolutions because of the injection of poor design and implementation choices into software. Code smells and antipatterns are symptoms of such bad choices [17]. Additionally, the introduction of antipatterns may affect software maintainability [53], [54], increase the change-proneness [26], the fault-proneness [27], and the complexity [50] of software modules. Therefore, the existence of antipatterns in software is a relevant indicator of its quality.

The research community recognized this issue and proposed several techniques to detect code smells and antipatterns [34], [37]. Most of them are concerned with identifying possible deviations from *Object-Oriented* (OO) principles, such as functional decomposition and encapsulation. Despite the availability of software development kits for mobile apps, specific code smells and antipatterns may still emerge, due to the limitations and constraints on resource like memory, CPU or screen sizes. Mobile apps may also exhibit OO code smells and antipatterns, such as Blob class, which may decrease their quality as they evolve. To the best of our knowledge, most of the proposed techniques and methods for code smells and antipatterns detection in mobile applications do not involve large empirical data to give a genuine picture of the issue, rather they detect antipatterns on a single version of few specific open source systems [49]. This is all the more true in the case of techniques for Android applications as the research field is in its infancy. Also, OO antipatterns and mobile-specific ones are rarely studied together in order to assess the software quality of Android applications.

Regarding software quality along software evolutions, most techniques are based on the analysis of variation of software metrics over time. For instance, studies have been performed on the distribution of bug introduction to software along its evolutions [36] or on an aggregation of metrics as a quality indicator [20]. However, these techniques were not applied to mobile applications.

In this paper, we present a fully automated approach that monitors the evolution of mobile apps and assesses their quality by considering antipatterns. To this end, we propose a software quality estimation technique based on the consistency between software artifacts size and OO and Android antipatterns. To identify these antipatterns from Android applications, we use a metrics-based detection technique. Rather than evaluating each application independently, our approach leverages the whole dataset to detect and evaluate the evolution of antipatterns in each application. The antipatterns detection and quality tracking are automatic processes included in PAPRIKA, our tooled approach. Typically, we intend to answer the two following research questions:

- **RQ1:** Can we infer software quality evolution trends across different versions of Android applications?
- **RQ2:** How does software quality evolve in Android applications?

The rationale behind our approach is the close relationship between antipatterns and software artifacts size. A recent large empirical study [45] showed that most antipatterns are introduced at the creation of software artifacts. The same study indicates that some antipatterns are introduced as a result of several changes performed on software artifacts along evolutions. Based on these findings, we propose to study the variations between software artifacts sizes and the number of antipatterns contained in them. Typically, we intend to track the lack of correlation between the number of antipatterns and the software size along evolutions. Our study is based on a dataset made of 106 Android applications and $3,568$ versions. We choose popular applications from the Google Play Store in order to be representative of complex and up-to-date

applications. However, most of these applications are not open-source. Thus, it was necessary to detect the antipatterns at the bytecode granularity. In this way, our approach could be used by both developers and app store providers to track the apps quality evolution at both global and local scales.

The main contributions of this paper are: (1) an approach to track and evaluate the quality of android apps along their evolutions via the detection of antipatterns, (2) the observation of particular relationships between these antipatterns in the case of mobile applications, (3) the identification of 5 quality evolution trends and their possible causes inferred by the analysis of our dataset and, finally (4) an empirical study involving over 3500 version on which we apply our tooled approach.

This paper is organized as follows: Section II gives a brief background on Android apps and bytecode-based techniques. Section III discusses different contributions related to our work. Our automatic tooled approach, PAPRIKA, is detailed in Section IV. The results of the application of our approach are presented in Section V. Section VI concludes the paper.

II. 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 [5]. 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 [5]. 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], [21], [42]. 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 [5], [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 the Google Play Store are obfuscated [52] in order to prevent reverse-engineering. The ProGuard tool used to obfuscate code is even pre-installed on the beta of Android Studio provided by Google to replace Eclipse ADT [3]. 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.

III. Related Work

In this section, we discuss the relevant literature about analysis and antipatterns detection in mobile apps and related work on software evolution.

Mobile apps are mostly developed using OO languages, such as Java or Objective-C. Since their definition by Chidamber and Kemerer [19], OO metrics have gained popularity to assess software quality. Numerous works validated OO metrics to be efficient quality indicators [12], [15], [28], [43]. This has lead to the creation of tooled approaches, such as DECOR [35] or iPLASMA [31], which use 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 [23] and Brown *et al.* [17]. 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.* [29] used DECOR to perform the detection of 18 different OO antipatterns in mobile apps built using *Java Mobile Edition* (J2ME) [6]. 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 with certain categories of Java mobile apps. Specifically in Android apps, Verloop [49] used popular Java refactoring tools, such as PMD [7] or JDEODORANT [44] 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 inherit 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.* [39] 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 IV-D. 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 [40]. 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 [33] 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.* [41] analyzed Android packages to understand the reuse of classes in Android apps. They extract bytecode and then analyze class signatures for this purpose. They discovered that software reuse via inheritance, libraries, and frameworks is prevalent in mobile apps compared to regular software. Xu [52] also examined APKs of $122,570$ applications and 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.

Much work has been done concerning the assessment of software quality throughout the evolution of OO non-mobile applications. Zhu *et al.* [56] monitor software quality throughout the evolution by considering three different modularity views: Package, Structural and Semantic. They assess software quality by computing the deviation trends between the considered modular views to indicate on quality evolution. The idea behind their approach is that if the considered different views are well aligned, then software quality is good. They studied the quality evolution of three open source Java systems. Zhang and Kim [55] propose to study software evolution quality by monitoring the number of defects using *Statistical Process Control* (SPC) and control charts (c-charts). They examined over 60 c-charts representing different Eclipse and Gnome components and identified 6 common quality evolution patterns. The quality evolution patterns serve as indicators of symptomatic situations that development teams should address. For instance, the *Roller Coaster* pattern, which represents large variations of defect numbers, suggests that software quality is unstable and that better management and planning is required to ensure high and consistent quality. Tufano *et al.* [45] conducted a large scale empirical study to investigate the code smells introduction by developers by analyzing the change history of software projects. The authors aim at identifying how code smells are introduced in software along its evolutions. Their major findings are that most code smells are introduced when files are created and new features are developed or existing ones are enhanced. Van Emden and Moonen [48] propose an approach to detect code smells in Java programs. They distinguish between two type of smells: primitive smells which are detected directly from code and derived smells which are deduced from the context. The detected antipatterns are then presented to the developers for inspection and quality assurance using visualization. In this approach, the developers are left with the responsibility of assessing the quality by analyzing the visualization. Our approach provides the developers an estimation of software quality based on antipatterns.

Unlike the above mentioned contributions, and regardless of software quality along evolutions, some studies analyze software evolution to abstract higher level information with the purpose of comprehension. Barry *et al.* [13] propose a method to identify software evolution patterns. The pattern identification is based on software volatility information. Volatility is approximated by computing the amplitude, dispersion, and periodicity of software changes at regular intervals in the software history. Each period is defined by a volatility class, and sequence analysis is applied to reveal similar patterns in time. Xing and Stroulia [51] present an approach for understanding evolution phases and styles of object-oriented systems. The authors use a structural differencing algorithm to compare changing system class models over time, and gather the system's evolution profile. The resulting sequence of structural changes is then analyzed to gain insight about the system's evolution. Finally, Benomar *et al.* [16] investigate the automatic detection of software development phases by studying software evolutions. They use a search-based technique to identify software evolution periods having similar development activities. Search heuristics, such as development rate, importance and type of changes are used to define and understand software evolution.

Our proposed approach aims at identifying hotspots in terms of software quality along evolutions. We consider the analysis of mobile apps evolution and apply a novel approach to detect antipatterns, which we use to estimate the software quality. The proposed approach is automatic and takes as input versions of mobile apps and monitors their quality along evolutions.

IV. 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.

A. Overview of the Approach

PAPRIKA builds on a four-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 V). As a second step, this model stored into a graph database (cf. Section IV-C). The third step consists in querying the graph to detect the presence of common

antipatterns in the code of the analyzed apps (cf. Section IV-D). PAPRIKA is built from a set of components fitting these steps in order to leverage different analyzers, databases or antipatterns detection mechanisms. Each analyzed APK is automatically stored into the database. Thus the database can contains a version history for each analyzed application. Finally, the last step use this version history to compute a software quality evolution score (cf. Section IV-E).

B. Step 1: Collecting Metrics from Application Artifacts

Input: One APK file and its corresponding metadata.

Output: A PAPRIKA quality model including entities, properties and metrics.

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 eight entities: App, Class, Method, Attribute, Variable,ExternalClass and ExternalMethod. ExternalClass and ExternalMethod represents entities from the Java API, the Android framework or third-party libraries. The properties described in Table I are attached as attributes to these entities, while they are linked together by the relationships reported in Table II.

TABLE I. 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	Argument position in the method signature

TABLE II. LIST OF PAPRIKA RELATIONSHIPS.

Name	Entities
`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 – (External)Method
`USES`	Method – Variable

PAPRIKA proceeds with the extraction of metrics for each entity. The 34 metrics currently available in PAPRIKA are reported in Table III. 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 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`, require more computation based on other raw metrics, thus they are computed at the end of the process.

Implementation: We use the SOOT framework [47] 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 with properties collected from the Google Play Store. Then, PAPRIKA proceeds with the extraction of metrics for each entity by exploring the SOOT model. In order to optimize performance and to reduce execution time, these steps are not executed sequentially, but in an opportunistic way while visiting the SOOT model.

Compared to traditional approaches for antipattern detection [49], 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 mentioned 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 getters/setters, we are not observing the method names, rather we focus on the number and types of instructions as well as the variables accessed by the method.

C. Step 2: Converting Paprika Model as a Graph Model

Input: A PAPRIKA quality model with entities, 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. We also wants to keep an history of each version analysis. 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 database is not depending on a rigid schema, the PAPRIKA model is almost as it is described in

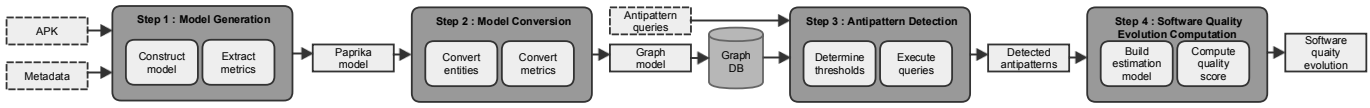

Fig. 1. Overview of the PAPRIKA approach to detect software antipatterns in mobile apps and analyze their evolution.

TABLE III. LIST OF PAPRIKA METRICS

Name	Type	Entities	Comments
NumberOfClasses	OO	App	
NumberOfInterfaces	OO	App	
NumberOfAbstractClasses	OO	App	
NumberOfMethods	OO	Class	
DepthOfInheritance	OO	Class	Integer value, minimum is 1.
NumberOfImplementedInterfaces	OO	Class	
NumberOfAttributes	OO	Class	
NumberOfChildren	OO	Class	
ClassComplexity	OO	Class	Sum of methods complexity, Integer value
CouplingBetweenObjects	OO	Class	Chidamber and Kemerer [19], Integer value
LackofCohesionInMethods	OO	Class	LCOM2 [19], Integer value
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	Numbers of called made by other methods
CyclomaticComplexity	OO	Method	McCabe [32], Integer value
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	

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 used 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 [25]. 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.

D. Step 3: Detecting Anti-patterns from Graph Queries

Input: A graph database containing a model of the mobile apps 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. The results are grouped by versions.

Implementation: We use the CYPHER query language [9] to detect common software antipatterns as illustrated in 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 for the whole dataset and we identify outliers. We use a Tukey Box plot [46] for this task. 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 15 for the number of methods in our example for Blob Class as described in Listing 1. It means that if the number of methods exceeds 15, then it is considered as an outlier 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 statistical 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 7 antipatterns, including 4 Android-specific antipatterns:

Blob Class (BLOB) - OO: A Blob class, also know as *God class*, is a class with a large number of attributes and/or methods [17]. 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 > 20
  AND cl.number_of_methods > 15
  AND cl.number_of_attributes > 8
RETURN cl
```

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 [23]. 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 [23]. The class complexity is calculated by summing the internal methods complexities. The complexity of a method can be calculated using McCabe's Cyclomatic Complexity [32].

Member Ignoring Method (MIM) - Android: In Android, when a method does not access to an object attribute, it is recommended to use a static method. The static method invocations are about 15%–20% faster than a dynamic invocation [2], [18]. 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 2. This request explores node properties to return non-static methods that are not constructors and relations to detect methods that 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 2. 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
```

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 [18], [30]. 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.

UI Overdraw (UIO) - Android: The Android Framework API provides two methods in the `Canvas` Class (`clipRect()` `quickReject()`) to avoid the overdraw of non-modified part of the UI. The usage of these method is recommended by Google to increase app performance and improve user experience [1]. Thus, we are using the data stored in PAPRIKA database external API calls to detect views which not use one of these methods.

Heavy Broadcast Receiver (HBR) - Android: Similarly to services, android broadcast receivers can trigger ANR when their `onReceive()` methods perform lengthy operations [4]. Therefore, we are using the same heuristic to detect them.

E. Step 4: Computing Software Quality using Anti-patterns

Input: The Android application detected antipatterns.

Output: Evolution of software quality score throughout the mobile applications versions.

Description: This step consists of scoring each version of the mobile application. The score serves as an estimation of the mobile app quality in a particular version. Our estimation of software quality is based on the consistency between applications size and the number of detected antipatterns. Typically, we compute a quality score for each version of the application and track the values of the score along the entire history of the application (successive versions).

Implementation: To compute our software quality score, we first build an estimation model using linear regression. The linear regression model represents the relationship between the number of antipatterns and the size of an application. The linear regression is computed by minimization of squared residuals. The number of classes is used at explanatory variable for *BLOB* (r=0.90), *LM* (r=0.91), *CC* (r=0.92), *MIM* (r=0.80). The number of inner classes is used for *LIC* (r=0.80). The number of views and the number of broadcast receivers are respectively used for *UIO* (r=0.95) and *HBR* (r=0.88). Then, the software quality score of an application at a particular version, with a value for the number of antipatterns and a value of size, is computed as the additive inverse of the residual. A larger positive residual value suggests worst software quality because it means that the current observed version of the application has more antipatterns with respect to its size than the norm (linear regression). Conversely, a larger negative residual value implies better quality because of the lower number of antipatterns in the current observed version. Figure 2 ilustrates how the quality score is estimated from the linear regression computed from our dataset.

Additionally, software quality score at any version is computed as an aggregation of the software quality scores of the preceding versions. The aggregation is performed by computing the average of previous versions scores. Hence, the

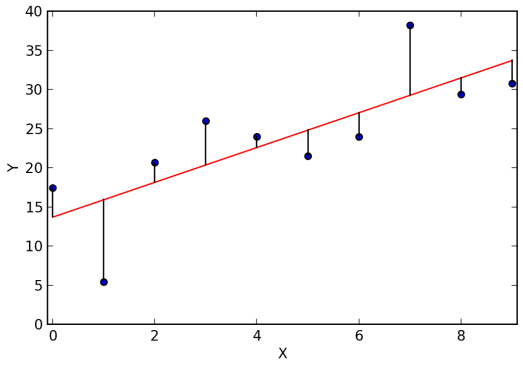

Fig. 2. Example of residuals for linear regression used for quality estimation.

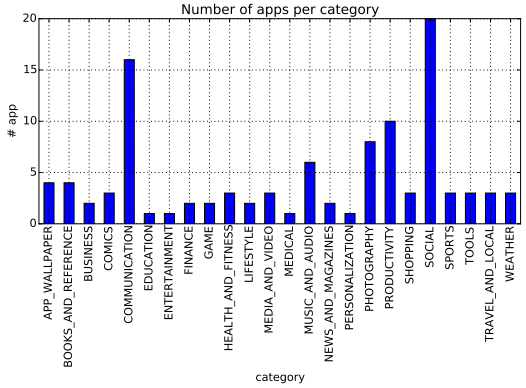

Fig. 3. Distribution of the Android application with regards to their category.

effect of past versions is propagated along the evolution of the application. The scores are computed for each antipattern.

V. Empirical Study

A. Dataset

As mentioned before, this study is based on a large dataset of Android applications downloaded from the Google Play Store. The fact that PAPRIKA operates on the applications' binaries allowed us to gather a wide variety of Android applications. The only challenge we had in data collection was to find a sufficient number of versions of applications so that our tracking of software quality makes sense.

We consider 106 Android applications that differ both in internal attributes, such as their size, and external attributes from the perspective of end users. Our tooled approach, PAPRIKA, is used to track software quality of Android applications. To this end, we collected several versions of each applications to form a total of $3,568$ versions, which we used to estimate software quality. These apps were collected between June 2013 and June 2014 from the top 150 of each category of the Google Play Store. Each app has a minimum of 20 versions. The detailed list of versions is located at https://goo.gl/MktCYM. We present the empirical data from different viewpoints, to illustrate the diversity of our dataset.

1) Category: We classify the downloaded applications according to the category they belong to. All the 24 categories of Google Play Store are represented in our dataset. Figure 3 shows the number of applications per category. For example, *Twitter* app belongs to the category *Social* in the Google Play Store. Also, the proportions revealed by Figure 3 are representative of those found in the Store. In particular, we notice that the majority of apps belong to one of four categories: *Social*, *Comunication*, *Productivity* and *Photography*.

2) Ranking: We describe our dataset from the end users rankings viewpoint. The distribution of applications, according to their ranking, is presented in Figure 4. We observe that most of the applications are ranked above 4.0 (around 90%) and values follow a normal distribution. Applications ranking scores are based on a Likert scale and thus, are computed as the mean of ordinale type values. Although, this is not desirable from measurement theory perspective [22], these scores give some insight about the feeling of end-users towards these applications.

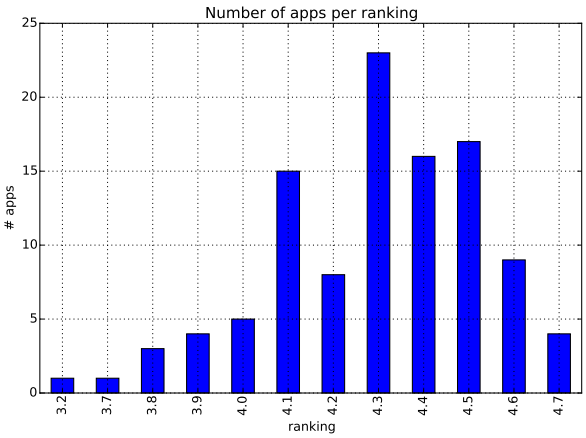

Fig. 4. Distribution of the Android application with regards to their ranking.

3) Download: The number of downloads of Android application is also a good indicator of its popularity among end-users. Our dataset includes applications that vary in number of downloads. The number of downloads is one of the metrics that characterizes android applications in the Google Play Store and is advertised as a token of popularity among end users, just as the ranking score.

An interesting finding about applications downloads is its correlation with the application size. Figure 5 plots the average number of classes in applications versus the number of downloads. This suggests that larger applications are more popular, probably providing more valuable features to the end-users compared to small apps. This observation also supports the claim that mobile applications are becoming complex software systems.

4) Versions: The number of versions of an application may indicate its maturity and the more versions the more stable, reflected in the appreciation of end-users. However, we can only collect information about relative time of versions (their order in time basically) and no information of the absolute time or the interval periods between versions. Therefore, we cannot speculate on the relationship between frequency of releases

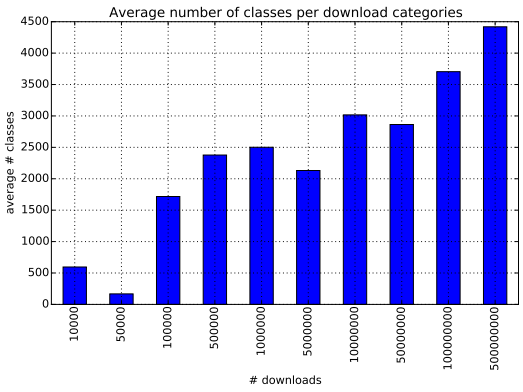

Fig. 5. Average number of classes in applications per number of downloads.

and end users ranking scores.

5) Size: The Android applications analyzed in this study vary in sizes. Among all the versions considered, the sizes in terms of number of classes range from 8 to over 9,000 classes. In general, larger applications have more versions, better ranking scores and are more downloaded. We also calculated the size in terms of special classes, such as number of activities, number of views, etc.

B. Analysis Results

We applied PAPRIKA on all the 106 mobile apps and generated evolution graphs for each application and each antipattern. Overall, we analyzed over 700 graphs representing the evolution of the quality scores discussed in Section IV. As suspected, the evolution of quality varies from one application to another and thus there is no general trend that reveals itself across all applications. This is due to the disparity between mobile apps in terms of antipatterns and size ratios. For instance, CAMERA360 app has 0.75 MIM antipatterns per class on average while TO DO CALENDAR PLANNER app has 2.5 MIM antipatterns per class. However, we could observe relationships across different antipatterns evolution scores and report some evolution trends of mobile apps quality.

1) Relationships between antipatterns: Our estimation of quality is based on the correlation between software antipatterns and software entities. Some antipatterns have better correlation with the number classes in mobile apps, such a *BLOB*, others are more closely related to the number of views (*e.g.*, *UIO*), the number of inner classes (*e.g.*, *LIC*) or the number of brodcast receivers (*e.g.*, *HBR*).

We observe that software evolution graphs based on antipatterns, which correspond to the same type of software entities, are similar. The type of software entity with which an antipattern is correlated comes directly from the antipattern definition. Here, we consider that antipatterns *LM* and *MIM* are correlated to classes although, in reality, they are correlated to the methods of classes. This approximation is supported by the correlation score computed between number of classes and *LM* and *MIM* antipatterns (0.8 for both antipatterns). Figure 6 illustrates the similarity of the evolution score for *BLOB*, *CC*, *LM* and *MIM* antipatterns for the IMO app. Notice that the 4 quality scores drop and rise in the same manner.

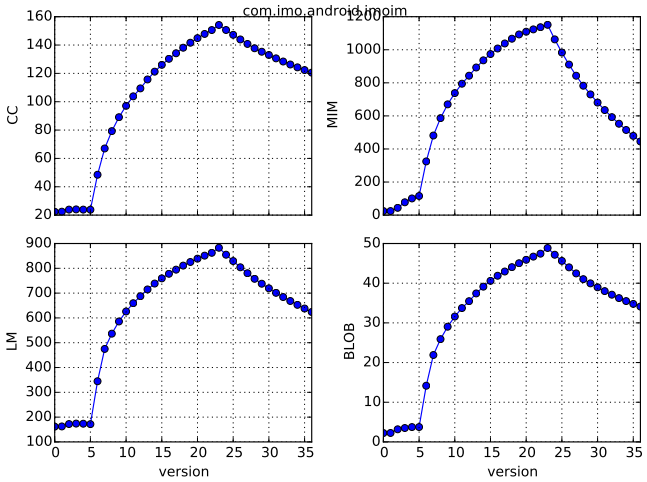

Fig. 6. Software quality scores of IMO app based on *BLOB*, *CC*, *LM* and *MIM* antipatterns.

In some special cases, we observe that the quality score based on *BLOB* and the one based on *CC* or *LM* evolve in opposite manner. Figure 7 shows the evolution of the quality scores based on *BLOB* and on *LM* for RETRO CAMERA app. This situation happens when *BLOB* antipattern "absorbs" *CC* and/or *LM* antipatterns. Often, a *BLOB* class is also a complex class (*CC*) and contains long complex methods (*LM*). Such classes tend to reduce the complexity of surrounding classes (*CC*) as well as reducing the length of their methods (*LM*).

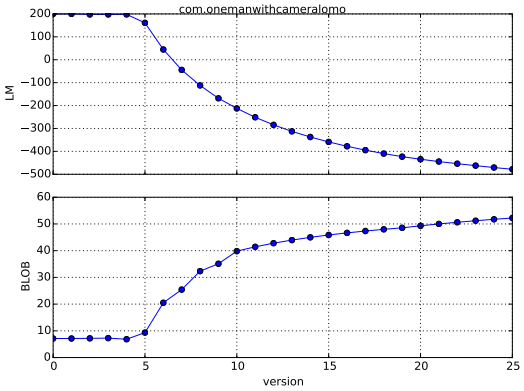

Fig. 7. Software quality scores of RETRO CAMERA app based on *BLOB* and *LM* antipatterns.

In the case of quality scores based on LIC, UIO and HBR, the evolution trends are different. These antipatterns are correlated to inner classes, views and broadcast receivers, which are special types of classes. They heavily depend on the mobile app implementation and hence have different evolutionary paths than the rest of the application. In particular, the number of views and the number broadcast receivers do not necessarily change during large periods of the evolution.

2) Quality Evolution Trends: From the analysis of software quality evolution graphs of the apps and antipatterns considered in our study, we have established 5 major quality evolution trends to answer RQ1 : Can we infer software quality evolution trends across different versions of Android applications?

A) Constant Decline: In general, application size increases along evolutions and new antipatterns are introduced. This evolution trend implies that no action has been done to fix the introduced antipatterns. Hence, quality declines because new antipatterns are introduced without fixing older ones. This is the most encountered quality evolution trend and can be present for either part of the application's evolution or for the entire evolution. Figure 8 shows the constant decline in quality based on LM antipattern.

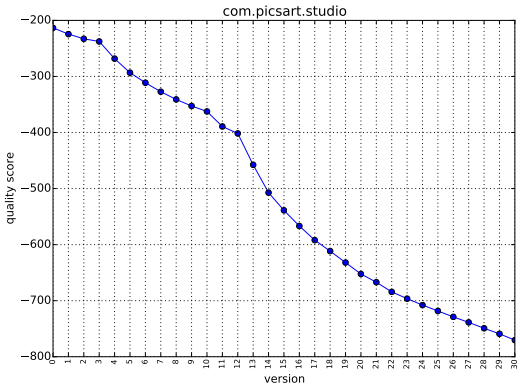

Fig. 8. Software quality scores of PICSART app based on LM antipattern.

B) Constant Rise: Despite the fact that application size evolves along evolutions, there are situations where its evolution seems controlled and less arbitrary. Such situations where development teams follow rigorous programming standards and where software evolution is well structured, exhibit a constant increase in quality over time regardless of changes to applications size. For example, FLIPBOARD app size is changed twice in the 26 considered versions, on versions 19 and 24, but it remains constant otherwise. Figure 9 depicts the constant rise in software quality.

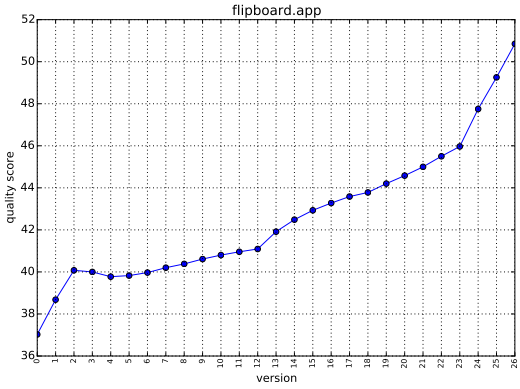

Fig. 9. Software quality scores of FLIPBOARD app based on CC antipattern.

C) Stability: This evolution trend represents periods where software quality is constant. The ratio of introduced antipatterns versus application size is comparable along several consecutive versions. Usually, this trend appears during a finite period, such as in the first 10 versions of FIREFOX app in the LIC based quality shown in Figure 10.

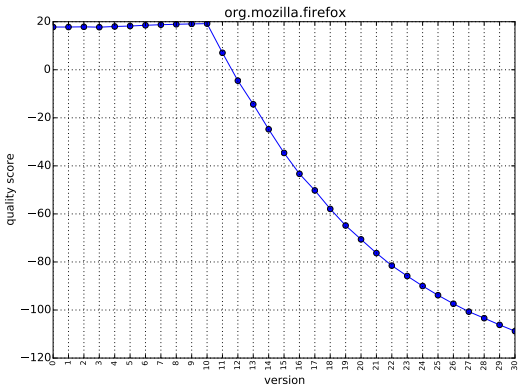

Fig. 10. Software quality scores of FIREFOX app based on LIC antipattern.

D) Sudden Decline: There are cases where the quality score drops abruptly at a particular version. It indicates a turning point in the evolution of the mobile app and is accompanied by a large variation in application size. This drop in quality at a certain version is propagated to the following versions and quality remains low as shown in Figure 11. The rapid decline in quality score of EVERNOTE app, happens in versions 5 and 6, then quality stabilizes, but stays low until version 10 to start a constant rise trend.

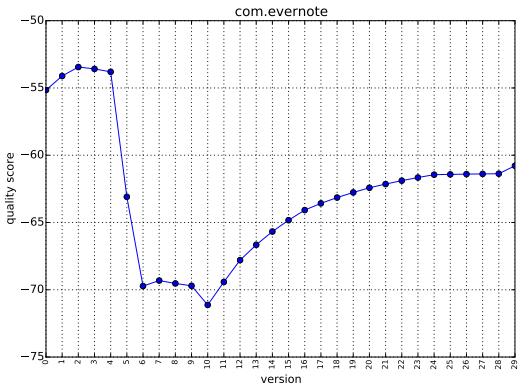

Fig. 11. Software quality scores of EVERNOTE app based on BLOB antipattern.

E) Sudden Rise: It represents the inverse of the previous trend (D) and is characterized by a rapid increase in quality score. Figure 12 illustrates the sudden rise trend in LIC antipattern-based quality of SKYPE app at version 5. Similarly, we observe the propagation of the good quality to successive versions (constant good quality). The rapid increase in quality occurs at the same time of major application size change. This suggests that developers perform refactorings to improve the code structure and enforce solid programming principles, which reflects on quality.

C. Case Study: Twitter

This section presents the quality evolution analysis of TWITTER app as an example of how we can utilize our tooled approach, PAPRIKA, to answer RQ2 : How does software quality evolve in Android applications?

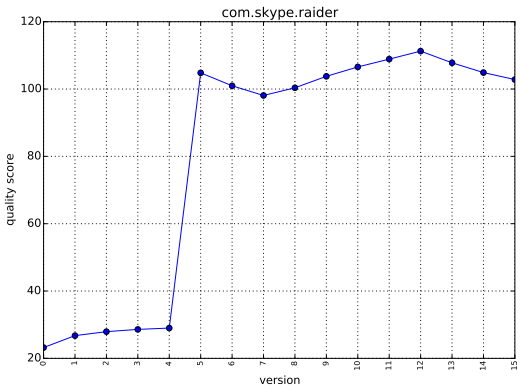

Fig. 12. Software quality scores of SKYPE app based on LIC antipattern.

TWITTER app is given a ranking score of 4.1 by end users. Also, it has been downloaded over $100M$ times, which indicates its high popularity. One question worth asking: how does TWITTER quality evolved over time? To this end, we applied PAPRIKA on the 75 versions of the app collected.

On the one hand, we investigate the changes in terms of number of classes that TWITTER app undergoes during its evolution. Figure 13 plots the number of classes per version. From the size information, we observe several situations of interest, which we will use throughout this case study. First, there is a large addition of classes from version 9 to 12 (from about 530 to over 2,800). Second, the app size constantly rises from versions 19 to 47 and from versions 48 to 74. However, there is a size drop in the middle of this growth in both cases (versions 31 and 59), which suggests minor refactorings. Finally, TWITTER app size declines drastically at version 48 (from about 4,700 to below 3,000). This implies that major refactorings were performed at version 48.

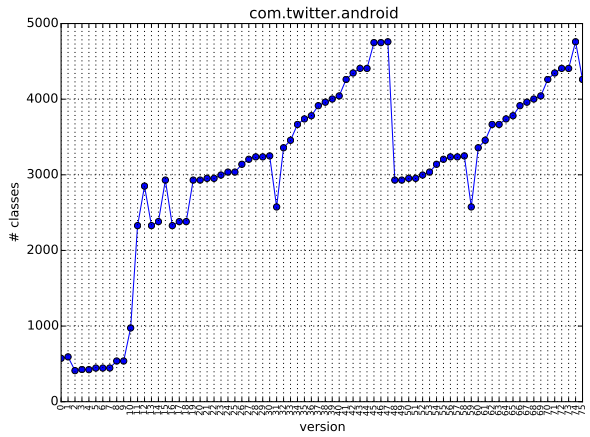

Fig. 13. Size changes of TWITTER app during evolution.

On the other hand, we analyze the evolution of the quality scores computed by PAPRIKA. For instance, Figure 14 presents the CC antipattern-based quality score per version. We focus on what happens to quality at the situations of interest of app size explained before. First, the large addition of classes to the app after version 9 corresponds to a rise in quality, which indicates that new additions occurred with better design that improved quality. This amelioration continues until version 31 when quality starts a constant decline trend. This coincides with the minor refactorings performed at that version as shown in Figure 13. The same pattern happens again at version 59, which is consistent with the app size situations. Finally, the sudden drop in size at version 48 enhanced the quality. This indicates that many antipatterns problems were resolved while doing these major refactorings. The quality keeps rising until the following refactorings at version 59 as mentioned before.

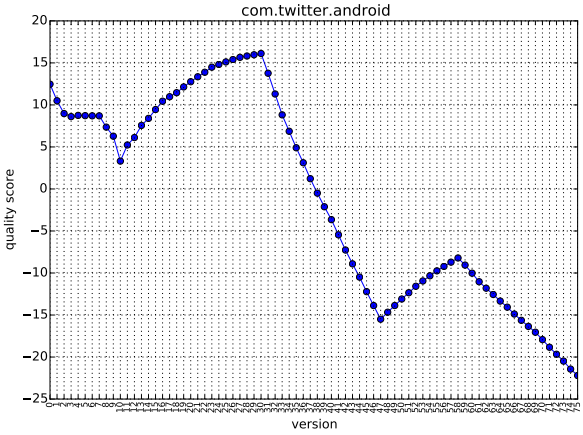

Fig. 14. Software quality scores of TWITTER app based on CC antipattern.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduced PAPRIKA, a tooled approach to monitor the evolution of mobile apps quality based on antipatterns. The process is fully automatic and takes as input mobile apps in the form of Android application packages. PAPRIKA is robust to code obfuscation and identifies antipatterns from applications' bytecode. The antipattern detection is based on software metrics computed by the tool. We consider 3 Object Oriented antipatterns and 4 Android antipatterns. The detected antipatterns are utilized in the evaluation of mobile apps quality. First, we empirically compute a baseline of software quality from the large set of mobile applications collected. Then, the quality of a mobile app is estimated as the deviation from the baseline. In this manner, we analyze the evolution of a mobile quality by propagating the quality at a particular version to its subsequent versions. We believe that our tooled approach could be useful for both Android app developers and App Store providers since PAPRIKA can help them to evaluate the quality of one or thousands of app versions. As future work, we intend to analyze the evolution of external attributes of mobile apps and investigate their potential correlation between mobile apps quality. An external attribute of mobile apps could be the end users impressions. We would use sentiment analysis [24] to track the evolution of end users feelings towards apps along with apps quality evolution.

Acknowledgments

The authors thank Kevin Allix and Jacques Klein from the University of Luxembourg for their help with the dataset. This study is supported by NSERC and FRQNT research grants.

REFERENCES

[1] Android Performance Patterns: Overdraw, Cliprect, QuickReject. https://youtu.be/vkTn3Ule4Ps?list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE. [Online; accessed May-2015].

[2] Android Performance Tips. http://developer.android.com/training/articles/perf-tips.html. [Online; accessed May-2015].

[3] Android studio. https://developer.android.com/sdk/installing/studio.html. [Online; accessed May-2015].

[4] AntiPattern: freezing a UI with Broadcast Receiver. http://gmariotti.blogspot.fr/2013/02/antipattern-freezing-ui-with-broadcast.html. [Online; accessed May-2015].

[5] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. [Online; accessed May-2015].

[6] Java platform, micro edition (java me). http://www.oracle.com/technetwork/java/embedded/javame/index.html. [Online; accessed May-2015].

[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 May-2015].

[9] CYPHER. http://neo4j.com/developer/cypher-query-language. [Online; accessed May-2015].

[10] NEO4J. http://neo4j.com. [Online; accessed May-2015].

[11] Tools to work with android .dex and java .class files. https://code.google.com/p/dex2jar. [Online; accessed May-2015].

[12] 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.

[13] E. Barry, C. Kemerer, and S. Slaughter. On the uniformity of software evolution patterns. In *Proc. of the International Conference on Software Engineering*, pages 106–113, May 2003.

[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] O. Benomar, H. Abdeen, H. Sahraoui, P. Poulin, and M. A. Saied. Detection of Software Evolution Phases based on Developement Activities. In *Proc. of the 23rd International Conference on Program Comprhension*, page To appear. IEEE, 2015.

[17] 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.

[18] M. Brylski. Android Smells Catalogue. http://www.modelrefactoring.org/smell_catalog, 2013. [Online; accessed May-2015].

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

[20] N. Drouin, M. Badri, and F. Tour. Analyzing Software Quality Evolution using Metrics: An Empirical Study on Open Source Software. *Journal of Software*, 8(10), 2013.

[21] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX security symposium*, volume 2, page 2, 2011.

[22] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

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

[24] E. Guzman and W. Maalej. How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews. In *Proc of the 22nd IEEE International Requirements Engineering Conference (RE)*, pages 153–162, Aug 2014.

[25] 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.

[26] F. Khomh, M. Di Penta, and Y. Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *16th Working Conference on Reverse Engineering (WCRE'09)*, pages 75–84, Oct 2009.

[27] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.

[28] 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.

[29] 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.

[30] 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 May-2015].

[31] 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.

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

[33] 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.

[34] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, Jan 2010.

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

[36] A. Murgia, G. Concas, S. Pinna, R. Tonelli, and I. Turnu. Empirical study of software quality evolution in open source projects using agile practices. *CoRR*, abs/0905.3287, 2009.

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

[38] D. L. Parnas. Software Aging. In *Proc. of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[39] J. Reimann, M. Brylski, and U. Amann. A Tool-Supported Quality Smell Catalogue For Android Developers, 2014.

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

[41] 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.

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

[43] 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.

[44] 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.

[45] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proc. of the 37th International Conference on Software Engineering*, page To appear. IEEE/ACM, 2015.

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

[47] 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.

[48] E. Van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. In *Proc. of the 9th Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 97–, Washington, DC, USA, 2002. IEEE Computer Society.

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

[50] G. Xie, J. Chen, and I. Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. In *IEEE International Conference on Software Maintenance (ICSM'09).*, pages 51–60, Sept 2009.

[51] Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. In *Intl. Conf. Softw. Maint.*, pages 242–251. IEEE, 2004.

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

[53] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, Sept 2012.

[54] A. Yamashita and L. Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proc. of the 2013 International Conference on Software Engineering*, ICSE '13, pages 682–691, Piscataway, NJ, USA, 2013. IEEE Press.

[55] H. Zhang and S. Kim. Monitoring Software Quality Evolution for Defects. *Software, IEEE*, 27(4):58–64, July 2010.

[56] T. Zhu, Y. Wu, X. Peng, Z. Xing, and W. Zhao. Monitoring software quality evolution by analyzing deviation trends of modularity views. In *Proc. of 18th Working Conference on Reverse Engineering*, pages 229–238, Oct 2011.