Lightweight Detection of Android-Specific Code Smells: The aDoctor Project

Fabio Palomba[1,2], Dario Di Nucci[2], Annibale Panichella[3], Andy Zaidman[1], Andrea De Lucia[2]
[1]Delft University of Technology — [2]University of Salerno — [3]University of Luxembourg
f.palomba@tudelft.nl, ddinucci@unisa.it, annibale.panichella@uni.lu, a.e.zaidman@tudelft.nl, adelucia@unisa.it

***Abstract*—Code smells are symptoms of poor design solutions applied by programmers during the development of software systems. While the research community devoted a lot of effort to studying and devising approaches for detecting the traditional code smells defined by Fowler, little knowledge and support is available for an emerging category of Mobile app code smells. Recently, Reimann *et al.* proposed a new catalogue of Android-specific code smells that may be a threat for the maintainability and the efficiency of Android applications. However, current tools working in the context of Mobile apps provide limited support and, more importantly, are not available for developers interested in monitoring the quality of their apps. To overcome these limitations, we propose a fully automated tool, coined ADOCTOR, able to identify 15 Android-specific code smells from the catalogue by Reimann *et al*. An empirical study conducted on the source code of 18 Android applications reveals that the proposed tool reaches, on average, 98% of precision and 98% of recall. We made ADOCTOR publicly available.**

***Index Terms*—Android-specific Code Smells; Detection Tool; Empirical Study;**

I. INTRODUCTION

During software maintenance and evolution, a software system undergoes several changes to be adapted to new contexts or to be fixed with regard to urgent bugs [1]. In such a scenario, developers need to manage the complexity of changes as soon as possible in order to meet the unavoidable time constraints, possibly adopting sub-optimal design choices leading to the introduction of so-called *technical debt* [2], *i.e., "not-quite-right"* code that programmers write to meet a deadline or to deliver the software to the market in the shortest time possible.

One noticeable factor contributing to technical debts are bad code smells (shortly "code smells" or simply "smells") originally defined by Fowler, *i.e.,* symptoms of poor design or implementation choices applied by programmers during the development of a software system [3].

Researchers and practitioners widely recognized code smells as a harmful source of maintenance issues [4], [5], [6], [7], which result in a lower productivity [8], [9] and higher rework [10], [11] for developers. For these reasons, researchers have been particularly active in the definition of techniques for detecting code smells [12], [13], [14], [15], as well as in the understanding of the effects of such code smells on non-functional attributes of source code [4], [5], [10], [16].

While the main focus of previous research was on the analysis of standard applications, little effort has been devoted to mobile apps [17]. In this context, a set of new peculiar bad programming practices of Android developers has been defined by Reimann *et al.* [18]. These Android-specific smells may threat several non-functional attributes of mobile apps, such as security, data integrity, and source code quality [18]. As highlighted by Hetch *et al.* [19], these type of smells can also lead to performance issues.

The aforementioned reasons highlight the need of having specialized detectors that identify code smells in Mobile apps. Hetch *et al.* [20] first faced the problem by devising PAPRIKA, a code smell detector for Android apps. However, the tool is able to detect a limited number of the Android-specific code smells defined by Reimann *et al.* (just 4 out of the total 30), and is not publicly available.

In this paper we introduce ADOCTOR (ANDROID Code smell detecTOR), a novel code smell detector that identifies 15 Android-specific code smells. The tool exploits the Abstract Syntax Tree of the source code and navigates it by applying detection rules based on the exact definitions of the smells provided by Reimann *et al.* [18]. We also conducted an empirical study to evaluate the overall ability of our tool in recommending portions of code affected by a design flaw. In particular, we ran ADOCTOR against the source code of 18 Android apps and compared the set of candidate code smells given by the tool with a manually-built oracle. According to the results, ADOCTOR is able to suggest code smells with an average precision of 98% and an average recall of 98%. The tool has been also employed in the evaluation of the impact of a subset of Android-specific code smells (*i.e.,* the ones supposed to be related to energy efficiency) on the energy consumption of Android apps [21].

Tool and Data Replication. Our detector, as well as the executable file and all the data used in the experiment are available on the ADOCTOR website [22].

Structure of the paper. Section II describes the detection rules and the underlying architecture of the proposed tool, while Section III reports the design and results of the empirical study conducted to measure the performances of ADOCTOR. Finally, Section IV concludes the paper.

II. THE ADOCTOR PROJECT

The ADOCTOR project is built on top of the Eclipse Java Development Toolkit (JDT)[1]. While the catalogue by Reimann *et al.* [18] proposes a set of 30 design flaws related to both

[1]http://www.eclipse.org/jdt/

implementation and UI design, in this demo we focus our attention solely on the smells characterizing a problem in the source code. Therefore, our tool supports the identification of 15 Android-specific code smells. In the following, we present the detection rules applied by ADOCTOR, as well as the underlying architecture supporting the identification.

A. Detecting Android-specific Code Smells

This section reports, the definition of each smell supported by ADOCTOR as well as the rule followed for its detection.

Data Transmission Without Compression (DTWC). The smell arises when a method transmits a file over a network infrastructure without compressing it, causing an overhead of communication [18]. ADOCTOR detects the smell if a method performs an `Http` request involving an instance of the class `File` without using a compression library such as ZLIB[2] or the APACHE HTTP CLIENT[3].

Debuggable Release (DR). In Android, the attribute `android:debuggable` of the `AndroidManifest` file is set during the development for debugging an app. Leaving the attribute `true` when the app is released is a major security threat since every external app can have full access to the source code. In this case, the detector simply parses the `AndroidManifest` file looking for the `android:debuggable` properties. If it is explicitly set to `true`, the smell is detected.

Durable Wakelock (DW). A Wakelock is the mechanism allowing an app to keep the device on in order to complete a task. However, when such task is completed, the lock should be released to reduce battery drain [18]. In Android, the class `PowerManager.WakeLock` is in charge to define the methods to acquire and release the lock. If a method using an instance of the class `WakeLock` acquires the lock without calling the `release`, a smell is identified.

Inefficient Data Format and Parser (IDFP). When analyzing XML or JSON files, the use of `TreeParser` slows down the app, and thus it should be avoided and replaced with other more efficient parsers (*e.g.,* `StreamParser`) [18]. In this case, ADOCTOR identifies the smell by evaluating whether a method uses the `TreeParser` class.

Inefficient Data Structure (IDS). The mapping from an integer to an object through the use of a `HashMap<Integer, Object>` is slow, and should be replaced by other efficient data structures, such as the `SparseArray` [18]. Therefore, methods using an instance of `HashMap<Integer, Object>` are identified by ADOCTOR as smelly.

Inefficient SQL Query (ISQLQ). In Android, the use of a `SQL` query is discouraged as it introduces overhead, while other solutions should be preferred (*e.g.,* using webservices) [18]. If a method defines a `JDBC` connection and sends an `SQL` query to a remote server, the smell is identified.

[2]http://www.zlib.net

[3]https://hc.apache.org

Internal Getter and Setter (IGS). In Android development, the use of accessors methods (*i.e.,* getters and setters) are expensive and, thus, internal fields should be accessed directly [18]. All the methods accessing other objects using getters and/or setters are identified by ADOCTOR as affected by this smell.

Leaking Inner Class (LIC). Reimann *et al.* defined this smell as a *"non-static nested class holding a reference to the outer class"* [18]. This could lead to a memory leak. Analyzing the files having nested classes, ADOCTOR identifies this smell by counting the relationships that the outer class has with the nested classes. If the counter is higher than 1, a *Leaking Inner Class* is detected.

Leaking Thread (LT). In Android programming a thread is a garbage collector (GC) root. The GC does not collect the root objects and, therefore, if a thread is not adequately stopped it can remain in memory for all the execution of the application, causing an abuse of the memory of the app. If an `Activity` starts a thread and does not stop it this is considered a design flaw [18]. ADOCTOR detects this smell if a method of an `Activity` class starts a thread without stopping it through the `stop` method.

Member Ignoring Method (MIM). Non-static methods that do not access any internal properties of the class should be made `static` in order to increase their efficiency [18]. In this case, our detector exploits the references of a method, and if it does not reference any internal fields, a smell is identified.

No Low Memory Resolver (NLMR). An Android developer can define the behavior of the app when it runs in background overriding the method `Activity.onLowMemory` [18]. This method should be used to clean caches or unnecessary resources. If it is not defined, the app can lead to abnormal memory use. Consequently, if a mobile app does not contain the method `onLowMemory`, ADOCTOR detects a smell.

Public Data (PD). This smell arises when private data is kept in a store that is publicly accessible by other applications, possibly threatening the security of the app [18]. In Android, this is done by setting the context of the class as private, using the `Context.MODE_PRIVATE` command. Classes that do not define the context or define the context as non-private are detected by ADOCTOR as smelly.

Rigid Alarm Manager (RAM). The `AlarmManager` class allows to execute operations at specific moments. Obviously, an Alarm Manager-triggered operation wakes-up the phone, possibly threatening the energy and memory efficiency of the app. It is recommended to use the `AlarmManager.setInexactRepeating` method, which ensures that the system is able to bundle several updates together [18]. Therefore, a code smell is identified by our detector if a class using an instance of `AlarmManager` does not define the method `setInexactRepeating`.

Slow Loop (SL). The standard version of the `for` loop is slower than the `for-each` loop [18]. Therefore, Android

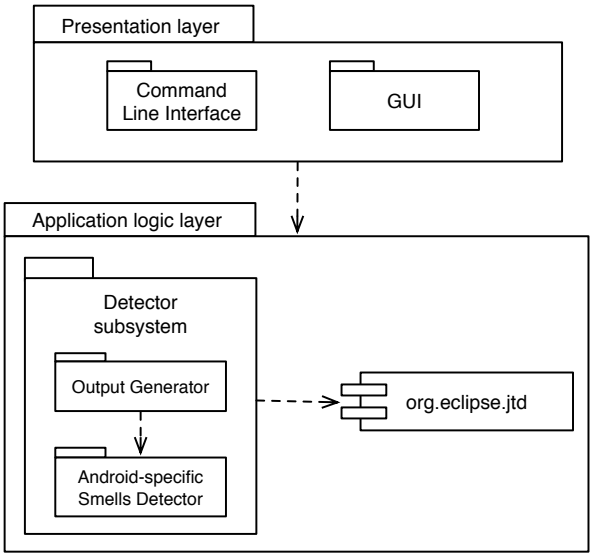

Fig. 1: ADOCTOR Architecture

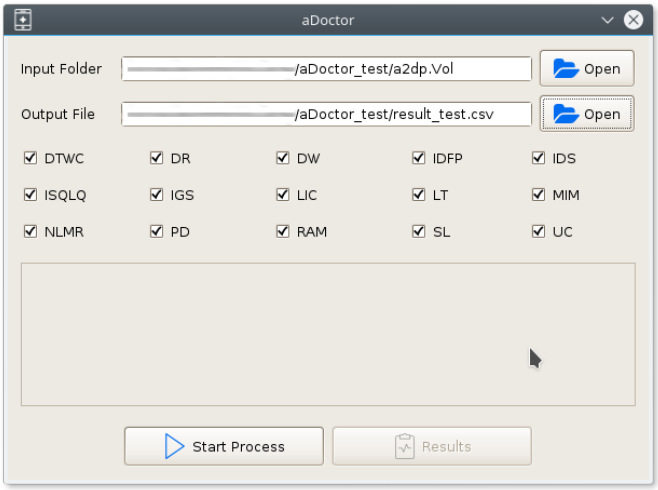

Fig. 2: ADOCTOR: Configuration View

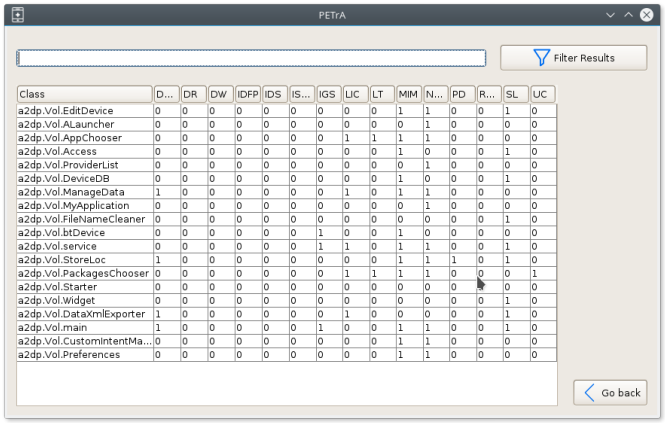

Fig. 3: ADOCTOR: Results View

developers should always use an enhanced version of the loop to improve the efficiency of the app. Our detector identifies smelly instances as all the methods using the `for` loop.

Unclosed Closable (UC). A class that implements the `java.io.Closeable` interface is supposed to invoke the `close` method to release resources that an object is holding [18]. If the class does not call such a method, ADOCTOR identifies a smell.

B. aDoctor Architecture and Its Inner-Working

Figure 1 depicts the architecture of ADOCTOR. The *Presentation Layer* is composed of the classes implementing two types of user interfaces, *i.e.,* command line and graphical user interfaces. The tool is executable via command line through the following command:

```
java -cp aDoctor.jar
RunAndroidSmellDetection <project-path>
<output-path> <smells>
```

where `<project-path>` is a string representing the path to the directory containing the source code of the Android app to analyze, `<output-path>` is the path to the file where the code smell candidates will be printed, and `<smells>` is a string defining the code smells to analyze. This type of interface allows our tool to be run programmatically and be employed in mining software repository studies. In addition, we provide a graphical user interface.

The configuration view in Figure 2 allows the software engineer to set the parameters needed for running the analysis, *i.e.,* folder where the project is located and CSV file where to save the candidate smells. Moreover, the software engineer can select the smells that she is interested in. Once the start button is pressed, the computation starts. When completed, the results are shown in a second view, depicted in Figure 3. The candidate smells can be filtered by class name, and in every case the results are saved in the `<output-path>`.

The *Application Logic Layer* is the core of the ADOCTOR project and it contains all the subsystems implementing the detection rules of the Android-specific smells described in the previous section, as well as the classes that output the candidate smells. The layer relies on the Eclipse JDT APIs in order to (i) extract the Abstract Syntax Tree of the source classes contained in the app under analysis, and (ii) navigate the Abstract Syntax Tree and compute the detection rules. The single smell detection rules are implemented as separate classes of the `Android-specific Smells Detector` subsystem. As for the `Output generator` subsystem, it is responsible for executing the detection process calling the classes of the `Android-specific Smells Detector` subsystem (see the dependence between the two subsystems in Figure 1), and output of the candidate code smells found. Specifically, the output is represented by a CSV file where:

- each line of the CSV file represents a code element of the analyzed app;
- the first column of each line specifies the granularity of the code element (*i.e.,* class or method);
- columns from #2 to #n in each line report a boolean value indicating the presence/absence of the 15 Android-specific code smells (*e.g.,* column #2 will be `true` if a *Data Transmission Without Compression* has been detected, `false` otherwise).

III. Evaluation

The empirical study has the *goal* to quantify the ability of aDoctor in recommending portions of source code affected by a design flaw, with the *purpose* of investigating its effectiveness during the detection of Android-specific code smells in Android applications. Specifically, our research question is the following:

> **RQ**$_1$: *What are the precision and recall scores of* aDoctor *in detecting Android-specific code smells?*

The *context* of the study consists of a set of 18 Android apps belonging to different categories, and having different scope and size. Due to space limitations, the complete list of apps considered in the study is available on the aDoctor website [22].

A. Empirical Study Design

To answer **RQ**$_1$ we ran aDoctor on the apps in our context. To evaluate its precision and recall, we needed an oracle reporting the actual code smell instances contained in the considered Android apps. Since there is not an annotated set of Android-specific code smells available in literature, we built our own oracle. To this aim, we asked a Master's student from the University of Salerno to manually analyze the apps taken into account in order to extract the methods affected by each of considered smells. Starting from the definition of the 15 smells, the student manually analyzed the source code of the latest version of the apps, looking for instances of those smells. This process took approximately 180 man-hours of work. Then, a second Master's student (still from the University of Salerno) validated the produced oracle, to verify that all affected code components identified by the first student were correct. Just 14 of the instances classified as smelly by the first student were classified as false positives by the second student. After a discussion performed between the two students, 8 of these 14 instances were definitively classified as false positives (and, therefore, removed from the oracle). Note that we cannot ensure about the completeness of the oracle. Moreover, to avoid bias the students were not aware of the experimental goals and of specific algorithms used by aDoctor to identify smells. The oracle defined is available on the aDoctor website.

Once the set of actual smells was ready and the set of candidate smells identified by aDoctor was available, we compared the two sets using two widely adopted Information Retrieval (IR) metrics, *i.e.,* precision and recall [23].

To have an aggregate indicator of precision and recall, we also report the F-measure, defined as the harmonic mean of precision and recall.

Due to space limitations, we report the overall precision and recall obtained analyzing each smell type on the 18 apps. The results achieved on the single apps are available on the aDoctor website [22].

TABLE I: Performance Of aDoctor On The Apps Object Of The Empirical Study

Code Smell	Precision	Recall	F-Measure
DTWC	87%	89%	88%
DR	100%	100%	100%
DW	100%	100%	100%
IDFP	100%	100%	100%
IDS	100%	100%	100%
ISQLQ	85%	88%	86%
IGS	100%	100%	100%
LIC	100%	100%	100%
LT	100%	100%	100%
MIM	100%	100%	100%
NLMR	100%	100%	100%
PD	100%	100%	100%
RAM	100%	100%	100%
SL	100%	100%	100%
UC	100%	100%	100%
Average	98%	98%	98%

B. Analysis of the Results

Over all the 18 apps considered, aDoctor detects 1,444 code smell instances (on average, 80 per app). The most frequent ones are the *Member Ignoring Method* (467 instances), Slow Loop (378 instances), and Data Transmission Without Compression (266 instances) smells. Since the analyzed apps contain on average 121 classes, our results reveal that the Android-specific smells are quite diffused and, thus, the phenomenon is worth investigating. Note that the complete results on the distribution of code smells are available on the aDoctor website [22].

Table I reports, for each Android-specific smell, the results achieved over the set of 18 apps taken into account. The results clearly show that aDoctor is able to correctly identify almost all the code smell instances present in the Android apps. Only in two cases the results do not reach 100% precision and recall, *i.e., Data Transmission Without Compression* and *Inefficient SQL Query*. We manually analyzed these cases in order to understand the reasons behind the results, finding that the detector missed some instances because the classes affected by such smells used different compression libraries with respect to the ones considered in the detection rules. Indeed, both smells are related to the communication with remote servers. To do so, Android apps usually rely on some widely spread libraries such as Zlib or the Apache HTTP Client. However, there are some cases where other libraries are employed and, therefore, the detector is not able to correctly identify the design flaws. For instance, aDoctors iden-

tifies a false positive *Data Transmission Without Compression* instance in the class `AndroidomaticKeyerActivity`, belonging to the package `com.templaro.opsiz.aka` of the ANDROIDOMATIC KEYER app. This class relies on the SILICOMPRESSOR library[4] to compress files before sending them, but ADOCTOR does not recognize the compression because the method calls done by the class do not refer to the libraries it consider.

While in this case ADOCTOR fails in the identification of the smell, it is worth noting that we configured our detector in order to work with the most common libraries used by Android developers. Moreover, the issue reveals a potential way to improve the detection accuracy of the tool. Indeed, as the support to other libraries will be implemented, the performances of the tool will be higher.

The discussion is different for the other smells, since ADOCTOR always reaches 100% of F-Measure. This is due to the fact that the detection rules described in Section II are effective in capturing all the small programming issues applied by Mobile developers. In conclusion, we can affirm that the proposed tool is efficient in terms of accuracy of the recommendations.

IV. DEMO REMARKS

In this demo we presented ADOCTOR, a tool supporting the detection of 15 Android-specific code smells from the catalogue by Reimann *et al.* [18]. To identify design flaws, the tool navigates the Abstract Syntax Tree of a class and applies detection rules implementing the exact definitions provided by Reimann *et al.*

We conducted an empirical study involving 18 Android apps to validate the proposed tool. The results showed an average precision and recall of 98%, clearly highlighting the ability of our tool to correctly identify design flaws in the source code. For two of the considered smells, *i.e., Data Transmission Without Compression* and *Inefficient SQL Query*, the average F-Measure is slightly lower than the others, but this is due to the fact that sometimes the apps use compression libraries different from the most popular ones.

We plan to integrate the code smell detector in the most common Integrated Development Environment (IDE) used by Android developers, *i.e.,* Android Studio. Moreover, we plan to extend the functionalities of ADOCTOR in order to allow the extraction and the automation of meaningful refactoring operations aimed at removing code smells from the source code.

REFERENCES

[1] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press London, 1985.

[2] W. Cunningham, "The WyCash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[4] https://github.com/Tourenathan-G5organisation/SiliCompressor

[4] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009, pp. 75–84.

[5] 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 Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[6] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 101–110.

[7] 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 *Proceedings of the International Conference on Software Engineering (ICSE) - Volume 1*. IEEE, 2015, pp. 403–414.

[8] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[9] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "Balancing agility and formalism in software engineering," B. Meyer, J. R. Nawrocki, and B. Walter, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266.

[10] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR*. IEEE, 2011, pp. 181–190.

[11] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

[12] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[13] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[14] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, May 2015.

[15] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[16] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*. Raleigh, USA: IEEE, 2016, p. to appear.

[17] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *Transactions on Software Engineering*, p. to appear., 2017.

[18] J. Reimann, M. Brylski, and U. Amann, "A tool-supported quality smell catalogue for android developers," 2014.

[19] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. ACM, 2016, pp. 59–69.

[20] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, "Detecting antipatterns in android apps," in *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on*, May 2015, pp. 148–149.

[21] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," in *Submitted to the Journal of Empirical Software Engineering*, ser. EMSE, 2017.

[22] "adoctor website," http://tinyurl.com/hnm2sla, 2016. [Online]. Available: http://tinyurl.com/hnm2sla

[23] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.