2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)

Detecting Android Smells Using Multi-Objective Genetic Programming

Marouane Kessentini
CIS department
University of Michigan,
Dearborn, USA
marouane@umich.edu

Ali Ouni
CSSE department
UAE University
Al Ain, UAE
ouni@uaeu.ac.ae

***Abstract*—The evolution rate of mobile applications is much higher than regular software applications having shorter release deadlines and smaller code base. Mobile applications tend to be evolved quickly by developers to meet several new customer requirements and fix discovered bugs. However, evolving the existing features and design may introduce bad design practices, also called code smells, which can highly decrease the maintainability and performance of these mobile applications. However, unlike the area of object-oriented software systems, the detection of code smells in mobile applications received a very little of attention. Recent, few studies defined a set of quality metrics for Android applications and proposed a support to manually write a set of rules to detect code smells by combining these quality metrics. However, finding the best combination of metrics and their thresholds to identify code smells is left to the developer as a manual process. In this paper, we propose to automatically generate rules for the detection of code smells in Android applications using a multi-objective genetic programming algorithm (MOGP). The MOGP algorithm aims at finding the best set of rules that cover a set of code smell examples of Android applications based on two conflicting objective functions of precision and recall. We evaluate our approach on 184 Android projects with source code hosted in GitHub. The statistical test of our results show that the generated detection rules identified 10 Android smell types on these mobile applications with an average correctness higher than 82% and an average relevance of 77% based on the feedback of active developers of mobile apps.**

Keywords-Search-based software engineering; Android apps; quality.

I. INTRODUCTION

During the last few years, mobile applications are becoming one of the largest parts of the current software market [14]. Google Play Store already reached more than 51 billion applications downloads and by 2020, users are projected to spend over 100 billion U.S. dollars on mobile applications [15]. One of the key factors of this success is the use of well-known programming languages based on Object-Oriented design principles (OO) including C# and Java.

The design and implementation of mobile applications is different than regular software applications [32]. Most of the mobile applications development is based on the reuse of many libraries having smaller size, in average, than regular software applications [16]. Furthermore, mobile apps always have short release deadlines forcing the developers to focus more on implementing the required features and fixing bugs rather than meeting different quality standards. Thus, several bad design practices, also called code smells, could be introduced by developers during the development of mobile apps and impact their quality negatively.

The presence of code smells in mobile apps may lead to a higher use of the resources such as CPU, battery, memory, etc. However, unlike regular object-oriented applications, the automated detection of code smells on mobile applications did not receive a lot of attention and very few studies are proposed [10]. Most of existing studies are based on a manual detection support of code smells [18]. A catalogue of 30 types of code smells dedicated to Android apps were defined by Reimann et al. [18]. They defined the symptoms associated with these different code smell types and their impact on several quality attributes including performance, maintainability, security, energy consumption, etc. Hetch et al. [10] manually defined a set of detection rules based on these symptoms to detect four Android-specific code smells. The proposed tool, called PAPRIKA, was used to analyze the evolution of the quality of mobile applications 106 Android apps. To summarize, most of existing detection tools are based on defining manually the symptoms of every code smell type in Android apps, then translating them into a combination of quality metrics with their thresholds. However, several limitations could be discussed around existing manual techniques to detect code smells in mobile apps.

First, deciding manually, which metrics to use to formalize a symptom of a code smell is not straightforward. Several possible combinations of metrics could be used with different possible threshold values. Second, several code smell types may have similar symptoms. Third, developers should adapt these rules manually for almost every new mobile app to evaluate. Fourth, there is no consensus about the symptoms to identify the code smells, making the manual procedure to define the rules challenging. Finally, it could be a time-consuming process to define these rules manually especially with the high number of possible quality metrics, and threshold values to consider.

In this paper, we propose an automated approach to generate rules that can detect code smells in Android apps. A rule is a combination of quality metrics with their threshold values to detect a specific type of code smell. The proposed approach takes as input a set of Android-specific code smell examples from multiple apps and uses a multi-objective genetic programming algorithm MOGP [30] to find the best set of rules that covers most of the expected

978-1-5386-2669-6/17 $31.00 © 2017 IEEE
DOI 10.1109/MOBILESoft.2017.29

Android code smells. The process aims at finding a trade-off between the conflicting precision and recall measures of rules coverage of all smell instances in the base of examples.

Developers may select the rules with high precision when they have limited time to fix these Android smells before the next release or they may select those with high recall when enough resources are available to fix most of the detected Android smells and inspect the results. Another alternative for the developer is to select the detection rules presenting the best trade-off between the two objectives of precision and recall. To the best of our knowledge and based on recent surveys [13], the detection of Android code smells was not addressed before using search-based techniques.

We implemented our proposed approach and evaluated it on a set of 184 Android projects with source code hosted in GitHub. The statistical test of our results shows that the generated detection rules identified 8 Android-code smell types on these mobile applications with an average manual correctness higher than 82% and an average relevance of 77% based on the feedback of active developers of Android apps. We have also compared our approach with the rules that are manually defined. The paper also describes a survey with active developers about their opinion of the relevance of identified Android code smells.

The remainder of this paper is structured as follows. Section 2 provides an overview of the related work. Section 3 describes our approach to detect Android code smells while the results obtained from our validation are described and discussed in Section 4. Finally, Section 5 concludes and describes some possible future research directions.

II. TYPE STYLE AND FONTS

In this section, we present the necessary background to understand the proposed contribution and an overview of existing studies, then a summary of existing challenges is described.

A. Background

Quality Metrics. Most of the structural proprieties of software projects are captured through software metrics. These metrics are then used to evaluate the quality of software systems such as those defined by Chidamber and Kemerer [6] including coupling, cohesion, etc. Since mobile apps are developed mainly using object oriented (OO) programming languages, the well-defined OO metrics can be used in the context of evaluating the quality of Android apps using several existing tools such as InFusion [17]. Some recent studies [10] [18] defined new metrics dedicated for mobile applications such as the number of Broadcast Receiver, etc. PAPRIKA [10] is an example of a tool that can be used to extract several Android-specific metrics. Table 1 summarizes the list of 35 quality metrics considered in our experiments. We used this list of metrics since they cover most of quality attributes and could be extracted using existing tools [10] [18].

Android Code Smells. Code smells are defined, in general, as bad design practices that may impact the quality of software systems such as maintainability, understandability, etc. [21][35]. Therefore, the software will become complex, hard to evolve and understand by the developers. The code-smells emerge most of the time during the evolution of a system. Although they do not create failures directly, but may introduce bugs indirectly. In general, they make a system difficult to change, which may in turn introduce bugs.

Code smells are detected using quality metrics that characterize the different symptoms of bad practices. In the OO paradigm, 22 code smells are defined informally by Fowler et al. [21]. Recently, different new types of code smells are defined for Android apps [18] [20]. A catalogue of 30 types of Android-specific code smells was proposed by Reimann et al. [18]. The authors also evaluate the impact of these code smells on different quality attributes such as energy efficiency, security, etc [20]. The authors also proposed a generic refactoring strategy to find these smells and proposed a tool called REFACTORY. We selected the following list of 6 Android-specific code smells and 4 general OO code smells [10]: *Internal Getter/Setter (IGS):* Internal fields are accessed via getters and setters. In Android, virtual methods are expensive. *Member Ignoring Method (MIM):* In Android, when a method does not have access to an object attribute, it is recommended to use a static method. The static method invocations are faster than a dynamic invocation. *HashMap Usage (HMU):* HashMap are supposed to be slow (mapping from an integer to an object). Therefore, creating small multiple HashMap instances can be considered as a code smell. *Leaking Inner Class (LIC)*: non-static inner and anonymous classes are holding a reference to the outer class, whereas static inner classes are not. This may generate memory leak in Android apps. *UI Overdraw (UIO)*: The Android Framework API several methods are proposed to avoid the overdraw of non-modified parts of the UI and improve the overall performance of the app. *Heavy Broadcast Receiver (HBR):* Android broadcast receivers can trigger ANR when they perform heavy and lengthy operations. *Blob:* Blob class, also known as God class, is a class with many attributes and/or methods (features). Blob classes are difficult to evolve and fix. *Feature Envy (FE):* it is defined as a method that invokes extensively methods of another class and has a high coupling and low cohesion. *Spaghetti Code (SC)*: It is a code with a complex and tangled control structure. *Functional Decomposition (FD):* It occurs when a class is designed with the intent of performing a single function (modularity issues).

B. Related Work

1) Object Oriented Code Smells

Existing OO code smells studies can be classified in three categories: manual, interactive and fully-automated. For the first category, Fowler et al. proposed a list of different types of code smells [21]. For every type of code smells, they specified a list of symptoms and a generic list of refactorings to fix it. In [29], another proposed approach is based on violations of design rules and guidelines. This approach consists of analyzing legacy code, specifying frequent design problems as queries and manually locating the occurrences of these problems in a model derived from the source code. Most of the existing manual approaches are

based on extracting the quality metrics from the source code such that the developers can select the parts of the code to inspect by analyzing the metrics values.

To reduce the effort of manually inspecting the quality metrics to identify code-smells, several semi-automated approaches were introduced based on visualization techniques. Kothari et al. [34] proposed a framework to visualize code-smells using different colors based on metrics threshold. In another category of work based on the use quality metrics, Marinescu et al. [31] have proposed a mechanism called "detection strategy" for formulating metrics-based rules that capture deviations from good design principles and heuristics. Detection strategies allow to a maintainer to directly locate classes or methods affected by a design code-smell. Erni et al. [24] introduce the concept of multi-metrics, as an n-tuple of metrics expressing a quality criterion (e.g., modularity). Unfortunately, multi-metrics neither encapsulate metrics in a more abstract construct, nor do they allow a flexible combination of metrics.

TABLE I. LIST OF CONSIDERED QUALITY METRICS

Metric	Description
NCL	#Classes
NI	#Interfaces
NAC	#Abstract Classes
NM	#Methods
DIT	Depth Of Inheritance
NIM	#Implemented Interfaces
NAT	#Attributes
NCD	#Children
COM	Class Complexity
COU	Coupling
COH	Lack of Cohesion
IsA	Is Abstract: Class, Method
IsF	Is Final: Class, Method
IsST	IsStatic : Class, Method
IsN	Is Inner: Class
IsIN	Is Interface: Class
NPR	#Parameters : Method
NDL	#Declared Locals : Method
NIST	#statements
NDCL	#Direct Calls : Method
NCAL	#Callers: Method
CC	Cyclomatic Complexity : McCabe
ISY	Is Synchronized: Method
NACT	#Activities of a Mobile App
NBR	#Broadcast Receivers of a Mobile App
NCP	#Content Providers of a Mobile App
NSER	#Services of a Mobile App
IsACT	Is Activity: Class(Mobile App)
IsAPP	Is Application: Class(Mobile App)
IsBR	Is Broadcast Receiver: Class(Mobile App)
IsCP	Is Content Provider: Class(Mobile App)
IsSER	IsService: Class(Mobile App)
NHM	Number of HashMap instances

Van Emden and Moonen [8] developed one of the first automated code-smells detection tools for Java programs. They developed a prototype code-smells browser that detects and visualizes code-smells in Java programs. Mantyla studied the manner of how developers detect and analyze code-smells [27]. Previous empirical studies have analyzed the impact of code-smells on different software maintainability factors [32] [21]. Yamashita et al. [1] show that the different types of code-smells can cover different maintainability factors.

2) Mobile Apps Code Smells

Several approaches proposed to use existing OO code smells detection tools for Android apps. Linares-Vasquez et al. [19] used an existing OO code smells detection tool, called DECOR, to detect around eighteen code smells, inspired from OO antipatterns, on 1,343 Android apps. The authors evaluated the impact of mobile code smells on the number of bugs found in these apps. In another study [10], an existing tool for the detection of OO code, PMD, were used to detect antipatterns in Android apps and identified the most frequent ones, however, none of the specific Android smells were considered in this study.

A catalogue of several antipatterns was proposed by Reimann et al. [18] that covers most of the quality issues of Android apps. The authors evaluated also the impact of these code smells on the energy/memory consumption, maintainability, etc. They proposed a generic set of symptoms of every type of Android code smell but no tool is developed to automate the detection of these smells. Recently, a tool for the detection of code smells in Android apps was introduced by Hecht et al. [10]. They can detect using their tool eight different types of code smells including four Android specific antipatterns. The authors found that these four types of Android code smells appear much more frequently than regular OO antipatterns. However, it is hard to generalize these obtained results since only 15 apps were used.

In another work, Tufano et al. [22] studied the evolution of 70 Android apps and they found that most of the code smells were introduced when the file was created. A survey with more than 70 developers about the importance of Android code smells were conducted by Delchev and Harun [23]. The authors found that Long Methods and Shotgun Surgery are the most important code smells to fix based on the opinions of Android apps developers.

3) Search Based Software Engineering

Software engineering could be considered a search problem to find an optimal or near-optimal solution [11][12][13][25]. This search is often complex with many competing constraints, and conflicting objectives. Recently, an emerging software engineering area, called Search-Based Software Engineering (SBSE) [13][33][28][26], is rapidly growing. SBSE is a software development practice that focuses on formulating software engineering problems as optimization problems based on the use of meta-heuristic techniques to automate the search for near optimal solutions. SBSE was successfully applied to a wide range of problems covering the whole software life cycle [9][1][2][3][4][5][7][36]. In this paper, we use, for the first time, an SBSE technique to address the problem of antipatterns detection in Android applications.

4) Challenges Summary

As discussed in the previous section, most existing studies used existing OO code smells detection tools to find antipatterns in Android apps. However, the context of implementing Android apps is different than regular other

OO applications. For example, Android apps tend to be smaller in size than other regular types of OO applications, and developed faster often by inexperienced developers. Everyone can develop his own app and release it on the apps store. Thus, the systematic application of generic detection rules, based on threshold metrics defined for regular applications, may generate a high rate of false positive antipatterns. Furthermore, there is no clear consensus about the symptoms of Android antipatterns since they are not widely studied yet like OO code smells. Thus, it is maybe difficult to manually translate the symptoms into rules.

Another important challenge is the ability to manually find generic rules that could be applied to different Android apps. In fact, developers may have different views about the best programming practices and that may impact their way to manually define the thresholds for the metrics. In addition to the above-mentioned limitations, existing work that try to automate the detection of code smells propose only one set of rules as output to the developer. However, different possible good sets of detection rules based on different developer preferences could be generated. Then, the developer may select the best set of rules that translates his view of best design/implementation practices. To address these challenges, we propose in this paper a new way to detect Android code smells by automatically generating detection rules using multi-objective Genetic Programming [30]. The next section describes the proposed technique.

III. SEARCH BASED DETECTION OF ANDROID CODE SMELLS

In this section, we describe the overview of our approach and the different adaption steps of multi-objective search to our problem.

A. Approach Overview

Fig. 1 provides an overview of the proposed approach to generate Android smells detection rules. In this work, we start from the observation that it is easier for developers to provide and identify examples of Android code smells rather than manually writing generic detection rules [32]. Several of these code smell examples can be found in the review reports of Android apps [16]. The goal of the proposed approach is to generate a set of rules, as a combination of quality metrics that cover as much as possible a set of antipattern examples extracted from various Android apps.

As described in Fig. 1, the first component of our approach consists of extracting the source code of a set of Android apps to get the value of the metrics described in Table 1 and collecting a set of examples of Android smells from the code review reports and developers. Then, the second component takes these inputs to generate a set of detection rules satisfying as much as possible two objectives. The first objective is to find the best set of rules that maximizes the detection of correct Android smells over the list of suggested ones to check (precision). The second objective maximizes the coverage of expected Android smells from the base of examples of multiple projects over the actual list of detected smells (recall).

The multi-objective Genetic Programming Algorithm is the main component of our approach. It starts first by generating a set of solutions. Every solution is composed by a set of rules. The number of rules per solution corresponds to the number of different Android smell types to detect (10 rules in our case). The rules are combination of metrics and logic operators (AND, OR) as explained the next section. They are based on the following template: IF (Combination of metrics and their thresholds) THEN Android code smell type. All the generated solutions in the population are evaluated using the two objectives then ranked as described in the previous section to generate different fronts. At every iteration, change operators are applied to generate new solutions then the same process is repeated until reaching a maximum number of iterations. Thus, the bi-objective trade-off solutions are obtained and then the developer can navigate through these solutions, called Pareto front as will be detailed later, to select his preferred solution (detection rules). When the developers are looking for the rules that may detect few code smells with high correctness then they will select the solution with high precision and acceptable recall. In the opposite case, when developers are interested to explore and fix most of the smells in the Android app or evaluate its quality then they will select the solution with a high recall and acceptable precision. Otherwise, the developers will select the solution that proposes the maximum trade-off (called knee point) [30].

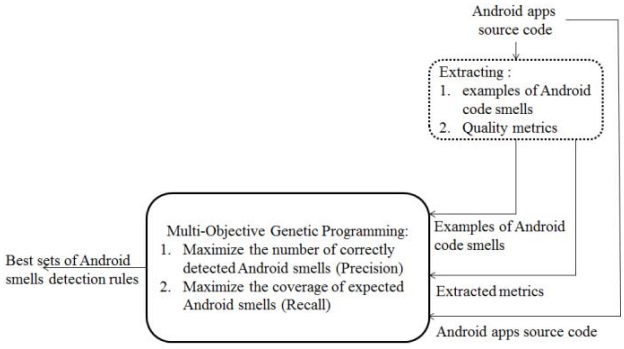

Figure 1. The proposed Android smells detection framework.

B. Solution Approach

A multi-objective optimization problem (MOP) consists of minimizing or maximizing an objective function vector under some constraints. Its general form is the following [30]:

$$\begin{cases} Min\ f(x) = [f_1(x), f_2(x), ..., f_M(x)]^T \\ g_j(x) \geq 0 \qquad j = 1, ..., P; \\ h_k(x) = 0 \qquad k = 1, ..., Q; \\ x_i^L \leq x_i \leq x_i^U \qquad i = 1, ..., n. \end{cases}$$

where M is the number of objective functions, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond to the lower and upper bounds of the variable x_i. A solution x_i

satisfying the (P+Q) constraints is said feasible and the set of all feasible solutions defines the feasible search space denoted by Ω. The resolution of a MOP yields a set of trade-off solutions, called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front (PF). In this paper, we selected multi-objective Genetic Programming as search technique to solve our problem. Multi-objective Genetic Programming (MOGP) is a powerful and widely-used evolutionary metaheuristic algorithm which extends the generic model of learning to the space of programs. Unlike other evolutionary approaches, in MOGP, solutions are themselves programs following a tree-like representation instead of fixed length linear string formed from a limited alphabet of symbols. MOGP is a process of program induction that allows automatically generating programs that solve a given task. Most exiting work on GP makes use of a single objective formulation of the optimization problem to solve using only one fitness function to evaluate the solution.

As described in Fig. 2, the first step in MOGP is to create randomly a population P_0 of individuals encoded as trees. Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation. The crossover operator is based on sub-trees exchange and the mutation is based on random change in the tree. Both populations are merged into an initial population R_0 of size N, and a subset of individuals is selected, based on the dominance principle and crowding distance [30] to create the next generation. This process will be repeated until reaching the last iteration (stopping criterion). We describe in the following subsections the three main adaptation steps: solution representation, change operators and fitness functions.

1) Solution representation

A solution consists of a set of multiple rules where every rule can detect a specific type of smells. In MOGP, a solution is composed of terminals and functions. The terminals correspond to different Android source code metrics with their threshold values. The functions that can be used between these metrics are Union (*OR*) and Intersection (*AND*). A solution is represented as a set of trees connected by the logic operators *OR*. Each rule of the solution is represented by a binary tree such that: each leaf-node (Terminal) belongs to the set of metrics described in Table 1 and their corresponding thresholds generated randomly. Each internal-node (Functions) belongs to the Connective (logic operators) set $C = \{AND, OR\}$.

The set of solutions (rules) corresponds to a logic program that is represented as a forest of *AND-OR* trees. Each sub-tree corresponds to a rule for the detection of specific Android smell as described in Section 2.1. Fig. 3 illustrates a simplified example of a solution per our formulation including two rules. The first rule is to detect an *MIM* antipattern using the two metrics *NBR* and *NCP* and the second rule detects an *HBR* antipattern using one metric *NM*. The threshold values are selected randomly along with the comparison and logic operators.

The initial population is generated by randomly assigning a set of metrics and their thresholds to the different nodes of the trees. The size of a solution, i.e. the tree's length is randomly chosen between upper and lower bound values. The determination of these two bounds is called the problem of bloat control in genetic programming where the goal is to identify the tree size limits. Thus, we performed several trial and error experiments using the HyperVolume (HP) performance indicator [30] to determine the upper bound after which, the indicator remains invariant. For the lower bound, it is arbitrarily chosen. The experiments section will specify the upper and lower bounds used in this study.

Begin
While *stopping criteria not reached* **do**
$R_t \leftarrow P_t \cup Q_t$;
$F \leftarrow$ fast-non-dominated-sort (R_t) ;
$P_{t+1} \leftarrow \emptyset$; i $\leftarrow 1$;
While $|P_{t+1}| + |F_i| \leq N$ **do**
Apply crowding-distance-assignment (F_i) ;
$P_{t+1} \leftarrow P_{t+1} \cup F_i$;
$i \leftarrow i + 1$;
End While
Sort (Fi, $\prec n$) ;
$P_{t+1} \leftarrow P_{t+1} P_{t+1} \cup F_i[1 : N - |P_{t+1}|]$;
$Q_{t+1} \leftarrow$ create-new-population (P_{t+1}) ;
$t \leftarrow t+1$;
End While
End

Figure 2. MOGP : A generic pseudo algorithm.

2) Solution variation

The mutation operator can be applied to a function node or a terminal node. It starts by randomly selecting a node in the tree. Then, if the selected node is a terminal (quality metric), it is replaced by another terminal (metric or another threshold value). If it is a function (*AND-OR*), it is replaced by a new function. If a tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree.

For the crossover, two parent individuals are selected and a sub-tree is picked on each one. Then crossover swaps the nodes and their relative sub-trees from one parent to the other. This operator must ensure the respect of the depth limits. The crossover operator can be applied with only parents having the same rules category (code-smell type to detect). Each child thus obtains information from both parents. When applying the change operators such as removing a metric, the different pre- and post-conditions are checked to ensure the applicability of the proposed rules.

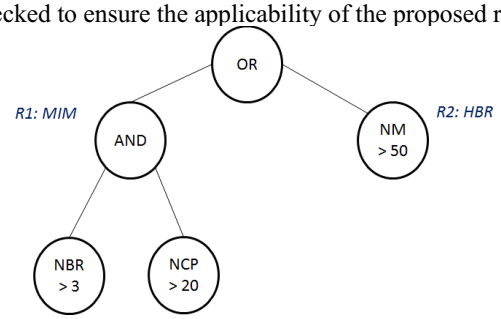

Figure 3. A simplified example of a solution.

3) Solution evaluation

The Android code-smells detection problem involves searching for the best metric combinations among the set of candidate ones, which constitutes a huge search space. A solution of our Android code-smells detection problem is a set of rules (metric combination with their thresholds values) where the goal of applying these rules is to detect code smells in an Android app. In our multi-objective formulation, we propose two objective functions to be optimized:

$$\begin{cases} \max f_1(S) = \dfrac{|\{DetectedAndroidAntiappterns\} \cap \{ExpectedAndroidAntipatterns\}|}{|\{DetectedAndroidAntipatterns\}|} \\ \max f_2(S) = \dfrac{|\{DetectedAndroidAntiappterns\} \cap \{ExpectedAndroidAntipatterns\}|}{|\{ExpectedAndroidAntipatterns\}|} \end{cases}$$

(1) Maximizing the detection of correct Android code smells over the list of suggested ones to check (precision).

(2) Maximizing the coverage of expected Android smells from the base of examples of multiple apps over the actual list of detected smells (recall).

The collected examples of Android code smells on 184 Android apps are taken as an input for our approach in our experiments. Analytically speaking, the formulation of the multi-objective problem can be stated as follows: Since we are considering a bi-objective formulation, we use the concept of Pareto optimality to find a set of compromise (Pareto-optimal) solutions. The fitness of a solution in MOGP corresponds to a couple (*Pareto Rank*, *Crowding distance*). MOGP classifies the population individuals (of parents and children) into different layers, called non-dominated fronts. Non-dominated solutions are assigned a rank of 1 and then are discarded temporary from the population. Non-dominated solutions from the truncated population are assigned a rank of 2 and then are discarded temporarily. This process is repeated until the entire population is classified with the domination metric.

IV. VALIDATION

To evaluate the ability of our Android smells detection framework to correctly identify design defects, we conducted a set of experiments based on 184 Android apps with source code hosted in open source repositories and found on GitHub. In this section, we first present our research questions and then describe and discuss the obtained results.

A. Research Questions and Evaluation Metrics

We defined four research questions to evaluate the performance of our approach in comparison to existing techniques. The three research questions are as follows:

RQ1: To what extent can our approach identify correctly Android code smells?

RQ2: How does our approach perform compared to a mono-objective algorithm (GP) and a manual approach not based on heuristic search?

RQ4: Can our approach be useful for Android app developers?

To answer these research questions, we considered the best detection rules solution by our approach as described in the previous section using the knee point strategy [30]. To answer RQ1, it is important to validate the proposed detection solution in terms of correctness. We asked a group of developers, as detailed later, to analyze manually the 184 Android Apps, as described in Table 2, to construct the base of examples. Thus, we calculated the precision and recall on these applications based on n-fold cross validation procedure:

$$\mathrm{RE}_{recall} = \frac{|\text{ suggested smells }| \cap |\text{ expected smells }|}{|\text{ expected smells }|} \in [0,1]$$

$$\mathrm{PR}_{precesion} = \frac{|\text{ suggested smells }| \cap |\text{ expected smells }|}{|\text{ suggested smells }|} \in [0,1]$$

When calculating the precision and recall, we consider a detected Android smell as a correct recommendation when it is detected by more than half of the developers of our experiments. Then, we calculated the f-measure defined as the average of precision and recall. To better investigate the relevance of detected Android smells, we asked the group of developers if they consider or not the correction of the analyzed defects. Thus, the relevance score *R* is defined as follows:

$$\mathrm{R}_{\text{relevance}} = \frac{\#\text{relevant Android smells}}{\text{detected Android smells}} \in [0,1]$$

We consider a defect as relevant if it was ranked relevant by more than half of the developers of our experiments. A correctly detected Android smells does not mean that it will be fixed by the developer. The high relevance of an Android smell indicate that the developer considers it as relevant to be fixed. To answer RQ2, we compared our approach to a mono-objective Genetic Programming algorithm based on one fitness function defined as the average of Precision and Recall. Then, the comparison was based on the *PR*, *RC* and *R* metrics. The goal of this question is to investigate if the fitness functions of our approach are conflicting. The fitness functions are not considered conflicting in case that a mono-objective GP outperforms our multi-objective approach. Furthermore, we compared our approach to a set of rules defined manually that translate the different symptoms of several Android smells. We used the metrics *PR, RC,* and *R* to perform the comparisons.

To answer RQ4, we used a post-study questionnaire that collects the opinions of developers on our tool and the severity of the detected Android smells. To this end, we received feedback from 27 software developers of mobile applications.

B. Experimental Setup

Our study involved 34 graduate students from a Software Quality Assurance course at the University of Michigan to analyze the results. Participants include 13 students who are working as full time developer or manager in software companies ranging from 4 to 9 years of programming experience. The remaining participants have minimum years of experience of 2 years in industry as programmer. All the participants are volunteers and familiar with Android apps, quality metrics and refactoring. We sent an online questionnaire to a total of 145 developers of mobile apps mainly via different LinkedIn groups on mobile development. 27 out of the 145 developers completed our

survey about the relevance of detecting Android smells in practice.

The participants were first asked to fill out a pre-study questionnaire containing six questions. The questionnaire helped to collect background information such as their programming experience, their familiarity with mobile apps development and software quality assessment. In addition, all the participants attended one lecture about Android smells and passed eight tests to evaluate their performance to evaluate and identify these defects. As described in Table 3, we formed 5 groups with 6 participants per group except one group (group 5) who received 4 participants. Thus, group 5 was assigned less apps to evaluate comparing to the remaining groups. Table 3 summarizes the survey organization including the category of Android apps and algorithms evaluated by the group. It is important to note that every app was evaluated by at least two participants. The groups were formed based on the pre-study questionnaire and the tests result to make sure that all the groups have almost the same average skills. We divided the participants into groups per the studied apps, the techniques to be tested and developers' experience. Thus, each group of participants who accepted to participate in the study received a questionnaire, a manuscript guide to help them to fill the questionnaire, the tools and results to evaluate and the source code of the studied apps. Since the detection of Android smells is a subjective process, it is normal that not all the developers have the same opinion. In our case, we considered the majority of votes to determine if suggested Android smells are correct or not. Each participant evaluates different solutions for the different techniques and apps.

We executed different scenarios. In the first scenario, we asked every participant to manually identify the Android smells in the apps assigned to them based on their definitions, the source code and metrics value. In the second scenario, we asked the developers to manually evaluate the relevance of the recommended Android smells by our algorithm. In the third scenario, we collected the opinions of the participants about the different evaluated tools based on a post-study questionnaire that will be detailed later. The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study. Participants do not know the research questions and the used algorithms. Finally, we prepared a survey of 13 questions about Android smells extracted from our experiments to understand their relevance for 27 external participants who are mobile application developers.

Parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 20, 50, 100, 150, 300 and 500. The stopping criterion was set to 10000 evaluations for all algorithms to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.7; mutation probability = 0.4 where the probability of gene modification is 0.1; stopping criterion = 10000 evaluations. Each algorithm is executed 30 times with each configuration and then the comparison between the configurations is done using the Wilcoxon test. To have significant results, for each couple (algorithm, category), we use the trial and error method in order to obtain a good parameter configuration.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is based on 30 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$). The latter tests the null hypothesis, H0, that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not, H1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value that is less than or equal to α ($\leq$ 0.05) means that we accept H1 and we reject H0. However, a p-value that is strictly greater than α ($>$ 0.05) means the opposite. In fact, for each problem instance, we compute the p-value obtained by comparing GP and the manually defined rules results with our MOGP ones. In this way, we determine whether the performance difference between our technique and one of the other approaches is statistically significant or just a random result. The results presented were found to be statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 95% confidence level ($\alpha < 5\%$).

TABLE II. THE DATA SET COLLECTED FROM 184 GIT REPOSITORIES CLASSIFIED WITH 17 CATEGORIES FROM THE F-DROID FORGE.

Category	#projects	#Android smells
Graphics	41	1724
Multimedia	17	628
Navigation	9	248
Health	34	963
Games	57	2431
Security	15	567
Education	11	379

TABLE III. SURVEY ORGANIZATION.

Participant groups	Systems	Techniques
Group 1	Graphics	MOGP GP Manual rules
	Multimedia	
	Navigation	
	Health	
Group 2	Graphics	MOGP GP Manual rules
	Health	
	Games	
	Security	
Group 3	Games	MOGP GP Manual rules
	Security	
	Education	
	Graphics	
Group 4	Graphics	MOGP GP Manual rules
	Games	
	Multimedia	
	Security	
Group 5	Multimedia	MOGP GP Manual rules
	Navigation	
	Health	

C. *Results and Discussions*

Results for RQ1. In this section, we evaluate the performance of our MOGP adaptation for the detection of different types of Android code smells on our benchmark of 184 Android apps. Fig. 4 summarizes our findings. The expected Android smells were detected with an average of more than 81% of precision and recall on the different apps. The highest precision was found in the Multimedia apps where 86% of code-smells were detected. The lowest precision was found in Health apps with 79% of detected Android smells. This finding indicates that the list of detected Android smells did not contain high number of false positive, thus developers will not waste a lot of their time for manual inspections. We found similar facts when analyzing the recall scores of MOGP on the different apps where an average of more than 83% of expected Android smells was covered. The highest and lowest recall scores were respectively 78% (Graphics) and 88% (Security). An interesting observation is that the performance of MOGP in terms of PR and RC is independent on the number of Android smells in the apps to analyze. The PR and RC scores of Multimedia are among the highest ones but these apps contained more Android smells to detect comparing to other categories such as Security, Education, etc. The same observations are valid for the F-measure and relevance results. Most of the identified smells were considered relevant, in average, by the developers and they should be fixed with average of more than 70% on the different categories of apps.

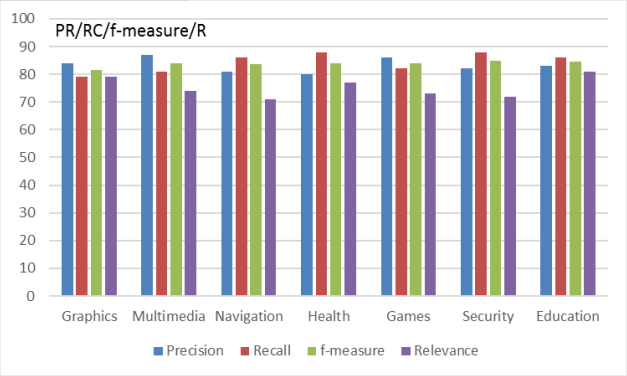

Figure 4. The median precision, recall, F-measure and relevance scores of MOGP over 30 runs obtained on the 184 Android apps.

We noticed that our technique does not have a bias towards the detection of specific Android smell types. As described in Fig. 5, in all apps, we had an almost equal distribution of each Android smell types. Having a relatively good distribution of Android smells is useful for a developer. Overall, all the types of smells are detected with good precision and recall scores in the different apps (more than 79%). Some types of smells have the highest precision and recall because they are the easiest to detect such as HMU. This ability to identify different types of Android smells underlines a key strength to our approach confirming the adequacy of the selected metrics.

To conclude, our MOGP approach detects well all the different types of considered Android-smells (RQ1).

Results for RQ2. In this section, we compare our MOGP adaptation to a mono-objective genetic programming formulation (aggregation of both precision and recall into one fitness function) and an approach where the rules are manually defined. Fig. 6 shows the overview of the average results of the significance tests comparison between all these algorithms over 30 runs. It is clear that MOGP outperforms GP in 100% of the cases in terms of precision, recall, f-measure and relevance. However, as it will be discussed later, the execution time (CT) of our MOGP algorithm is slightly higher than GP. The statistical tests are based on multiple pairwise comparisons using the Wilcoxon test. Thus, we should adjust the p-values. To achieve this task, we used Holm method that is reported to be more accurate than the Bonferroni one. The adjusted p-values confirming that the results are statistically significant with a 95% confidence level ($\alpha = 5\%$). The results confirm that the two objectives of precision and recall are conflicting and that the multi-objective algorithm helps to improve the diversity of the generated detection solutions.

Since it is not sufficient to compare our proposal with only another search algorithm, we compared the performance of MOGP with the manual approach using the same types of Android smells that could be detected by both tools. Fig. 6 summarizes the results of the precision, recall, relevance and f-measure obtained on the 184 Android apps. The average recall for the manual approach was better than MOGP; however the precision scores are lower than our proposal on all apps. In fact, the higher recall achieved by the manual approach can be easily explained by the use of relatively flexible constraints with high thresholds. This can be explained by the high calibration effort required to define manually the detection rules for some specific Android smell types and the ambiguities related to the selection of the best metrics set. The recall of MOGP is lower than the manual approach, on average, but it is an acceptable recall average which is higher than 83%.

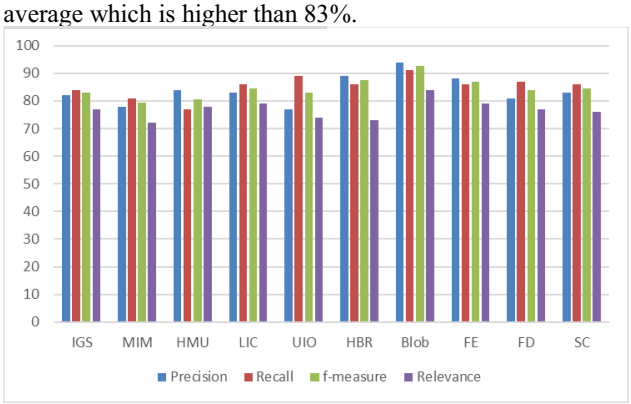

Figure 5. The median precision, recall, f-measure and relevance scores of MOGP obtained for every Android smell type over 30 runs on the 184 Android apps.

Since our proposal is based on a search algorithm, it is important to evaluate the execution time (CT). It is evident that MOGP requires higher execution time than GP and the manual approach since MOGP has higher number of objectives and generates a Pareto front of solutions. All the search-based algorithms under comparison were executed

on machines with Intel Xeon E3 Quad-Core and 32 GB RAM. We recall that all algorithms were run for 10000 evaluations. This allows us to make a fair comparison between CPU times. Overall, GP and the manual algorithms were faster than BLOP. In fact, the average execution time for MOGP and GP were respectively 9 and 7 minutes. However, the execution for MOGP is reasonable because the algorithm is executed only once then the generated rules will be used to detect Android smells. There is no need to execute MOGP again except in the case that the base of examples will be updated with a high number of new Android smell examples.

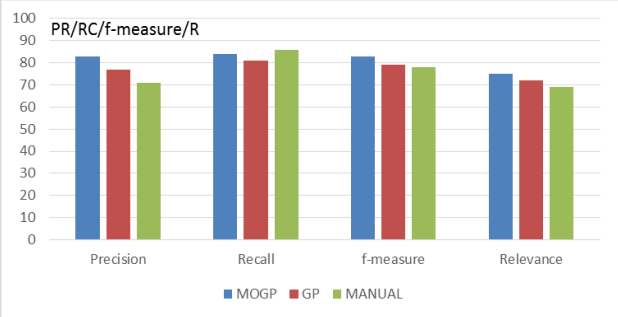

Figure 6. The median precision, recall, f-measure and relevance scores of MOGP, GP and a manual approach obtained over 30 runs on the 184 Android apps. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 95% confidence level.

One of the advantages of using our MOGP adaptation is that the developers do not need to provide a very large set of Android smell examples to generate the detection rules. However, the reliability of the proposed MOGP approach is sensitive to the identified examples. In our study, we showed that by using several examples from Android apps directly, without any adaptation, the MOGP method can be used out of the box, and this will produce good detection results for the detection of Android smells for all the studied apps. In an industrial setting, we could expect a company to start with existing Android apps, and gradually transform its set of good code examples to include context-specific data. This might be essential if we consider that different developers have different best/worst practices.

In conclusion, we answer RQ2 by concluding that the results in our experiments confirm that our proposed MOGP is adequate, and it outperforms, in average, both GP and the manual approach.

Results for RQ3. We summarize, briefly, in the following the feedback of the participants during the think aloud sessions. Most of the participants mention that the detection rules generated by MOGP represents a faster solution than manual detection techniques. The manual techniques represent a time-consuming process to find the locations where the quality issues should be located in mobile apps or to calibrate the metrics threshold or the combination of metrics to identify a maintainability issue manually. The participants found the detection rules useful to maintain a good quality of the design and to make sure that some quality issues are fixed after introducing changes. In addition, the developers liked the flexibility to modify the rules (metrics or thresholds) if required. Some possible improvements for our detection techniques were also suggested by the participants. Some participants believe that it will be very helpful to extend the tool by adding a new feature to prioritize the identified Android smells based on several criteria such as severity, risk, cost and benefits. They believe that current tools do not provide any support to estimate the severity, risk, cost and benefits of fixing some maintainability issues and how to manage them. In fact, most of the developers mentioned that they do not have enough time to focus on fixing all detected Android smells thus it is important to prioritize them for the developers of mobile apps.

In another scenario, we evaluated the relevance of the detected code-smells on the different open source systems by both internal and external participants of our study. The first part of the questionnaire includes questions to evaluate the relevance of some detected Android smells using the following scale: 1. Not at all relevant; 2. Slightly relevant; 3. Moderately relevant; and 4. Extremely relevant. If a detected Android smell is considered relevant, then this means that the developer considers that it is important to fix it. The second part of the questionnaire includes questions for those Android smells that are considered at least "moderately relevant", we asked the subjects to specify their usefulness to perform some maintenance activities: 1. Refactoring guidance;2. Quality Assurance/Performance; 3. Code inspection; 4. Effort prediction and 5. Bug prediction. For external participants, we shared with them a total of 10 Android smell examples for every type extracted from our experiments.

Fig. 7 illustrates that only less than 12% of detected Android-smells are considered not at all relevant by the developers. Around 68% of the code-smells are considered as moderately or extremely relevant by the different groups, and this confirms the importance of the detected Android smells for developers, and also the results are similar between internal and external participants. To better evaluate the relevance of the detected Android smells, we investigated the types of smells that developers perhaps consider them more or less important than others. It is clear that the detected blobs are considered very relevant for developers. One of the reasons can be the importance of distributing the behavior/functionalities of a mobile app. In addition, it is very difficult for developers to understand existing functionalities to add new ones if most of them are implemented in one or few modules. Another interesting observation is that UIO are not considered very relevant by developers. In fact, UIO is very subjective to detect and fix.

It is also important to evaluate the usefulness of the detected Android smells for the developers of mobile apps in their daily development activities. To this end, we asked the external participants to justify the usefulness of the Android smells ranked as moderately or extremely relevant. Fig. 8 describes the obtained results.

The main usefulness is related to refactoring guidance to improve the performance and quality of mobile apps. In fact, most of the developers we interviewed found that the detected Android smells give relevant advices about where refactorings should be applied and what are the common

used refactorings to fix these defects. In addition, they found that the Android smells detection process is much more helpful than the traditional analysis of software metrics to find refactoring opportunities. They consider the use of traditional quality metrics for Quality Assurance as a time-consuming process, and it is easier to interpret the results of detected code-smells and apply the appropriate refactorings to improve the overall quality of the mobile apps. Furthermore, the participants agreed that the popularity of mobile apps heavily depends on their quality, stability and performance. Thus, they consider such activity to detect and fix Android smells extremely important especially to reduce the bugs rate and improve the overall competitiveness of the developed mobile apps.

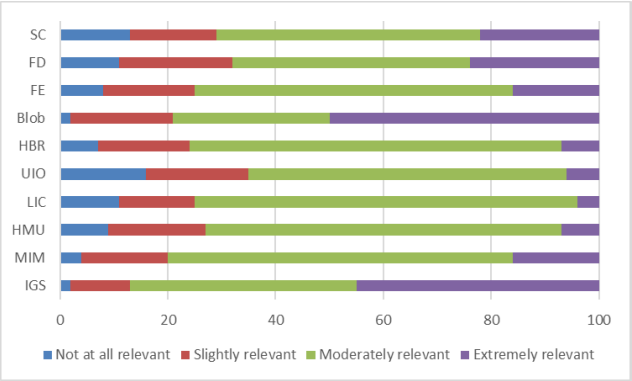

Figure 7. The relevance of the different types of Android smells as evaluated by the developers of mobile Apps.

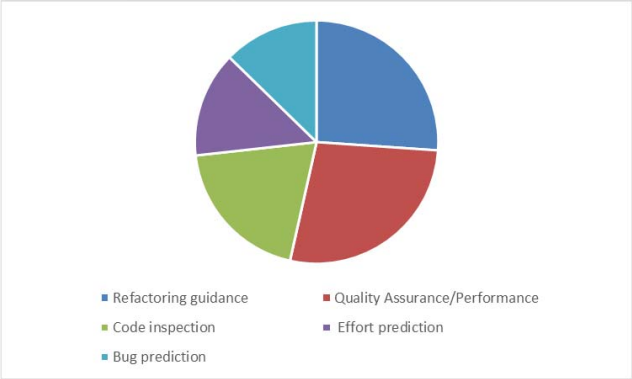

Figure 8. The usefulness of the detected Android smells as evaluated by the developers of mobile Apps.

D. Threats to Validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We addressed conclusion threats to validity by performing 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Wilcoxon rank sum test with a 95% confidence level (α = 5%). However, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution to provide the best possible performance. In addition, our multi-objective formulation treats the different types of Android smells with the same weight in terms of severity when detecting them.

Internal validity is concerned with the causal relationship between the treatment and the outcome. We dealt with internal threats to validity by performing 30 independent simulation runs for each problem instance. This makes it highly unlikely that the observed results were caused by anything other than the applied multi-objective approach. The second internal threat is related to the variation of correctness between the different groups when using our approach and other tools. In fact, our approach may not be the only reason for the superior performance because the participants have different skills and familiarity with Android smells. To counteract this, we assigned the developers to different groups per their programming experience to reduce the gap between the different groups and we also adapted a counter-balanced design.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on many Android apps belonging to different domains and having different sizes. However, we cannot assert that our results can be generalized to other applications, and to other practitioners. Future replications of this study are necessary to confirm our findings. Further empirical studies are also required to deeply evaluate the performance of the MOGP algorithm using the same problem formulation. The first threat is the limited number of participants and evaluated apps, which externally threatens the generalizability of our results. In addition, our study was limited to the use of specific types of Android smells.

V. Conclusion and Future Work

We proposed, in this paper, an approach to generate rules that can detect code smells in Android apps. A rule is a combination of quality metrics with their threshold values to detect a specific type of code smell. The proposed approach takes as input a set of Android-specific code smell examples from multiple apps and uses a multi-objective genetic programming algorithm MOGP to find the best set of rules that covers most of the expected Android code smells by finding a trade-off between the conflicting precision and recall measures of rules coverage of the base of examples.

We implemented our proposed approach and evaluated it on a set of 184 Android projects. The statistical test of our results shows that the generated detection rules identified 8 Android-code smell types on these mobile applications with an average manual correctness higher than 81% and a relevance average of 77% based on the feedback of 27 developers of mobile applications.

Future work involves validating our technique with additional Android smells type and more apps to conclude about the general applicability of our methodology. Furthermore, we only focused, in this paper, on the detection of Android smells. We plan to extend the approach by automating the correction of detected Android smells. In addition, we will consider the prioritization of detected defects during the correction step.

REFERENCES

[1] Aiko F. Yamashita, and Leon Moonen. 2012. Do code smells reflect important maintainability aspects? In Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM), IEEE Computer Society, Washington, DC, USA, 306-315.

[2] Ali Ouni, Marouane Kessentini, Houari A. Sahraoui, Katsuro Inoue, Kalyanmoy Deb: Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. ACM Trans. Softw. Eng. Methodol. 25(3): 23:1-23:53 (2016)

[3] Ali Ouni, Marouane Kessentini, Houari A. Sahraoui, Katsuro Inoue, Mohamed Salah Hamdi: Improving multi-objective code-smells correction using development history. Journal of Systems and Software 105: 18-39 (2015)

[4] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. Germán, Katsuro Inoue: Search-based software library recommendation using multi-objective optimization. Information & Software Technology 83: 55-75 (2017)

[5] Bader Alkhazi, Terry Ruas, Marouane Kessentini, Manuel Wimmer, William I. Grosky: Automated refactoring of ATL model transformations: a search-based approach. MoDELS 2016: 295-304

[6] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), 476-493.

[7] Dilan Sahin, Marouane Kessentini, Manuel Wimmer, Kalyanmoy Deb: Model transformation testing: a bi-level search-based software engineering approach. Journal of Software: Evolution and Process 27(11): 821-837 (2015)

[8] E. Van Emden and L. Moonen. 2002. Java Quality Assurance by Detecting Code Smells. In Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE). IEEE Computer Society, Washington, DC, USA, 97-.

[9] G. Adnane Ghannem, Ghizlane El-Boussaidi, Marouane Kessentini: On the use of design defect examples to detect model refactoring opportunities. Software Quality Journal 24(4): 947-965 (2016)

[10] G. Hecht, B. Omar, R. Rouvoy, N. Moha, and L. Duchien. Tracking the software quality of android applications along their evolution. In 30th IEEE/ACM International Conference on Automated Software Engineering, page 12. IEEE, 2015.

[11] Hanzhang Wang, Marouane Kessentini, Ali Ouni: Bi-level Identification of Web Service Defects. ICSOC 2016: 352-368

[12] Hanzhang Wang, Marouane Kessentini, William I. Grosky, Haythem Meddeb: On the use of time series and search based software engineering for refactoring recommendation. MEDES 2015: 35-42

[13] Harman, M., Mansouri, S.A. and Zhang, Y., 2012. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR), 45(1), p.11.

[14] https://www.abiresearch.com/press/ android-will-account-for-58-of-smartphone-app-down. [Online; accessed January-2017].

[15] https://www.mobileworldlive.com/apps/news-apps/app-revenue-to-hit-100b-in-2020-says-app-annie/ [Online; accessed January-2017].

[16] 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.

[17] Infusion hydrogen: Design flaw detection tool. Available at: http://www.intooitus.com/products/infusion, 2012.

[18] 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 Software modernisierung MMSM, volume 2014, 2014.

[19] Linares-Vásquez, Mario, et al. "API change and fault proneness: a threat to the success of Android apps." Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM, 2013.

[20] M. Brylski. Android smells catalogue. http://www.modelrefactoring.org/smell_catalog, 2013. [Online; accessed January-2017].

[21] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[22] 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. IEEE/ACM, 2015.

[23] Mannan, Umme Ayda, et al. "Understanding code smells in Android applications." Proceedings of the International Workshop on Mobile Software Engineering and Systems. ACM, 2016.

[24] Mäntylä Mika, Jari Vanhanen, and Casper Lassenius. 2003. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In Proceedings of the International Conference on Software Maintenance (ICSM). IEEE Computer Society, Washington, DC, USA, 381-384.

[25] Marouane Kessentini, Rim Mahouachi, Khaled Ghédira: What you like in design use to correct bad-smells. Software Quality Journal 21(4): 551-571 (2013)

[26] Marouane Kessentini, Wafa Werda, Philip Langer, Manuel Wimmer: Search-based model merging. GECCO 2013: 1453-1460

[27] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code Bad Smells: a review of current knowledge. J. Softw. Maint. Evol. 23, 3 (April 2011), 179-202.

[28] Mohamed Wiem Mkaouer, Marouane Kessentini: Model Transformation Using Multiobjective Optimization. Advances in Computers 92: 161-202 (2014)

[29] Oliver Ciupke. 1999. Automatic Detection of Design Problems in Object-Oriented Reengineering. In Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS). IEEE Computer Society, Washington, DC, USA, 18-32.

[30] P.J.Angeline. Genetic programming and emergent intelligence. In K.E.Kinear, Jr., editor, Advances in Genetic Programming, chapter 4, pages 75-98. MIT Press, 1994

[31] R. Marinescu, Detection strategies: metrics-based rules for detecting design flaws, Proceedings of the 20th International Conference on Software Maintenance, IEEE Computer Society Press, pp. 350-359, 2004.

[32] 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, 2013.

[33] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, Ali Ouni: Recommending relevant classes for bug reports using multi-objective search. ASE 2016: 286-295

[34] S. C. Kothari, Luke Bishop, Jeremias Sauceda, and Gary Daugherty. 2004. A Pattern-Based Framework for Software Anomaly Detection. Software Quality Control, 12, 2 (June 2004), 99-120.

[35] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. M. III, and T. J. Mowbray. Anti Patterns: Refactoring Software,Architectures, and Projects in Crisis. John Wiley and Sons, 1st edition, 1998

[36] Wenquan Wang, Marouane Kessentini, Wei Jiang: Test Cases Generation for Model Transformations from Structural Information. MDEBE@MoDELS 2013: 42-51

.