Arabian Journal for Science and Engineering (2020) 45:3289–3315
https://doi.org/10.1007/s13369-020-04365-1

RESEARCH ARTICLE-COMPUTER ENGINEERING AND COMPUTER SCIENCE

Recovering Android Bad Smells from Android Applications

Ghulam Rasool[1] · **Azhar Ali**[1]

Received: 16 July 2019 / Accepted: 19 January 2020 / Published online: 1 February 2020

Abstract
The demand for Android mobile software applications is continuously increasing with the evolution of technology and new enriching features to make the life of people easy and comfortable. The mobile-based software applications are frequently updated as compared to other web and desktop applications. Due to these frequent updating cycles, the developers sometimes make changes in a rush which leads to poor design choices known as antipatterns or code bad smells. Code bad smells degrade the performance of applications and make evolution difficult. The recovery of bad smells from mobile software applications is still at infancy but it is a very important research realm that requires the attention of researchers and practitioners. The results of recovery may be used for comprehension, maintenance, reengineering, evolution and refactoring of these applications. Most state-of-the-art approaches focused on the detection of code bad smells from object-oriented applications and they target only a few code smells. We present a novel approach supplemented with tool support to recover 25 Android code bad smells from Android-specific software applications. We evaluate our approach by performing experiments on 4 open source and 3 industrial Android-specific software applications and measure accuracy using standard metrics.

Keywords Smart phone · Mobile software applications · Android smells · Software quality · Code refactoring

1 Introduction

In the last decade, the adoption of mobile devices and usage of mobile software applications have experienced exceptional and exponential growth around the globe. The market for mobile applications is expected to generate revenue around 189 billion dollars till 2020 that represents an excellent opportunity for mobile application developers interested to develop high quality and successful software applications [1, 2]. There were around 3.8 million mobile applications on Google Play Store till March 2018 and the number of downloads on famous app stores was 178 billion till 2017 [1]. It is reported that Android accounts more than 85.9% of global smartphone sale worldwide.[1] It is predicted in a Gartner[2] report that demand for mobile application development will soar at least 5 times faster till 2021 than IT capacity of companies to deliver these applications. It is predicted that the number of smart mobile phone users in the world is expected to pass the five billion mark by 2019.[3] Android is the most popular open source mobile operating system designed for smartphones and tablets and it has market share of around 80% [3]. Due to its openness and ease of use, it extracts a large number of developers, researchers and vendors for working on Android development environment. It is also reported that there are 2 billion Android devices in the world.[4] Indeed, the market for mobile applications has skyrocketed and it represents a rich opportunity for developers, researchers and practitioners. However, there is a dearth of research on evaluating the quality of mobile applications, especially Android and iOS applications. The success of mobile applications is dependent on their quality, reliability, correctness, performance and overall satisfaction of users.

The development of high quality, robust and versatile mobile applications require the use of formally grounded

✉ Ghulam Rasool
grasool@cuilahore.edu.pk

[1] Department of Compute Science, COMSATS University Islamabad, Lahore, Pakistan

[1] https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

[2] https://www.mendix.com/blog/5-key-themes-from-the-gartner-application-strategies-solutions-summit/.

[3] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide.

[4] https://youtu.be/Y2VF8tmLFHw.

methods and standard practices. In order to achieve these objectives in Android applications, the mobile software application developers apply standard practices called design patterns for the design and development of mobile applications because patterns are proven solutions and they improve quality and development productivity of developers [4]. It is also observed that developers sometimes overlook standard practices due to frequent and urgent changes, inexperience, time pressure, framework constraints and other reasons. The violation of standard principles and practices cause the birth of bad smells in software applications that are poor solutions. These bad smells have an impact on the quality, performance, comprehension and maintenance of mobile applications. A large number of papers are presented on the specification, detection and correction of code bad smells related to object-oriented software applications but a little attention is still paid on the detection and correction of code bad smells related with mobile applications. Riemann et al. [5] presented a catalog of 30 code bad smells(antipatterns) specific to Android mobile applications. The recovery of Android-specific code bad smells is still an open issue and it yields many benefits and supports a number of disciplines such as comprehension, reusability, maintenance, reengineering and refactoring.

The recovery of code bad smells from existing mobile software applications is more complex as compared to the recovery of design patterns. This is because of the fact that description containing detail of design patterns may exist in the available documentation but bad smells can never be part of any document. The manual analysis of bad smells from source code is a very daunting and challenging task due to the size and complexity of mobile software applications. The recovery is a prerequisite for the automatic refactoring of mobile software applications in order to improve the structure and quality of these applications without altering their external behavior [4]. Most authors perform static analysis on Android programs to uncover bad smells and bugs [4, 6]. A comprehensive systematic literature review on the static analysis of Android applications and limitations of static analysis for Android applications are presented by Li et al. [4]. All existing techniques and tools applied for the recovery of code smells from object-oriented software applications are not capable to recover bad smells from Android-specific applications due to the different structure of Android applications.

Based on the comprehensive analysis of literature review, we ascertained that many research works are presented on the recovery of code bad smells [7–38] from object-oriented software applications but only a limited number of approaches are presented for the detection of Android-specific code smells from mobile software applications [36, 39–51]. The reason is very obvious that mobile applications have still very young history and first-ever catalog of Android-specific bad smells is presented in 2012 by Reimann et al. [5]. In the last 5 years, it is being observed that research work for the specification and detection of Android bad smells is getting more and more attention of researchers due to unprecedented growth in the field of mobile applications. A number of empirical studies have highlighted that Android/OO code bad smells have an impact on comprehension, maintenance, reliability, change proneness, fault proneness, evolution, security, complexity, performance, energy efficiency, CPU performance, bugs, bugs prediction, exceptions handling, prioritization for refactoring, accuracy and quality of software systems [49, 52–103]. These publications clearly reflect the importance of field and motive researchers for recovering and removing bad smells from Android and other types of software applications.

Through a review of the literature, we identified the following gaps related to the research presented for detection of Android code bad smells. First, there is nonexistence of recent and comprehensive state of the art on all Android code bad smell detection techniques and tools. The comprehensive review of Android bad smell techniques and tools is important for the research community. Secondly, the standard specifications and detection rules for the recovery of Android bad smells are not publically available. The standard and agreed upon specifications from the research community are essential for the accurate recovery of smells. Thirdly, all presented techniques focused on a few Android code smells during recovery of smells from the source code. The generalization of these techniques for other code smells becomes questionable. Fourthly, most techniques applied PAPRIKA or extended versions of PAPRIKA tool to detect Android code smells. The PAPRIKA is limited to detect only 4–12 Android-specific code smells. It is very difficult to customize PAPRIKA and set dynamic threshold values for different Android bad smells. Finally, there is still no standard benchmark corpus for evaluating results of Android-specific code smells detection techniques and tools. We did not find a single approach that made detection results publically available for detail review and comparison of researchers. It becomes very difficult to compare the results of different techniques and tools in the absence of publically available benchmarks.

The motivation for this paper is to solve above-mentioned problems. We present a comprehensive state of the

art on all techniques and tools used for the detection of Android code smells from different Android applications till 2019. The standard specifications, detection rules and source code metrics related to 25 Android code smells are presented. We present a flexible approach by integrating the concept of source code analysis and source code metrics for the recovery of 25 Android code bad smells from Android software applications. Our prototype tool is flexible enough to accommodate variable code bad smell definitions and is easily extendable for other code smells and programming languages. The tool is publically available on the Web.[5] We publish results of our approach on the web for the review and comparisons of researchers [104]. We also compare the results of our approach with a representative state-of-the-art approach on smell by smell basis [42].

Following are the notable contributions of this paper:

- The comprehensive and recent state of the art on code bad smell detection techniques and tools for Android apps.
- The Specification of 25 code bad smells for mobile applications.
- An approach supplemented with tool support for the recovery of 25 Android code bad smells.
- The evaluation and comparison of results of the proposed approach with a state-of-the-art approach.
- A publically available benchmark on results of Android code bad smells.

The rest of this paper is organized as follows: The state of the art related to Android bad code smells is discussed and summarized in Sect. 2. We describe specifications and source code metrics for Android bad smells in Sect. 3. The architecture and working of approach through an example are demonstrated in Sect. 4. We also discuss prototyping tool and limitations of the proposed approach in Sect. 4. We present the evaluation of our approach, comparison of results, discussion on results and accuracy in Sect. 5. The threats to the validity of approach and its results are discussed in Sect. 6. Section 7 concludes the whole approach and recommends future work.

2 Related Work

A number of approaches are presented for automatic detection of code smells from different types of software applications. Most approaches focused on the recovery of code smells from object-oriented software applications as highlighted in the Introduction Section. Here, it will be unnecessary to discuss these approaches as a number of reviews, surveys and empirical studies are presented on these approaches [27, 43, 105–118]. A limited number of approaches are presented for the detection of Android-specific code smells. We focus on Android-specific code smells in this paper. We partially followed guidelines of systematic literature review presented by kitchenham[119] for the literature review in this section. We follow four steps for our review: (i) Define a search string, (ii) select the primary studies, (iii) snowballing, (iv) assess the quality of selected primary studies. In the first step, we define the following search string for the selection of studies:

((‘Android Code Smells’OR‘Android Code Bad Smells’OR‘Android Bad Smell’OR‘Android Antipatterns’ OR‘Android Design Smells’OR‘AndroidDesignFlaws’)AND(‘Detection ’OR‘Recovery’OR‘Recognition’OR‘Identification’) AND(‘Approaches’OR‘Methods’OR‘Techniques’ OR’ Tools’)AND (In ‘Title’OR‘Abstract’OR’ Keywords’))

Initially, we found 344 publications with the above search string. The search string was executed on IEEE Explore, ACM, Springer and Google scholar. Secondly, we applied inclusion/exclusion criteria and selected 19 publications. These papers are listed in Table 1. The time scale for the selection of papers is from 2014 to June 2019. The reason for this time frame is that research on android code smells detection started in the year 2014. We selected only studies in this section that have specific focus on the detection of Android bad smells from Android/object-oriented software applications. We selected three studies published as Master thesis in the year 2018 on Android code smells [1, 47, 49]. We also selected three studies that focus on the detection of object-oriented code smells from Android applications [1, 53, 120]. The papers that only analyze the impact of Android/object-oriented code smells are not included [3, 46, 57, 74, 121–128]. The studies focusing on single duplicate code smell from Android/object-oriented software applications are also excluded as there are already reviews on that particular bad smell. Thirdly, we also applied the snowball method for the selection of studies from references of 19 selected papers and found 3 relevant studies [44, 71, 72]. The total 25 papers are selected and we assessed the quality of selected studies from their publication venues.

The key facts that we want to explore from state-of-art approaches are technique and tool applied, language of Android applications, datasets, detected smells and accuracy. A classification for categorization of code smell detection technique is presented by Kessentini et al. [137]. We follow

[5] https://github.com/AzharAli92/DAAP.

Table 1 Summarized information of mobile antipatterns detection techniques

References	Technique/tool	Tool (type, availability)	Language	Examined applications	Detected bad smells	Accuracy (%)
[39]	Search based	Test Generation Framework(Third party tools)/no	Java	30 apps Google Play Store	4 energy bugs/smells	NM
[40]	Metric based	PAPRIKA Open Source/yes	Java	15 popular apps Google Play Store	4 OO code smells 4 Android code smells	Prec.98% Rec. 98%
[41]	Metric based	PAPRIKA Open Source/yes	Java	15 popular apps Google Play Store	4 OO code smells 4 Android code smells	Prec.98% Rec. NA
[52]	Metric based	PAPRIKA Open Source/yes	Java	106 Android applications Google Play Store	3 OO code smells 4 Android code smells	NM
[129]	Search based	ASYNCDROID Research/no	Java	9 open source Android apps from Github	Async code smells	NM
[130]	Metric based	PAPRIKA Open Source/yes	Java	2 open source Android apps from F-Droid	3 Android code smells	NM
[53]	Metrics based	InFusion Commercial/no	Java	500 Android apps 750 desktop apps	6 OO code smells	NM
[131]	Metric based	Extended PAPRIKA Open Source/yes	Objective C& Swiff	279 open source iOS apps from Github	4 OO code smells 3 IoS code smells	NM
[132]	Search based	EARMO Research/no	Java	20 Android applications from F-Droid	5 OO code smells 3 Android code smells	F1. 84%
[133]	Search based	Research Prototype Research/no	Java	184 Android Projects from GitHub	4 OO code Smells 6 Android code smells	82%
[44]	Metric based	PAPRIKA + NAGA-VIPER Open Source/yes	Java	5 Android open source apps from F-Droid	3 Android code smells	N/A
[42]	Metric based	ADOCTOR Research/yes	Java	18 Android apps	15 Android code smells	Prec.98% Rec. 98%
[54]	Symptom based	Research Prototype Prototype/no	Java	46 000 apps from Google Play Store	10 Security code Smells	NM
[134]	Search based	P-Lint Prototype/yes	Java	40 Android apps	4 Permission code smells	NM
[45]	Metric based	PAPRIKA Open Source/yes	Java	395 different apps from F-Droid	4 OO code smells 4 Android code smells	NM
[135]	Search based	Research Prototype Prototype/no	XML	Web Application	6 Usability code smells	NM
[55]	Search based	Leafactor, Open source/yes	Java	140 open source Android apps from F-Droid	5 Energy code smells	NM
[1]	Search based	ML based detector Publically available on GitHub	Java	5 Android apps from Google Play Store	5 OO code smells	F1. 29%
[120]	Metrics based	DART(Eclipse Plug-In)/no	Java	1 Alogcat application from GitHub	3 OO code smells	NM
[46]	Metric based	PAPRIKA Open Source/yes	Java&Kotlin	925 Open source Android apps from F-Droid	4 OO code smells 6 Android code smells	NM
[47]	Search based	tsDetect Open Source/yes	Java	656 Android open source apps from F-Droid	19 Android Test code smells	NM
[63]	Symptoms and Search based	Android Lint/Publically available on GitHub	Java	732 apps from GitHub	12 Security smells	F1. 85–100%
[49]	Search based	BadDroidDetector Open Source/yes	Java	1500 Android apps from F-Droid	13 Android Smells	NM

Table 1 (continued)

References	Technique/tool	Tool (type, availability)	Language	Examined applications	Detected bad smells	Accuracy (%)
[48]	Search and Symptoms based	Modelio, OntoUM and Protégé	Java	29 Android apps from APK Mirror	15 Android Smells	NM
[136]	Search based	FAKIE Prototype/no	Java	48 open source mobile apps from F-Droid	4 OO code smells 6 Android code smells	F1. 95%

OO object oriented, *NM* not mentioned, *Prec* precision, *Rec* recall, *F1* combined effect of precision and recall

this classification for the categorization of Android bad smells detection techniques. Most state-of-the-art Android smells detection techniques apply tools for the implementation of the proposed technique. We analyze tools based on their type and availability. The features of these tools are used for the development of our prototyping tool discussed in Sect. 4. Similarly, datasets, number of detected smells and accuracy are other key features analyzed critically. These attributes are applied for the empirical evaluation of studies. Table 1 presents the summarized information about these selected approaches. We briefly discuss the key features of selected primary studies.

Banerjee et al. [39] presented an approach to detect energy bugs from Android mobile applications by integrating concepts of graph-based search algorithm and guidance heuristics. The approach is based on automation test generation framework that systematically generates test inputs. These test inputs are used for the detection of bugs. They detected 4 energy bugs that degrade the performance of mobile applications. Authors performed experiments on 30 Android applications and claimed that approach has only a few false positives. However, they did not calculate the accuracy of the approach.

Hecht et al. [40, 41] presented code smell detection approaches supplemented with a tool named as PAPRIKA to identify object-oriented and Android-specific code smells. The PAPRIKA approach follows a two-step process: (i) generating java source code from Android package file and (ii) Detecting OO and Android-specific code smells. Authors focused only on 8 code smells in both publications. Hecht et al. [52] presented an empirical-based approach that examines the evolution of mobile apps and assesses their quality by considering code smells. They performed experiments to extract the same set of code smells presented in their previous publications [40, 41]. Hecht et al. [130] also presented an approach to evaluate the impact of Android code smells on the performance of software applications. They performed experiments on different projects to evaluate the impact of three Android performance code smells. They concluded that correction of these code smells can improve the user interface and memory performance.

Lin et al. [129] presented an approach to detect and refactor inappropriate Asynchronous constructs from Android applications that cause memory leaks, lost results, and wastage of energy. They implemented an approach with a tool named ASYNCDROID. They evaluated the approach using a large number of open source Android applications. They also evaluated their results and research questions through a survey by involving 10 experts (Android developers).

Palomba et al. [42] presented an approach supplemented with ADOCTOR tool for the detection of code smells specific to Android mobile applications. The proposed approach is able to detect 15 code smells from a catalog of Android-specific code smells presented by Reimann et al. [5]. The approach is a mixture of metric and simple text comparison claiming an average precision and recall of 98%. A number of code smells are detected by rules based on source code metrics and by a simple text comparison. There is a lack of accuracy while conducting text comparisons as the intended metrics may exist in the comments section of source code files. The ADOCTOR is available publically for experimentation. Moreover, the authors did not compare their results with state-of-the-art techniques and tools.

Morales et al. [132] reported an approach to detect and correct antipatterns/code smells related to energy consumptions from mobile software applications. Authors applied a tool named as EARMO based on three multi-objective search-based algorithms. The experiments are performed on 20 Android mobile applications available at F-Droid[6] to refactor 5 object-oriented code bad smells and 3 Android mobile bad smells. Authors claim that the battery life of the mobile phone is extended up to 29 min after refactoring operation.

Mannan et al. [53] presented an empirical study to compare the type, density and distribution of code smells in mobile vs desktop applications. They analyzed a dataset of 500 open source Android applications and 750 desktop applications. They applied InFusion,[7] a commercial tool for detection and analysis of code smells from both type of applications. They concluded that mobile applications have different structure and workflow as compared to desktop applications but the variety and density of code smells

[6] https://f-droid.org/en/.

[7] http://www.intooitus.com/inFusion.html.

is similar in both platforms. However, they found that the distribution of code smells is different in both platforms.

Carette et al. [44] reported an empirical study to access the impact of energy Android code smells on the performance of Android applications. They proposed an automated approach called HOT-PEPPER supported by a framework to detect and correct code smells and evaluated their impact on energy consumption on Android applications. The authors claimed that they succeeded to improve the performance of examined applications after correction of code smells.

Habchi et al. [131] conducted a study to detect code smells from iOS-specific applications. The authors focused both on object-oriented and iOS-specific code smells using adapted tool PAPRIKA. They extended PAPRIKA tool to support detection of code smells from Objective-C and Swift. The experiments are performed on 279 open source iOS applications to detect 4 object-oriented and 3 Android-specific code smells with a good accuracy. They documented the first catalog of 6 iOS-specific code smells from different blogs and other sources of literature.

Kessentiniet et al. [133] reported an automatic approach for the detection of code smells from Android applications using a multi-objective genetic programming algorithm (MOGP). They automatically generate the best set of rules that are used for the detection of code smells. A rule is a combination of quality metrics with their threshold values to detect a specific type of code smell. The approach is evaluated on 184 Android projects with source code hosted on GitHub. Authors achieved 82% accuracy while detecting 10 Android-specific code smells.

Ghafari et al. [54] presented an approach supplemented with static analysis to detect security code smells from 46,000 Android software applications. Security code smells indicate the vulnerabilities in the source code related to security properties. Authors discovered that the majority of applications suffered from at least 3 security code smells and only 9% applications were free from security code smells. Through manual analysis of 160 applications, authors realized that security code smells are good indications of security vulnerabilities.

Dennis et al. [134] proposed a static source code analysis based approach supplemented with the support of P-Lint prototype tool to detect 4 permission code smells. Permission code smells are indications of permission related bad programming practices. The major objective of authors is to provide awareness to developers related to permission standards in Android applications to avoid security vulnerabilities. We realized that P-Lint has limited user interface and limited support for data flow analysis.

Cruz et al. [55] reported an automatic approach for the detection and correction of energy code smells from Android applications. They presented a tool named as Leafactor that is applied on 140 open source Android applications for the detection of 5 energy code smells. Authors claim that they were successful to correct 222 code smells from 45 applications. They found that approximately 32% examined applications had at least one energy code smell that requires refactoring.

Grano et al. [45] presented a dataset and approach for accessing the quality of Android applications and detection of code smells. They applied PAPRIKA tool for the detection of 4 object-oriented and 4 Android-based code smells from 395 different applications. The primary focus of authors was on developing a standard dataset for Android applications based on user's feedback. The dataset has 600 versions including 280,000 user's views and 450,000 user's feedbacks.

Dustin Lim [1] presented an approach for the detection of code smells from Android applications. The author focused on 5 OO code smells namely Feature Envoy, Large Class, Long Method, Message Chain and Spethai code. He selected 5 existing well-known code smell detection tools to evaluate that how existing tools are capable to detect code smells from Android applications and compared the results of code smell detectors on the same examined Android applications. The author concluded that the existing state-of-the-art tools developed to detect code smells from OO applications have poor performance and they detect different instances of code smells from same examined applications.

Ibrahim et al. [120] presented an automatic approach for the detection and refactoring of 3 code smells from Android applications to reduce the redundancy of test case generation. They implemented their approach using Eclipse Plug-in for the detection and refactoring of code smells. Authors embark that they were successful to achieve their goal by reducing test case generation up to 28% and improving test coverage up to 5%.

Mateus et al. [46] presented an empirical study to investigate the quality of Android applications implemented in Kotlin programming language. The major focus of authors was to determine the adoption of Kotlin for development of Android applications and to measure the quality of Android applications using this new programming language introduced by Google. They applied PAPRIKA tool on 925 open source Android applications to detect 4 object-oriented and 6 Android-based code smells. Authors concluded that 11.78% Android applications are implemented in Kotlin and quality of applications developed using Kotlin increases as compared to applications developed using Java programming language. They also realized that object-oriented smells are more frequent in Kotlin as compared to Android smells.

A test smells detection approach is presented to recover test smells from different Android applications using a prototyping tool named tsDetect [47]. The author evaluated the impact of test smells on the distribution and

maintenance of Android applications. By performing extensive experiments on 656 open source Android applications, he concluded that most Android applications lack support for automated verification of the testing mechanism. He also realized that substantial test smells exist in unit test files.

Gadient et al. [63] presented an approach supplemented with a tool support Android Lint to recognize 12 security code smells from different applications. They applied a lightweight static analysis methodology that analyzes source code under development and provides just-in-time feedback about presence of security smells. They further analyzed the impact of detected security smells on vulnerabilities. They realized that about half of security smells are good indicator of security vulnerabilities. The Android Lint tool is publically available on GitHub.[8]

An approach based on concepts of data mining and source code parsing is presented to detect 13 Android code smells from different applications [49]. During the detection process, Author separates different types of files such as Java, XML and AndroidManifest files using different parsers. The results of the first step are saved in CVS files. The CVS files are used by prototyping tool named as BadDroidDetector to detect 13 Android code smells.

A generic approach based on different tools, namely Modelio, OntoUML and Protégé, is presented to automatically detect 15 smells from different Android applications [48]. Authors claim that proposed approach is capable to detect smells from code and design levels. They detected semantic and structural smells from popular Android applications.

Rubin et al. [136] presented a recent study on detection and evaluation of Android code smells from different software applications. Authors applied the concept of association rule mining to generate detection algorithms. They validated approach on manually analyzed data source code of 48 open source Android applications. Authors also conducted empirical study on a large number of Android applications to extract additional information related to Android code smells.

We come up with the following observations after in-depth and critical review of the state of the art presented in this Section.

1. Most techniques are implemented based on the concept of source code metrics and searching algorithms as indicated in Table 1. Both types of techniques have their limitations for the detection of code bad smells as discussed by Kessentini et al. [137]. The metrics-based techniques compute source code metrics itself or they use metrics extracted from other third party tools for the detection of smells. The accuracy of these techniques is dependent on the accuracy of metrics and their threshold values. There is still no consensus on threshold values among researchers. Most search-based techniques perform static analysis on the source code to detect different Android smells. These techniques also apply machine learning methods for the detection of smells. However, machine learning-based approaches depend on the quality and efficiency of data.
2. PAPRIKA and extended versions of PAPRIKA are applied by different techniques. This tool is capable to detect a limited number of Android smells and its generalization on other Android smells is questionable. ADOCTOR is applied on 15 Android smells but the accuracy of its results need investigation. The rest of the tools are not focusing on Android smells presented by Riemann et al. [5].
3. It is evident from data presented in Table 1 that most authors performed experiments on a subset of smells related to OO and/or Android code smells and generalization of their results for these approaches are questionable. There is no single approach that focuses on all Android bad smells of Riemann et al. [5] for detection. This poses a serious threat to empirical studies that base their research on very low number of smells.
4. The detailed results of the presented approaches are not publically available for analysis and comparison of results. In such cases, researchers cannot compare the results of these approaches for evaluation of existing tools and for the development of new tools.
5. Most authors performed experiments on different open source projects/applications and undertake different code smells for the detection. In such cases, the analysis and comparison of results become arduous when tools are not publically available.
6. Most published papers on Android smells are from 2014 to 2019 and they are from conference proceedings. It indicates that filed of Android code smells detection is still at the stage of infancy and reaching maturity. It also indicates that due to rising number of mobile applications, the detection of refactoring of Android bad smells may be a promising area for researchers.
7. A few approaches measure the accuracy of detected code bad smells as depicted in Table 1. The accuracy is a key aspect for the application and adoption of approaches for researchers and practitioners.
8. The target language for most subject systems under experimentations is Java language.

[8] https://github.com/pgadient/AndroidLintSecurityChecks.

Table 2 Specification of selected Android bad smells

Android bad smells	Definition	Specification/detection rule
Durable WakeLock (DW)	A Durable WakeLock exists if the resources such as CPU, Network and sensors used by WakeLock are not released by calling release method after their utilization	Co (M_AC) >= {1} AND Co (M_RE) = ∅
Inefficient Data Structure(IDS)	The application of HashMap as Integer to object mapping is expensive and slow in Android applications	Co (HM as T(key) = Integer) AND (T(value) = Object) = {1}
Inefficient Data Format and Parser (IDFP)	Tree parsers like XML parsers are too expensive and complex for mobile systems. An Inefficient Data Format and Parser is defined in terms of usage of Document Builder Factory and Node List	Co(DBF) OR Co(DB) OR Co(NL) >= {1}
Internal Getter/Setter (IGS)	Accessing fields in terms of getters and setters internally is expensive and three times slow process	Co(M_Set) >= {1} OR Co(M_Get) >= {1}
Leaking Inner Class (LIC)	Non-static inner classes holding a reference of their outer class causes a memory leak	Co(R_O_C) >= {1}
Leaking Thread (LT)	Threads are used to share responsibilities but starting a thread without stopping it properly after completion of its work is an improper use of resources	Co(M_STA) = {1} AND Co(M_STO) = ∅
Member Ignoring Method (MIM)	A method defined in a class without using properties of its class is an indication of structural programming	In MB Co(FACC) = ∅
No Low Memory Resolver (NLMR)	Usually, mobile systems are assembled with little RAM and no Virtual memory for swap space. In Android Applications, a method onLowMemory is used to kill the processes in order to reclaim any part of memory. Absence of overrideable method onLowMemory is an indication of poor design implementation	Co(OLM) = ∅
Public Data(PD)	Private data is stored in the publically accessible area of application and it can be changed by other applications installed on the device	Co(MWR) >= {1} OR Co(MWW >= {1}
Rigid Alarm Manager (RAM)	In Android applications, an alarm manager is used to define operations to be executed at a specific time in the future. It is quite helpful if Android developers use setInexactRepeating method instead of setRepeating in order to bundle several updates together as it optimized resource utilization	Co(SR) >= {1}
Slow Loop(SL)	In Android applications, traditional for loop is very expensive and slow, so an enhanced version of for loop should be used	Co(TFL) >= {1}
Unclosed Closable(UC)	In Java, a closeable interface is used to close all resources being used after the job is done. Missing the call to close method defined in closeable is considered bad practice	Co(MC) = ∅
Static Views (SV)	Holding static reference of any view type object is a resource (memory) intensive process	(FM is static) AND (FT >= {1})AND (Co(F) >= {1})
Static Context (SC)	In Android applications, declaring field of type Context leads to memory leak as it is never collected by garbage collector of JVM	(FM is static) AND (FT is Context) AND (Co(F) >= {1})
Static Bitmap (SB)	Bitmaps are heavy objects and should be dealt with proper case otherwise it can lead to memory leaks. Holding static reference of the bitmap is a resource extensive task	(FM is static) AND (FT is Bitmap) AND (Co(F) >= {1})

Table 2 (continued)

Android bad smells	Definition	Specification/detection rule
Collection of Views (CV)	The collection of views is a resource intensive process in Android applications and should be avoided	(FT is AC) AND (CT is AV)
Collection of Bitmaps (CB)	The collection of bitmaps is a resource intensive process and should be avoided	(FT is AC) AND (CT is Bitmap)
Debuggable Release (DR)	In Android applications, debuggable is an attribute defined in Manifest file as true for development mode and false for release mode. Setting debuggable true in release mode is a serious security threat	AT(AD) = {true}
Dropped Data(DD)	In Android applications, the data filled by user in any Activity or fragment may be lost if the focused screen is interrupted. Data should be saved and restored using overridable methods on SaveInstanceState and on RestoreInstanceState	Co(OSIS) = ∅ AND Co(ORIS) = ∅
Untouchable(Tn)t	Any touchable view in Android application should be greater than 48dp (density independent pixel) by both width and height	ALW < 48dp && ALH < 48dp
Uncontrolled Focus Order(UFO)	The usage of default (up, down, right and left directional) control for navigation is not recommended as in some cases it might be illogical	Co(ANFD) = ∅ AND Co(ANFU) = ∅ AND Co(ANFR) = ∅ AND Co(ANFL) = ∅
Nested Layout (NL)	In Android UI, the layouts having elements with attribute weight must be computed twice. This particular phenomenon increases the computation time exponentially. Mostly, the Linear Layouts creates that problem	Co(LL(AT(ALWT))) = 1
Not Descriptive UI (NDUI)	Android provides an app named as TalkBack used for visually impaired people to keep track of navigation and keep track of applications that user is currently using. TalkBack maintains navigation track by reading the contents which are defined as contentDescription in UI views. Missing of contentDescription indicates poorly descriptive UI	Co(ACD) = ∅
Set Config Changes (SCC)	In Android applications, it is considered a code smell if attribute android:configChanges is defined in Manifest file. This phenomenon leads to memory leaks	Co(ACC) >=1
Overdrawn Pixel (OP)	In Android application design, most of the UI part is built using XML layouts that consist of a stack of several UI elements. In a stack of UI elements, it might be possible to overdraw a pixel multiple times using Android:background property which is an extensive task for GPU	Co(Ch(AL)) > 0 AND Co(BGC(AL)) = 1 AND Co(BGC(ACV)) = 1

T Type, *Co* Count, *HM* HashMap, *M_AC* Method Acquire, *M_RE* Method Release, *DBF* Document Builder Factory, *DB* Document Builder, *NL* NodeList, *M_Set* Method Setter, *M_Get* Method Getter, *R_O_C* Reference of Outer Class, *M_STA* Method Thread Start, *M*_STO Method Thread Stop, *FACC* Field Access, *MB* Method Body, *OLM* onLowMemory, *MWR* MODE_WORLD_READABLE, *MWW* MODE_WORLD_WRITEABLE, *SR* setRepeating, *TFL* Traditional For Loop, *MC* Method Close, *FM* Field Modifier, *F* Field, *FT* Field Type, *AC* Any Collection like List, Map etc., *CT* Collection Type, *AV* Any View, *AA* Android:Attribute, *ALW* Android:LayoutWwidth, *ALH* Android:layout_height, *ANFD* Android:Next Focus Down, *ANFU* android:nextFocusUp, *ANFR* Android:nextFocusRight, *ANFL* Android:NextFocusLeft, *ALWT* Android:layout_weight, *ACD* Android:Content Description, *ACC* Android:ConfigChanges, *Ch* Child, *AL* Any Layout, *ACV* Any Child View, *BGC* Background Color

3 Specifications of Android Code Smells

The standard and accurate specifications of Android code smells are backbone for their accurate detection and correction. A catalog on definitions of 30 Android code smells is presented by Riemann et al. [5]. We undertake the Android code smells presented by authors of this catalog for specification and detection. The definitions of some code

smells presented in this catalog are at very abstract level. This catalog also does not have information about metrics and detection rules that may be used for the detection of these all smells. Due to these reasons, we selected most Android smells from this catalog for the specification. The rest of 5 Android code smells are selected from the state of the art and different blogs. The summarized information about selected smells is presented in Table 2. We uploaded detailed definitions and detection rules of Android code smells presented in Table 2 on the web [104].

The source code metrics are very important to measure the features of software processes and products [138]. They are used to measure accuracy, flexibility, extensibility, complexity and other properties of software applications. They have been proven useful for measuring the quality of object-oriented and Android applications [139–144]. The source code metrics can be used to locate possible issues and propose improvements for the quality and efficiency of software applications. The dimensional, complexity, object-oriented and Android-oriented metrics are discussed by Mercaldo et al. [145]. These different types of source code metrics are the key artifacts used for the detection of different code smells. Rehman et al. [146] presented 21 static source code metrics to the predict security and privacy risks of Android applications. Authors extracted these metrics from third party tool SonarQube [147]. They did not explain that which actual Android smells are detected using these metrics. It is realized that third- party libraries are widely used in Android applications. These libraries facilitate developers but they make analysis of source code and detection of smells challenging.

We elicited source code metrics related to Android code smells from different sources as shown in Table 3. We want to clarify that all source code metrics presented for the detection of object-oriented code smells are not capable to detect all Android code smells. Although, many source code metrics are common for accessing the quality of OO and Android mobile applications. However, there are very few source code metrics that have been proven to work for Android applications such as User Interface complexity, Broadcast Receivers, etc. The source code metrics for analysis of Android applications and detection of Android smells are discussed by different authors [52, 84, 146, 148].

4 Detection Approach

Based on the intensive state of the art in the field of Android code smell detection discussed in Sect. 2, we present our approach in this section. Our approach is capable to detect 25 Android code smells and is flexible for the extension to detect other types of code smells. We used the concept of source code parsing for the implementation of our approach. The definitions of Android code smells, source code metrics and detection rules presented in Sect. 3 are the backbones for the presented approach. The architecture of the proposed approach is given in Fig. 1.

4.1 Source Code Parsing

The source code analysis is the key step for the detection of bad smells using our approach. For this purpose, we used an open source JavaParser for analyzing source code and creating a customized program data model. The program data model consists of the main elements and relations among these elements in the source code. The elements are software artifacts or constructs such as classes, interfaces, methods, fields and relationships. The relationships among these core components are structural or behavioral properties. JavaParser [149] provides a simple and lightweight set of tools for analysis of Java source code. JavaParser is implemented in Java programming Language.

JavaParser is used for preprocessing of Java source code with visitor methods to visit different components of source code. It is an open source tool available under the Apache License terms and conditions. It is an independent set of lightweight tools used to generate, analyze and process Java source code. JavaParser is the latest commit in the family of Java Parsers and being used in many open source and commercial projects.

JavaParser generates an AST that provides a structural representation of Java source code. It provides an abstract VoidVisitorAdapter with a set of visitor methods in order to visit different components of the source code. All visitor methods are overloaded methods with the difference in their parameters. The prototypes of two visitor methods in VoidVisitorAdapter are given below:

- public void visit (final MethodDeclaration n, final A arg)
- public void visit (final ConstructorDeclaration n, final A arg).

The first visitor method “visit” is used to visit all definitions of methods in a given class and the second visitor method “visit” is used to visit all Constructor declarations in the given class. Similarly, there are 90 visitor methods to visit different parts of source code such as classes, method declarations, field declarations, constructors, loops, conditions, comments, annotations and object creations. JavaParser also provides different individual compilation units used for the analysis of source code by parts given in a string format.

A directory handler named as Directory Explorer is used to iteratively visit all the folders of a given input path. The Directory Explorer filters all Java files from the input set of directories and passes them to Visitor Adapter to visit all

Table 3 Source code metrics for Android-specific smells

Android smells	Source code metrics used for detection
Member Ignoring Method	NSM, AM, FACC
Internal Getter/Setter	M_Set, M_Get
Public Data	MP, MWR, MWW
Rigid Alarm Manager	SRA, MC, SR
Debuggable Release	DT, AT, AD
Dropped Data	RB, OSIS, ORIS
Durable WakeLock	GPS, WKL, M_AC,M_RE
Inefficient Data Structure	HM, Obj
Inefficient Data Format and Parser	Network, IO, XML, DBF, DB, NL
Leaking Inner Class	NSIC
Leaking Thread	M_STA, M_STO, TS
Nested Layout	UI, NL, LL, ALWT
Unclosed Closable	OCI, MC
No Low Memory Resolver	OAM, OLM
Not Descriptive UI	ICD, ACD
Overdrawn Pixel	UI, BGC, ACV, AL
Set Config Changes	OCC, ACC
Slow Loop	NIFL, TFL
Uncontrolled Focus Order	UIP, ANFD, ANFU, ANFR, ANFL
Untouchable	UI, UX, ALW, ALH
Static Views	FM, FT
Static Context	FM, FT
Static Bitmap	FM, FT, BM, FS
Collection of Views	FT,AC, CT, AV
Collection of Bitmaps	FT, AC,CT, BM
Data Transmission Without Compression	DS, ZF
Prohibited Data Transfer	Network, IO
Unnecessary Permission	UI, UX
Interrupting from Background	Services, BR [52]
Inefficient SQL Query	JDBC, JSON
Tracking Hardware Id	CLI
Early Resource Binding	LM, LS
Bulk Data Transfer on Slow Network	BS, NB
Network & IO Operations In Main Thread	MT, IO

DS Data Space, *NB* Network Bandwidth, *ZF* Zip Format, *DT* debuggable True, *RB* Restore Bundle, *CPU* Central Processing Unit, *GPS* Global Positioning System, *LM* Location Manager, *LS* Location Service, *HM* Hash Map, *JDBC* Java Database Connection, *JSON* Java Server Object Notation, *IO* Input/Output, *BR* Broadcast Receiver, *NSIC* Non-static Inner Classes, *TS* Thread Start & Stop, *NSM* Non-Static Methods, *AM* Accessing Members, *UI* User Interface, *NL* Nested Layout, *MT* Main Thread, *OAM* Overridden Activity Method, *ICD* Ignoring Content Description, *MP* Mode Public, *SRA* Set Repeating Alarm, *OCC* On Configuration Changed, *NIFL* Native Iterative For Loop, *CLI* Context Layout Inflator, *OCI* Opened Closeable Interface, *UIP* UI Parameters, *UX* User Experience, *WKL* WakeLocks with no Timeout, *Obj* Object, *DBF* Document Builder Factory, *DB* Document Builder, *NL* Node List, *M_Set* Method Setter, *M_Get* Method Getter, *M_STA* Method Thread Start, *M_STO* Method Thread Stop, *FACC* Field Access, *ALWT* Android: layout Width, *LL* Linear Layout, *OLM* On Low Memory, *ACD* Android: Content Description, *AL* Any Layout, *BGC* Background color, *MWR* MODE_WORLD_READABLE, *MWW* MODE_WORLD_WRITEABLE, *SR* SetRepeating, *ACC* Android:Config Changes, *TFL* Traditional For Loop, *MC* Method Close, *ANFD* Android: Next Focus Down, *ANFU* Android: Next Focus Up, *ANFR* Android Next Focus Right, *ANFL*Android Next Focus Left, *ALW* Android Layout_Width, *ALH* Android Layout_Height, *F* Field, *FT* Field Type, *AC* Any Collection like List, Map, etc., *CT* Collection Type, *AV* Any View, *BM* Bitmap, *FS* File Size)

Fig. 1 Architecture of detection approach

classes and other source code artifacts. In Visitor Adapter, we collect source code data model by collection of source code artifacts.

We want to clarify that JavaParser has some limitations: First, it is not easy to customize the existing features of JavaParser. Most of the implemented classes in JavaParser are immutable classes and we cannot extend or modify these classes. Second, JavaParser lacks attribute level analysis and is unable to identify the real scope of declared attributes in the case of inner classes. The attributes analysis scope resolution is very important for the detection of Android code bad smells such as Internal Getter/Setter. Finally, JavaParser has a symbol solver named as Javasymbolsolver. The Javasymbolsolver is a collection of ReflectionTypeSolver and JavaParserTypeSolver. It is used for scope and type resolution. Currently, the integrated symbol solver is immature and still under development. To overcome these limitations, a developer has to identify the parent of all source code components and assign scope levels accordingly. In order to determine the true scope of attributes, we filtered the attributes list in all the classes by comparing their ancestor type (using method getAncestorType()) with the class name. Through this step, we were able to resolve the problem of the scope of attributes within classes.

JavaParser is not able to parse XML documents and to extract information from XML documents. In our experimental setup, 7 Android code smells selected for detection are based on XML files (containing both meta-data (Manifest file) and layout files (Design files). The smells with such information cannot be detected directly using JavaParser. The motivation for using DOM parser is to extract these 7 smells given below:

1. Debuggable Release (Detected from Manifest file)
2. Set ConfigChanges (Detected from Manifest file)
3. Not Descriptive UI (Detected from Layout file)
4. Nested Layout (Detected from Layout file)
5. Untouchable (Detected from Layout file)
6. Uncontrolled Focus Order (Detected from Layout file)
7. Overdrawn Pixel (Detected from Layout file).

The Document Object Model DOM is a hierarchy-based parser used to create an object model against the whole XML document. Conventionally in the software industry, DOM is used to parse the XML documents rather than SAX parser and JAXB (Java Architecture for XML Binding). The DOM breaks down the entire document into the following three pieces.

1. Elements (tags)
2. Attributes
3. Data (values)

The above-mentioned entities are represented by the Node class. Nodes can be accessed by using getNodeName, getNodeType and getNodeValue. Using DOM, we implemented parse level detection, i.e., detection of Android bad smells. DOM Parser is available at the link.[9]

4.2 Detection Example: Leaking Thread

Leaking Thread is a code bad smell specific to Android mobile applications. It is a result of improper implementation of threads while distributing responsibilities. Sometimes, developers write code that starts a thread in a class but they forget to define a mechanism for stopping the thread.

We explain the presence of Leaking Thread code bad smell in the source code of an application as shown in Listing 1. Here, we can see a thread object with name smartWorker that is created in WorkHelper class. Before closing statement of class, the created thread smartWorker is started with method start(). It is necessary to stop a thread after completion of its job. We cannot see any mechanism of stopping the smartWorker by using stop() method in the example as shown in Listing 1.

[9] https://dom4j.github.io/.

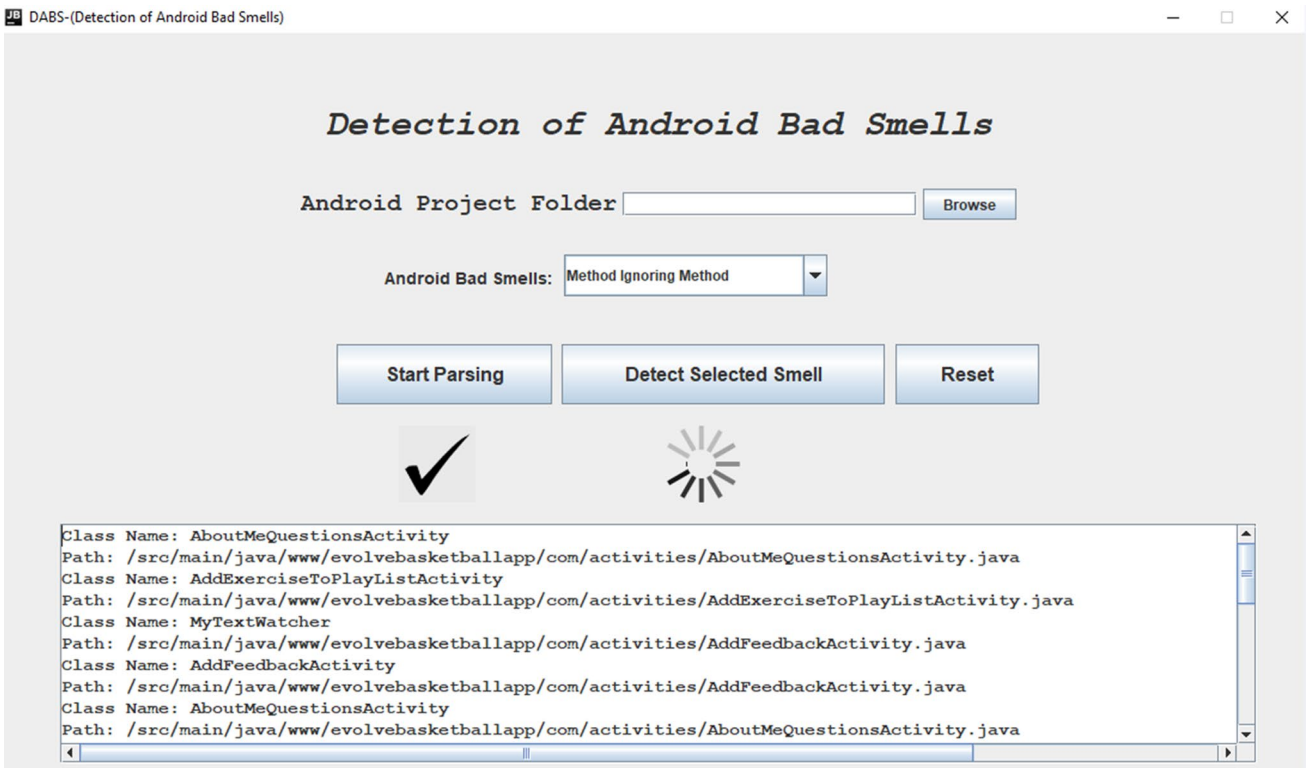

Fig. 2 GUI of prototyping tool

```
Public class WorkHelper {
Thread smartWorker = new Thread() {
  Public void run() {
  Try {
doMyTask();
} catch ( InterruptedException ex) {
    //handle the exception
    }
  }
};
smartWorker.start();
}
```

Listing 1: Example of Leaking Thread Bad Smell

Our approach and ADOCTOR report a successful detection in the above source code example. Following is the detection rule for Leaking Thread considering class C is using a thread named as T using our proposed approach.

- Class C is using T by calling run method.
- There is an absence of stop method for stopping T.

We present another example to show how our results are different from ADOCTOR while detecting Leaking Thread bad smell as shown in Listing 2. Our approach reports zero instance for Leaking Thread bad smell from the above source code as shown in Listing 2. The reason is that our approach is implemented based on method calling and filtering comment sections. The ADOCTOR reports one instance for Leaking Thread on the same source code as the approach implemented in ADOCTOR is based on simple text comparison and there is no method for filtering comments.

```
Public class WorkHelper {
//Thread smartWorker = new Thread() {
//Public void run() {
//Try {
//doMyTask();
// } catch ( InterruptedException ex) {
//     handle the exception
//    }
// }
//};
//smartWorker.start();
}
```

Listing 2: Example of Leaking Thread Bad Smell with Comments

4.3 Prototyping Tool

We developed a prototyping tool to evaluate and realize the concept of our proposed approach. The prototype is implemented in Java programming language. It is a stand-alone tool but it can be integrated with other tools as a Plug-in.

Table 4 Brief description and statistics of selected open source and industrial android applications

Android application	Type	Description	Java files	Packages	Classes	Methods	Attributes
LeafPic	Open Source	LeafPic is an open source Android App, providing an alternative to the native gallery	131	23	171	1172	624
AmazeFileManager	Open Source	It is a utility Android App providing basic features to its users such as cut, copy, paste, delete, compress and extract	207	45	279	1846	1387
Easy Sound Recorder	Open Source	It is an open source simple material designed sound recording app	14	5	19	105	76
MLManager	Open Source	It is an open source Android application providing customizable APK manager	18	6	20	171	93
Evolve Basketball App	Industrial/Commercial	It is a training application and has training features to become a good player	1261	183	5079	9989	213,817
SANAD	Industrial/Commercial	It is designed and developed for all types of labor facilities	52	5	78	588	1780
Foga	Industrial/Commercial	Foga is an Android application providing social interaction with friends and families	200	27	290	2323	5954

Currently, the prototype is capable to detect 25 Android-based code bad smells but it can be extended for other types of code smells. The tool is publically available on the Web. To the best of our knowledge, there is no freely available tool that is capable to detect these 25 Android code bad smells. Figure 2 depicts the graphical user interface of our prototyping tool. A user can browse the path of desired source code directory and desired smell(s) for detection. After this, the parsing step is executed for analysis of source code and creation of program data model. The user has the option to select one or all code bad smells for the detection. The bottom text of the tool is a customized console for detailed analysis of detected results. The user can view directly the parts of source code that have bad smells.

4.4 Limitations of Approach

The presented approach relies on open source JavaParser for parsing source code and to extract information related to different Android code bad smells. If a file has syntax errors in any part of source code that is participating in the definition of an anti-pattern, then it may generate false positives or false negatives. We want to clarify that there will be no threat for non-availability of technique due to this problem. Secondly, the extension of our approach requires the creation of new parsers for other languages, extensive knowledge of source code metrics, definitions of Android code smells, threshold values for the detection of Android code smells and knowledge of data model used for intermediate representation. However, a user can extend our approach by adding definitions and detection algorithms for new Android bad smells. Java Parser requires source code that is error free and is executable.

5 Evaluation

Evaluation of our proposed approach is imperative to measure its accuracy, effectiveness, performance and quality. At the start, we performed experiments on small examples of Android APIs for testing definitions of selected code smells. We refined our definitions based on the initial results. After initial experiments, we selected 4 open source and 3 industrial Android-based applications of varying size for our further experimental evaluation. The open source applications are available freely on the web. The size for open source code applications with respect to number of classes ranges from 19 to 279 classes. The industrial applications are taken from software industry based on the commitment that they will be kept secret due to license issue. The size for Industrial applications in terms of number of classes varies from 78 to over 5000 classes. The objective of selecting both types of applications is to determine the accuracy and performance of our approach for all type of software applications. The selected Android applications are from different categories such as entertainment, management, graphics, blockers, monitoring and social. Table 4 summarizes key statistics of Android applications containing name, type, brief description and key source code statistics of each examined application.

We compare the results of our approach with state-of-the-art approach [42]. We tried to run PAPRIKA tool for comparison of our results but we cannot succeed to execute the

Table 5 Detection results for Android bad smells

Bad smells/projects	Evolve Basketball		SANAD		Doctors		LeafPic		AmazeFileManager		Easy sound recorder		Foga	
	[PA]	[42]	[PA]	[42]	[PA]	[42]	[PA]	[42]	[PA]	[42]	[PA]	[42]	[PA]	[42]
MIM	314	261	25	28	62	53	96	54	158	107	14	10	146	111
NLMR	96	62	12	4	26	3	11	3	8	22	2	7	40	32
IGS	71	8	6	2	3	2	11	5	28	8	9	1	19	4
IDFP	6	6	3	3	0	0	0	0	0	0	0	0	3	2
IDS	0	0	2	0	0	0	3	0	4	0	1	0	0	0
LT	0	38	0	6	0	1	0	6	1	10	0	2	2	31
SL	82	40	26	7	17	14	45	14	107	39	1	1	92	35
PD	0	1	0	1	0	1	1	1	1	1	1	1	0	0
DW	3	3	1	1	2	3	1	1	2	2	1	1	5	6
RAM	4	4	1	1	2	2	1	1	2	2	1	1	5	5
UC	11	26	3	0	6	1	2	11	7	6	2	1	3	9
LIC	9	123	3	9	4	19	2	22	3	40	2	5	5	32
DR	1	551	0	0	1	0	1	180	1	238	0	0	1	0
SV	2	N/A	4	N/A	1	N/A	0	N/A	0	N/A	3	N/A	11	N/A
SC	3	N/A	3	N/A	2	N/A	0	N/A	1	N/A	2	N/A	3	N/A
SB	0	N/A	0	N/A	0	N/A	0	N/A	0	N/A	1	N/A	5	N/A
CV	3	N/A	4	N/A	8	N/A	0	N/A	6	N/A	0	N/A	28	N/A
CB	0	N/A	0	N/A	0	N/A	0	N/A	0	N/A	0	N/A	1	N/A
DD	140	N/A	12	N/A	26	N/A	19	N/A	20	N/A	8	N/A	49	N/A
Untouchable	120	N/A	16	N/A	1	N/A	0	N/A	15	N/A	1	N/A	56	N/A
UFO	219	N/A	56	N/A	46	N/A	43	N/A	157	N/A	6	N/A	101	N/A
NL	44	N/A	16	N/A	1	N/A	24	N/A	22	N/A	1	N/A	15	N/A
NDUI	221	N/A	50	N/A	42	N/A	45	N/A	162	N/A	6	N/A	99	N/A
SCC	61	N/A	0	N/A	0	N/A	7	N/A	0	N/A	2	N/A	2	N/A
OP	134	N/A	47	N/A	30	N/A	3	N/A	3	N/A	0	N/A	91	N/A

MIM Member Ignoring Method, *NLMR* No Low Memory Resolver, *IGS* Internal Getter/Setter, *IDFP* Inefficient Data Format and Parser, *IDS* Inefficient Data Structure, *LT* Leaking Thread, *SL* Slow Loop *PD* Public Data *DW* Durable WakeLock, *RAM* Rigid Alarm Manager, *UC* Unclosed Closable, *LIC* Leaking Inner Class *DR* Debuggable Release, *SV* Static Views, *SC* Static Context, *SB* Static Bitmap, *CV* Collection of Views, *CB* Collection of Bitmaps *DD* Dropped Data *Un*t Untouchable, *UFO* Uncontrolled Focus Order, *NDUI* Not Descriptive UI, *NL* Nested Layout, *SCC* Set Config Changes, *OP* Overdrawn Pixel, *PA* Presented Approach

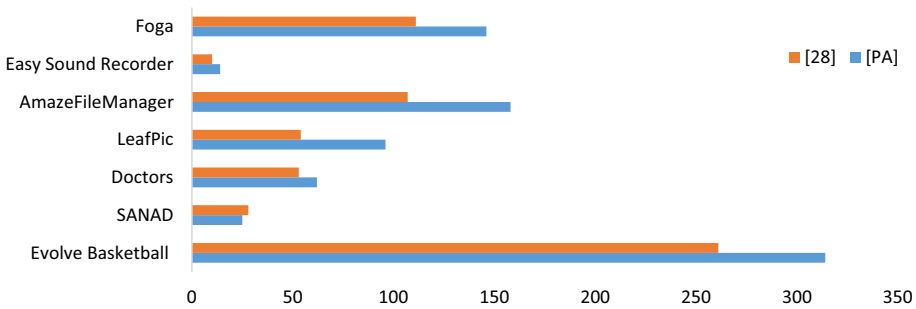

Fig. 3 Frequency of MIM code smell

tool due to its configuration issues. The authors of PAPRIKA were requested for help but they said that the tool is still under development and upgradation. The manual comparison of results for each smell was very time-consuming and daunting task. We tried our best to compile manual results with great care. There is a wide disparity in the results of our approach as compared to approach [42]. The reasons of disparity are different detection criteria applied by ADOCTOR in the case of different Android code bad smells. The entry of "N/A" in Table 5 means that the tool is not able to detect that particular code bad smell and comparison is not possible.

The results extracted from each selected Android application are presented in Table 5. Table 5 presents a comprehensive comparison of detected results extracted from ADOCTOR [42] and our approach on the same applications. It is visible from the results that "*Member Ignoring Method*" code smell has maximum occurrences in all selected applications. This information is very valuable for researchers working on refactoring of Android code smells. We also discover that our approach extracted more instances of this particular smell from all applications except "SANA" application as compared to ADOCTOR [42]. The reason for this variation in results is due to different detection criteria applied by our approach. The approach adopted by ADOCTOR detects only one instance of "*Member Ignoring Method*" smell in each class but our approach detects all instances of this smell from one class.

Figure 3 presents graphical representation of the results from all applications for "*Member Ignoring Method*" smell. Similarly, we can see that "*Not Descriptive UI*" and "*Uncontrolled Focus Order*" code smells have maximum occurrences in all applications. We notice that "*Public Data*" and "*Inefficient Data Structure*" code smells have minimum occurrences in all selected applications. We cannot identify the reasons for low frequency of these smells. Finally, we also realize that proportion of occurrences of all code smells in open source and Industrial applications is approximately equal. This fact gives us confidence on the generalization of our results for both type of applications. The manual analysis of results to check precision, recall and *F*-Measure for 4 applications further strengthen confidence of generalization of our results.

It can be observed that the major difference in results is in the case of "*Leaking Inner Class*," "*No Low Memory Resolver*," "*Internal Getter Setters*," "*Slow for Loop*" and "*Unclosed Closable and Leaking Inner Class*." We analyzed the following reasons for the variations in the results of these bad smells extracted by our approach and ADOCTOR as given in Table 6.

The variation in results of other bad smells is due to the reason of weak detection strategy implemented by ADOCTOR [42]. The ADOCTOR detects most of the bad smells by just comparing the text of the whole class with the definition of a bad smell. For example, ADOCTOR detects a "*Leaking thread* "bad smell by just searching a presence of text "run()" and absence of text "stop()." The ADOCTOR has no capability to find the call run() for an object of Thread class which is a true definition of *Leaking thread* bad smell. Similarly, the detection strategy for other bad smells implemented in ADOCTOR is very weak and volatile that leads to a large number of false positives in detected results. We extracted 114 instances of "*Debuggable Release"* bad smell from *Evolve Basketball* App using ADOCTOR tool. There can be only one instance of *Debuggable Release* code smell in one application according to the nature of this smell. This is a bug in ADOCTOR tool.

Due to the wide variation in the results of our approach as compared to ADOCTOR, we further analyzed the shared instances of bad smells. For this purpose, we compared the result of our approach from all examined applications on each smell with ADOCTOR. Table 7 presents the shared instances of bad smells extracted by both approaches. These shared instances are extracted automatically by comparing the results of both approaches. We implemented a separate class in our tool for this comparison because the manual comparison of shared instances was a very time-consuming task.

We can see from Table 7 that a number of bad smells have shared instances detected by both approaches but it is still not clear that which instances are exactly shared. In order to view extract shared instances, we went one step further to analyze shared instances of bad smells extracted by both approaches. Tables 8 and 9 present these exact shared instances with complete path of source code for *Easy Sound Record* and *SAND* applications, respectively. In order to curtail space, we mention only one shared instance of bad smells that are extracted more than once from same Java class. For example, 9 instances of "Internal Getter/Setter" bad smell are extracted by our approach from FileViewerAdapter.java class but we mention only one instance in Table 8. The rest of 8 instances are extracted by our approach from same class but they are not extracted by ADOCTOR [42].

We evaluated the accuracy of our approach by using well-established metrics *j* such as precision, recall and *F*-measure given below:

$$\text{Precision} = \text{TP}/(\text{TP} + \text{FP})$$

$$\text{Recall} = \text{TP}/(\text{TP} + \text{FN})$$

$$F\text{-Measure} = 2 * (\text{Precision} * \text{Recall})/(\text{Precision} + \text{Recall})$$

TP represents the number of true positive instances of smells detected by our approach and FP represents the number of wrongly identified instances of code smells. FN is the number of correct instances of smells not identified by our approach. There may be a trade-off between precision and recall metrics, and therefore, we measure the accuracy of our approach by using harmonic means of precision and recall known as *F*-measure. We concede that our approach has false positive and false negative instances detected in the case of few code smells as indicated in Table 10. For example, our approach is unable to detect "Member Ignoring Method" bad smell when the method has a local variable or parameters with the same name as the field within the body of the method. In such scenario, our approach assumes that the method is actually accessing and using the class field and it cannot differentiate between the local variable and the class field with the same name. As a result, our approach

Table 6 Reasons for variation in results

Android smell	Our approach	ADOCTOR [42]
MIM	Detects all instances in one class	Detects only one instance from one class
SL	Detects all slow for loops in any class	Detects only one slow for loop in one class
IGS	Detects all internal Getter Setter calls in any class	Detects only one setter or getter call in one class. ADOCTOR also uses a simple text comparison technique by calling equals() method defined in Object class
LIC	Detects when a non-static inner class has a reference of its outer class. A true definition of Leaking Inner class according to Reimann et al. [5]	Detects when there is a presence of non-static inner class found in a Java class
NLMR	Detects by finding an absence of overridden method onLowMemory in any Activity or Fragment type Java class	Detects by using regular expression "onLowMemory" with an input of whole text of Java class
IDS	Detects by finding an instance field or variable declaration of type HashMap with key as an Integer or Long and value as String	Detects by using regular expression (.*)HashMap < (\\s*)(Integer|Long)(\\s*), (\\s*)(. +) (\\s*) > with an input of whole text of Java class
LT	Detects by finding a presence of call to method run and an absence of call to method stop defined in Thread class	Detects by just comparing the whole text of Java class with Strings "run()" and "stop()" using contains method defined in Object class of Java
DW	Detects by finding a presence of call to method acquire and an absence of call to method release defined in PoweManager's WakeLock class	Detects by using regular expression (.*)acquire(\\s*)() with an input of whole text of method
RAM	Detects by finding a presence of call to method setRepeating defined in AlarmManager class	Detects by comparing the whole method text with Strings "AlarmManager" and "setRepeating"
UC	Detects by finding the implementation of Closeable interface with the absence of method close in any Java class	Detects by using regular expression (.*)close(\\s*)() with an input of whole text of Java class
DR	Detects when an attribute Android:debuggable in application tag is found with true value in only AndroidManifest.XML file using XML DOM Parser	Detects by comparing simple text of Android:debuggale="true" in whole project

MIM Member Ignoring Method, *SL* Slow Loop, *IGS* Internal Getter/Setter, *LIC* Leaking Inner Class, *NLMR* No Low Memory Resolver, *IDS* Inefficient Data Structure, *LT* Leaking Thread, *DW* Durable WakeLock, *RAM* Rigid Alarm Manager, *UC* Unclosed Closable, *DR* Debuggable Release

Table 7 Shared instances of Android bad smells

Bad smells/ systems	Evolve Basketball			SANAD			Doctors			LeafPic			AmazeFileManager			Easy sound recorder			Foga		
	[PA]	[42]	SS	[PA]	[42]	SS	[PA]	[42]	SS	[PA]	[42]	SS	[PA]	[42]	SS	[PA]	[42]	SS	[PA]	[42]	SS
MIM	314	261	193	25	28	18	62	53	41	96	54	30	158	107	80	14	10	8	146	111	92
NLMR	96	62	19	12	4	2	26	3	1	11	3	0	8	22	2	2	7	1	40	32	20
IGS	71	8	8	6	2	1	3	2	2	11	5	5	28	8	7	9	1	1	19	4	4
IDFP	6	6	6	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0	3	2	2
IDS	0	0	0	2	0	0	0	0	0	3	0	0	4	0	0	1	0	0	0	0	0
LT	0	38	0	0	6	0	0	1	0	0	6	0	1	10	0	0	2	0	2	31	2
SL	82	40	40	26	7	6	17	14	12	45	14	14	107	39	39	1	1	1	92	35	33
PD	0	1	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	0	0
DW	3	3	3	1	1	1	2	3	2	1	1	1	2	2	2	1	1	1	5	6	5
RAM	4	4	4	1	1	1	2	2	2	1	1	1	2	2	2	1	1	1	5	5	5
UC	11	26	0	3	0	0	6	1	0	2	11	0	7	6	0	2	1	0	3	9	0
LIC	9	123	9	3	9	3	4	19	4	2	22	2	3	40	3	2	5	2	5	32	5
DR	1	551	1	0	0	0	1	0	0	1	180	1	1	238	1	0	0	0	1	0	0

SS Shared Smells, *PA* Presented Approach, *MIM* Member Ignoring Method, *NLMR* No Low Memory Resolver, *IGS* Internal Getter/Setter, *IDFP* Inefficient Data Format and Parser,, *IDS* Inefficient Data Structure, *LT* Leaking Thread, *SL* Slow Loop *PD* Public Data *DW* Durable WakeLock, *RAM* Rigid Alarm Manager, *UC* Unclosed Closable, *LIC* Leaking Inner Class *DR* Debuggable Release

Table 8 Exact shared instances from Easy sound record

Bad smells	Source CodePath	PA	[42]
Member Ignoring Method	/app/src/main/java/com/danielkim/soundrecorder/activities/MainActivity.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/activities/MainActivity.java	✓	✗
	/app/src/main/java/com/danielkim/soundrecorder/activities/SettingsActivity.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/adapters/FileViewerAdapter.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/DBHelper.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/DBHelper.java	✓	✗
	/app/src/main/java/com/danielkim/soundrecorder/fragments/LicensesFragment.java	✓	✗
	/app/src/main/java/com/danielkim/soundrecorder/fragments/PlaybackFragment.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/fragments/SettingsFragment.java	✓	✗
	/app/src/main/java/com/danielkim/soundrecorder/listeners/OnDatabaseChangedListener.java	✓	✗
	/app/src/main/java/com/danielkim/soundrecorder/model/Recordings.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/RecordingItem.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/RecordingService.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/RecordingService.java	✓	✗
No Low Memory Resolver	/app/src/main/java/com/danielkim/soundrecorder/activities/MainActivity.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/activities/SettingsActivity.java	✓	✗
Internal Getter/Setter	/app/src/main/java/com/danielkim/soundrecorder/adapters/FileViewerAdapter.java	✓	✓
Inefficient Data Structure	/app/src/main/java/com/danielkim/soundrecorder/activities/SettingsActivity.java	✓	✗
Slow For Loop	/app/src/main/java/com/danielkim/soundrecorder/activities/SettingsActivity.java	✓	✓
Public Data	/app/src/main/java/com/danielkim/soundrecorder/MySharedPreferences.java	✓	✓
Durable WakeLock	/app/src/main/java/com/danielkim/soundrecorder/activities/SettingsActivity.java	✓	✓
Rigid Alarm Manager	/app/src/main/java/com/danielkim/soundrecorder/activities/SettingsActivity.java	✓	✓
Unclosed Closable	/app/src/main/java/com/danielkim/soundrecorder/activities/SettingsActivity.java	✓	✗
	/app/src/main/java/com/danielkim/soundrecorder/MySharedPreferences.java	✓	✗
Leaking Inner Class	/app/src/main/java/com/danielkim/soundrecorder/model/MyData.java	✓	✓
	/app/src/main/java/com/danielkim/soundrecorder/model/Recordings.java	✓	✓

Table 9 Exact shared instances from SAND

Bad smells	Source code path	PA	[42]
Member Ignoring Method	/src/kw/com/sanad/BaseSwipeListViewListener.java	✓	✗
	/src/kw/com/sanad/adapters/CallLogAdapter.java	✓	✓
	/src/kw/com/sanad/adapters/HomeActivityAdapter.java	✓	✓
	/src/kw/com/sanad/adapters/LaunchAdapter.java	✓	✓
	/src/kw/com/sanad/adapters/ServiceAdapter.java	✓	✓
	/src/kw/com/sanad/adapters/ServicePagerAdapter.java	✓	✓
	/src/kw/com/sanad/AdvertiseFormActivity.java	✓	✓
	/src/kw/com/sanad/ActivityKhaMukha.java	✓	✗
	/src/kw/com/sanad/ChooseLanguageActivity.java	✓	✓
	/src/kw/com/sanad/helper/GPSTracker.java	✓	✓
	/src/kw/com/sanad/helper/Webservices.java	✓	✓
	/src/kw/com/sanad/Home.java	✓	✓
	/src/kw/com/sanad/HomeActivity.java	✓	✓
	/src/kw/com/sanad/LaunchActivity.java	✓	✓
	/src/kw/com/sanad/NotificationClickHandler.java	✓	✓
	/src/kw/com/sanad/NotificationPopup.java	✓	✓
	/src/kw/com/sanad/PopupWindows.java	✓	✓
	/src/kw/com/sanad/SANAD.java	✓	✗
	/src/kw/com/sanad/SANAD.java	✓	✗
	/src/kw/com/sanad/SettingsActivity.java	✓	✓
	/src/kw/com/sanad/SwipeListViewListener.java	✓	✓
	/src/kw/com/sanad/ActionItem	✗	✓
	/src/kw/com/sanad/BaseActivity	✗	✓
	/src/kw/com/sanad/CallsLogsActivity	✗	✓
	/src/kw/com/sanad/ChooseLanguageActivity	✗	✓
	/src/kw/com/sanad/GCMIntentService	✗	✓
	/src/kw/com/sanad/SanadApp	✗	✓
	/src/kw/com/sanad/helper/AppSP	✗	✓
	/src/kw/com/sanad/helper/HelperMethods	✗	✓
No Low Memory Resolver	/src/kw/com/sanad/BaseActivity.java	✓	✓
	/src/kw/com/sanad/AdvertiseFormActivity.java	✓	✗
	/src/kw/com/sanad/ActivityKhaMukha.java	✓	✓
	/src/kw/com/sanad/ChooseLanguageActivity.java	✓	✗
	/src/kw/com/sanad/Home.java	✓	✗
	/src/kw/com/sanad/HomeActivity.java	✓	✗
	/src/kw/com/sanad/LaunchActivity.java	✓	✗
	/src/kw/com/sanad/NotificationClickHandler.java	✓	✗
	/src/kw/com/sanad/NotificationPopup.java	✓	✗
	/src/kw/com/sanad/SANAD.java	✓	✗
	/src/kw/com/sanad/ServiceActivity.java	✓	✗
	/src/kw/com/sanad/helper/GPSTracker.java	✗	✓
	/src/kw/com/sanad/SanadApp	✗	✓
Internal Getter/Setter	/src/kw/com/sanad/helper/GPSTracker.java	✓	✓
	/src/kw/com/sanad/CallsLogsActivity	✗	✓
Inefficient Data Format and Parser	/src/kw/com/sanad/helper/Webservices.java	✓	✓
	/src/kw/com/sanad/Home.java	✓	✓
	/src/kw/com/sanad/ServiceActivity.java	✓	✓
Inefficient Data Structure	/src/kw/com/sanad/BaseActivity.java	✓	✗
	/src/kw/com/sanad/SettingsActivity.java	✓	✗

Table 9 (continued)

Bad smells	Source code path	PA	[42]
Leaking Thread	/src/kw/com/sanad/ServiceActivity.java	✗	✓
	/src/kw/com/sanad/helper/GPSTracker.java	✗	✓
	/src/kw/com/sanad/SANAD.java	✗	✓
	/src/kw/com/sanad/HomeActivity.java	✗	✓
	/src/kw/com/sanad/AdvertiseFormActivity.java	✗	✓
	/src/kw/com/sanad/BaseSwipeListViewListener.java	✗	✓
Slow For Loop	/src/kw/com/sanad/HomeActivity.java	✓	✓
	/src/kw/com/sanad/QuickAction.java	✓	✓
	/src/kw/com/sanad/ServiceActivity.java	✓	✓
	/src/kw/com/sanad/SettingsActivity.java	✓	✓
	/src/kw/com/sanad/SwipeListView.java	✓	✓
	/src/kw/com/sanad/SwipeListViewTouchListener.java	✓	✓
	/src/kw/com/sanad/CallsLogsActivity	✗	✓
Public Data	/src/kw/com/sanad/helper/AppSP	✗	✓
Durable WakeLock	/src/kw/com/sanad/AdvertiseFormActivity.java	✓	✓
Rigid Alarm Manager	/src/kw/com/sanad/AdvertiseFormActivity.java	✓	✓
Unclosed Closable	/src/kw/com/sanad/BaseActivity.java	✓	✗
	/src/kw/com/sanad/helper/AppSP.java	✓	✗
	/src/kw/com/sanad/SettingsActivity.java	✓	✗

cannot detect "Member Ignoring Method" bad smell from examined applications and we get false negative result. Similarly, we get false positive result while detecting same smell when a method does not directly access the field. In contrary, the method accesses a field indirectly using getter method or any other method holding the value of a field.

One of the key challenges for measuring the accuracy of our approach is the unavailability of standard benchmark systems on the results of selected 25 Android smells. We selected three open source and one industrial application for measuring the accuracy of our approach. The manual analysis of source code was conducted by both authors and two master students for calculation of false negatives. Similarly, false positives are determined through comparison of detected instances with manual analysis of source code. It was a very time-consuming and daunting task. The authors cross-checked results of master students to ensure reliability in results to the best level.

In Table 10, precision, recall and *F*-measure for four applications, namely Easy Sound Recorder, LeafPic, SAND and AmazeFileManager, are presented. We calculated precision, recall and *F*-Measure metrics for these four applications. There are a few false positives and false negatives as shown in Table 10. The average precision, recall and *F*-Measure for four applications are presented in Fig. 4 and Table 10.

6 Validity Threats

Validity is important for researchers and practitioners for empirically validating results of different approaches [145]. The generalization of results for different groups and datasets is the key requirement from researchers and practitioners. External validity focuses on evaluating the generalization of results. The approach presented in this paper mitigates external validity threats as we performed experiments on 4 Android open source applications of different types and sizes for the evaluation of our approach. Moreover, we also evaluated our approach on 3 industrial applications of moderate size for the generalization of our results. We performed experiments on a large number of Android code bad smells as compared to previous approaches that are restricted to only a few Android code bad smells. However, we cannot assert that our results may produce same accuracy for other large and complex Android applications. We also accept that there might be a chance of errors during manual evaluation of false positives and false negatives from source code. To minimize this threat, both authors carefully cross-validated results. Internal validity is used to evaluate the effect of independent variables on dependent variables. The lack of standard and formal definitions, Android source code metrics and unavailability of standard benchmark systems for results of Android code bad smell detection techniques is a major

Table 10 Precision, Recall and *F*-Measure for Easy sound recorder, LeafPic, SAND and AmazeFileManager applications

Bad smells/ projects	Easy sound recorder			LeafPic		
	Precision	Recall	*F*-Measure	Precision	Recall	*F*-Measure
MIM	13/(13 + 1) = 92.85%	13/(13 + 0) = 100%	96%	90/(90 + 6) = 93.75%	90/(90 + 10) = 90%	92%
NLMR	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%	10/(10 + 1) = 90.90%	10/(10 + 0) = 100%	95%
IGS	8/(8 + 0) = 100%	8/(8 + 0) = 100%	100%	9/(9 + 2) = 90%	9/(9 + 0) = 100%	92%
IDFP	N/A	N/A	N/A	N/A	N/A	N/A
IDS	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	3/(3 + 0) = 100%	3/(3 + 0) = 100%	100%
LT	1/(1 + 1) = 50%	2/(2 + 0) = 100%	67%	N/A	N/A	N/A
SL	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	41/(41 + 4) = 91.11%	41/(41 + 6) = 87.23%	89%
PD	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%
DW	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%
RAM	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%
UC	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%
LIC	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%
DR	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%
SV	3/(3 + 0) = 100%	3/(3 + 0) = 100%	100%	N/A	N/A	N/A
SC	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%	N/A	N/A	N/A
SB	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	N/A	N/A	N/A
CV	N/A	N/A	N/A	N/A	N/A	N/A
CB	N/A	N/A	N/A	N/A	N/A	N/A
DD	7/(7 + 1) = 87%	8/(8 + 1) = 88%	87%	17/(17 + 2) = 89.47%	17/(17 + 4) = 80.95%	86%
Unt	N/A	N/A	N/A	N/A	N/A	N/A
UFO	4/(4 + 1) = 80%	5/(5 + 0) = 100%	89%	39/(39 + 4) = 90.69%	39/(39 + 2) = 95.12%	93%
NL	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	24/(24 + 0) = 100%	24/(24 + 0) = 100%	100%
NDUI	5/(5 + 0) = 100%	5/(5 + 0) = 100%	100%	43/(43 + 2) = 95.55%	43/(43 + 3) = 93.47%	94%
SCC	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%	6/(6 + 1) = 85.71%	6/(6 + 0) = 100%	92.30%
OP	N/A	N/A	N/A	3/(3 + 0) = 100%	3/(3 + 0) = 100%	100%
Average	87.14%	99.4%	96.9%	95.47%	96.76%	96%
Bad smells/ projects	SAND			AmazeFileManager		
	Precision	Recall	*F*-Measure	Precision	Recall	*F*-Measure
MIM	21/(21 + 4) = 84%	21/(21 + 6) = 77.77%	80.76%	141/(141 + 17) = 89.24%	141/(141 + 11) = 92.76%	90.96%
NLMR	12/(12 + 0) = 100%	12/(12 + 0) = 100%	100%	8/(8 + 0) = 100%	8/(8 + 0) = 100%	100%
IGS	4/(4 + 2) = 66.66%	4/(4 + 3) = 57.14%	61.53%	21/(21 + 7) = 75%	21/(21 + 9) = 70%	72.41%
IDFP	3/(3 + 0) = 100%	3/(3 + 0) = 100%	100%	N/A	N/A	N/A
IDS	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%	4/(4 + 0) = 100%	4/(4 + 0) = 100%	100%
LT	N/A	N/A	N/A	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%
SL	26/(26 + 0) = 100%	26/(26 + 0) = 100%	100%	107/(107 + 0) = 100%	107/(107 + 0) = 100%	100%
PD	N/A	N/A	N/A	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%
DW	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	1/(1 + 1) = 50%	1/(1 + 0) = 100%	66.66%
RAM	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%	2/(2 + 0) = 100%	2/(2 + 0) = 100%	100%
UC	2/(2 + 1) = 66.66%	2/(2 + 0) = 100%	79.99%	3/(3 + 4) = 42%	3/(3 + 2) = 60%	49.41%
LIC	3/(3 + 0) = 100%	3/(3 + 0) = 100%	100%	3/(3 + 0) = 100%	3/(3 + 0) = 100%	100%
DR	N/A	N/A	N/A	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%
SV	4/(4 + 0) = 100%	4/(4 + 0) = 100%	100%	N/A	N/A	N/A
SC	3/(3 + 0) = 100%	3/(3 + 0) = 100%	100%	1/(1 + 0) = 100%	1/(1 + 0) = 100%	100%
SB	N/A	N/A	N/A	N/A	N/A	N/A
CV	2/(2 + 2) = 50%	2/(2 + 0) = 100%	66.66%	5/(5 + 1) = 83.33%	5/(5 + 0) = 100%	90.90%
CB	N/A	N/A	N/A	N/A	N/A	N/A
DD	9/(9 + 3) = 75%	9/(9 + 1) = 90%	81.81%	16/(16 + 4) = 80%	16/(16 + 3) = 84%	81.95%

Table 10 (continued)

Bad smells/ projects	SAND			AmazeFileManager		
	Precision	Recall	*F*-Measure	Precision	Recall	*F*-Measure
Unt	16(16 + 0) = 100%	16(16 + 0) = 100%	100%	15(15 + 0) = 100%	15(15 + 0) = 100%	100%
UFO	39/(39 + 17) = 69.64%	39/(39 + 11) = 78%	73.58%	143/(143 + 14) = 81.71%	143/(143 + 0) = 100%	89.93%
NL	16/(16 + 0) = 100%	16/(16 + 0) = 100%	100%	22/(22 + 0) = 100%	22/(22 + 0) = 100%	100%
NDUI	50/(50 + 1) = 100%	50/(50 + 0) = 100%	100%	162/(162 + 0) = 100%	162/(162 + 0) = 100%	100%
SCC	N/A	N/A	N/A	N/A	N/A	N/A
OP	38/(38 + 9) = 80.85%	38/(38 + 3) = 92.68%	86.36	3/(3 + 0) = 100%	3/(3 + 0) = 100%	100%
Average	89.09%	94.50%	91.08. %	90.06%	95.33%	92.11%

MIM Member Ignoring Method, *NLMR* No Low Memory Resolver, *IGS* Internal Getter/Setter, *IDFP* Inefficient Data Format and Parser, *IDS* Inefficient Data Structure, *LT* Leaking Thread, *SL* Slow Loop PD *DE RAM* Rigid Alarm Manager, *UC* Unclosed Closable, *LIC* Leaking Inner Class *DR* Debuggable Release, *SV* Static Views, *SC* Static Context, *SB* Static Bitmap, *CV* Collection of Views, *CB* Collection of Bitmaps *DD* Dropped Data *Unt* Untouchable, *UFO* Uncontrolled Focus Order, *NDUI* Not Descriptive UI, *NL* Nested Layout, *SCC* Set Config Changes, *OP* Overdrawn Pixel

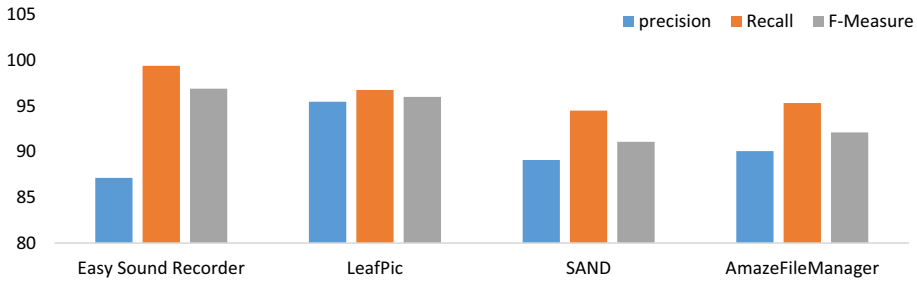

Fig. 4 Precision, Recall and *F*-Measure

internal validity concern. We mitigate that threat by providing customizable definitions, algorithms and threshold values of metrics for the detection of Android code bad smells. Second, we compare the results of our approach with one representative state-of-the-art approach on smell by smell basis for the validity of our results. However, our results for smells that are not detected by ADOCTOR may have internal validity threats. Reliability validity measures that a variable or set of variables are consistent in their values results of our study can be replicated. In order to mitigate that threat, we selected open source software applications and source code is available publically on the web for validation. We also present our prototyping tool, Android smell definitions, detection algorithms and results of the tool on the web which eliminates reliability threats. However, we admit that there might be reliability threats while checking our results for the calculation of recall in the case of large software projects.

7 Conclusion and Future Work

The mobile applications expect high reliability and performance from end users as compared to desktop and web applications. The detection of code smells from Android applications is beneficial for refactoring, maintenance and evolution of mobile applications. A very few approaches are presented in the state of the art for detection of Android-specific code bad smells from mobile applications. The existing approaches are limited only to few Android code bad smells and they have issues of accuracy and flexibility. We present a flexible approach by integrating the concepts of source code analysis and source code metrics. The presented approach is validated with tool support to recover 25 Android code bad smells from 7 mobile Android applications. We evaluate our approach by performing experiments on 7 Android mobile applications and compare the recovered results with a representative state-of-the-art approach. We publish our results on the web as the first benchmark for researchers. The precision, recall and *F*-measure are the accuracy measures used for the validation of results. Currently, our prototyping tool is limited to Java programming language but the proposed approach is flexible and extendable for other programming languages. Due to lose specification of bad smells and their threshold values, the detection results may vary. There is a need for a comprehensive and standard benchmark for specifications of Android-specific code smells. We have a plan to extend our prototyping tool for other mobile application platforms like iPhone. Other mobile application languages like C#, Swift, Kotlin will be added

in order to detect bad smells. We also plan to extend our approach for correction of these Android bad smells.

Acknowledgements The authors are thankful to anonymous reviewers for their valuable comments and suggestions on early version of this paper.

References

1. Lim, D.: Detecting code smells in Android applications. Master Thesis, TU Delft, Netherlands
2. Fowler, M.; Beck, K.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (1999)
3. Saborido, R.; Morales, R.; Khomh, F.; Guéhéneuc, Y.G.; Antoniol, G.: Getting the most from map data structures in Android. Empir. Softw. Eng. **23**, 2839–2864 (2018)
4. Li, L.; Bissyandé, T.F.; Papadakis, M.; Rasthofer, S.; Bartel, A.; Octeau, D.; Traon, L.: Static analysis of android apps: a systematic literature review. Inf. Softw. Technol. **88**, 67–95 (2017)
5. Reimann, J.; Brylski, M.; Aßmann, U.: A tool-supported quality smell catalogue for android developers. In: Proceedings of the Conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM (2014)
6. Payet, É.; Spoto, F.: Static analysis of Android programs. Inf. Softw. Technol. **54**(11), 1192–1201 (2012)
7. Trifu, A.; Marinescu, R.: Diagnosing design problems in object oriented systems. In: 12th Working Conference on Reverse Engineering, pp. 155–164 (2005)
8. Hassaine, S.; Khomh, F.; Guéhéneuc, Y.-G.; Hamel, S.: IDS: an immune-inspired approach for the detection of software design smells. In: 2010 Seventh International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 343–348 (2010)
9. Travassos, G.; Shull, F.; Fredericks, M.; Basili, V.R.: Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: ACM Sigplan Notices, pp. 47–56 (1999)
10. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: 20th IEEE International Conference on Software Maintenance. Proceedings, pp. 350–359 (2004)
11. Khomh, F.; Vaucher, S.; Guéhéneuc, Y.-G.; Sahraoui, H.: A bayesian approach for the detection of code and design smells. In: 9th International Conference on Quality Software. QSIC'09, pp. 305–314 (2009)
12. Stoianov, A.; Şora, I.: Detecting patterns and antipatterns in software using Prolog rules. In: 2010 International Joint Conference on Computational Cybernetics and Technical Informatics (ICCC-CONTI), pp. 253–258 (2010)
13. Khomh, F.; Vaucher, S.; Guéhéneuc, Y.-G.; Sahraoui, H.: BDTEX: a GQM-based Bayesian approach for the detection of antipatterns. J. Syst. Softw. **84**, 559–572 (2011)
14. Maiga, A.; Ali, N.; Bhattacharya, N.; Sabané, A.; Guéhéneuc, Y.-G.; Antoniol, G.: Support vector machines for anti-pattern detection. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 278–281 (2012)
15. Sjøberg, D.I.; Yamashita, A.; Anda, B.C.; Mockus, A.; Dybå, T.: Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. **39**, 1144–1156 (2013)
16. Kessentini, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Ouni, A.: A cooperative parallel search-based software engineering approach for code-smells detection. IEEE Trans. Softw. Eng. **40**, 841–861 (2014)
17. Sahin, D.; Kessentini, M.; Bechikh, S.; Deb, K.: Code-smell detection as a bilevel problem. ACM Trans. Softw. Eng. Methodol. (TOSEM) **24**, 6 (2014)
18. Ujhelyi, Z.; Horváth, Á.; Varró, D.; Csiszár, N.I.; Szoke, G.; Vidács, L.; et al.: Anti-pattern detection with model queries: a comparison of approaches. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), pp. 293–302 (2014)
19. Velioğlu, S.; Selçuk, Y.E.: An automated code smell and anti-pattern detection approach. In: 2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA), pp. 271–275 (2017)
20. Maiga, A.; Ali, N.; Bhattacharya, N.; Sabane, A.; Gueheneuc, Y.-G.; Aimeur, E.: SMURF: a SVM-based incremental anti-pattern detection approach. In: 2012 19th Working Conference on Reverse Engineering (WCRE), pp. 466–475 (2012)
21. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; De Lucia, A.; Poshyvanyk, D.: Detecting bad smells in source code using change history information. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 268–278 (2013)
22. Di Nucci, D.; Palomba, F.; Tamburri, D.A.; Serebrenik, A.; De Lucia, A.: Detecting code smells using machine learning techniques: are we there yet?. In: International Conference on Software Analysis, Evolution, and Reengineering. IEEE (2018)
23. Moha, N.; Gueheneuc, Y.-G.; Duchien, L.; Le Meur, A.-F.: DECOR: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**, 20–36 (2010)
24. Moha, N.; Gueheneuc, Y.-G.; Leduc, P.: Automatic generation of detection algorithms for design defects. In: 21st IEEE/ACM International Conference on Automated Software Engineering. ASE'06, pp. 297–300 (2006)
25. Fokaefs, M.; Tsantalis, N.; Chatzigeorgiou, A.: Jdeodorant: Identification and removal of feature envy bad smells. In: 2007 IEEE International Conference on Software Maintenance, pp. 519–520 (2007)
26. Rasool, G.; Arshad, Z.: A lightweight approach for detection of code smells. Arab. J. Sci. Eng. **42**(2), 483–506 (2017)
27. Fontana, F.A.; Mäntylä, M.V.; Zanoni, M.; Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. **21**(3), 1143–1191 (2016)
28. Mansoor, U.; Kessentini, M.; Maxim, B.R.; Deb, K.: Multi-objective code-smells detection using good and bad design examples. Softw. Qual. J. **25**(2), 529–552 (2017)
29. Guggulothu, T.: Code smell detection using multilabel classification approach (2019). arXiv preprint arXiv:1902.03222
30. Hadj-Kacem, M.; Bouassida, N.: A hybrid approach to detect code smells using deep learning. In: ENASE, pp. 137–146 (2018)
31. Li, Z.; Chen, T.H.P.; Yang, J.; Shang, W.: Dlfinder: characterizing and detecting duplicate logging code smells. In: Proceedings of the 41st International Conference on Software Engineering, pp. 152–163 (2019)
32. Blouin, A.; Lelli, V.; Baudry, B.; Coulon, F.: User interface design smell: automatic detection and refactoring of Blob listeners. Inf. Softw. Technol. **102**, 49–64 (2018)
33. Verloop, D.: Code smells in the mobile applications domain. Master Thesis, TUDelft, Neitherland (2013)
34. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A.: Mining version histories for detecting code smells. IEEE Trans. Softw. Eng. **41**(5), 462–489 (2014)
35. Fontana, F.A.; Dietrich, J.; Walter, B.; Yamashita, A.; Zanoni, M.: Antipattern and code smell false positives: preliminary conceptualization and classification. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 609–613 (2016)

36. Palomba, F.; Panichella, A.; De Lucia, A.; Oliveto, R.; Zaidman, A.: A textual-based technique for smell detection. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp. 1–10 (2016)
37. Di Nucci, D.; Palomba, F.; Tamburri, D.A.; Serebrenik, A.; De Lucia, A.: Detecting code smells using machine learning techniques: are we there yet?. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 612–621 (2018)
38. Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., Zhang, L.: Deep learning based code smell detection. IEEE Trans. Softw. Eng. (2019). https://doi.org/10.1109/TSE.2019.2936376
39. Banerjee, A.; Chong, L.K.; Chattopadhyay, S.; Roychoudhury, A.: Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 588–598 (2014)
40. Hecht, G.: An approach to detect Android antipatterns. In: Proceedings of the 37th International Conference on Software Engineering, vol. 2, pp. 766–768 (2015)
41. Hecht, G.; Rouvoy, R.; Moha, N.; Duchien, L.: Detecting antipatterns in android apps. In: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems, pp. 148–149 (2015)
42. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A.: Lightweight detection of Android-specific code smells: the aDoctor project. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 487–491 (2017)
43. Rasool, G.; Arshad, Z.: A review of code smell mining techniques. J. Softw. Evol. Process **27**(11), 867–895 (2017)
44. Carette, A.; Younes, M.A.A.; Hecht, G.; Moha, N.; Rouvoy, R.: Investigating the energy impact of android smells. In: 24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER), p. 10 (2017)
45. Grano, G.; Di Sorbo, A.; Mercaldo, F.; Visaggio, C.A.; Canfora, G.; Panichella, S.: Android apps and user feedback: a dataset for software evolution and quality improvement. In: Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics, pp. 8–11 (2017)
46. Mateus, B.G.; Martinez, M.: An empirical study on quality of Android applications written in Kotlin language (2018). arXiv preprint arXiv:1808.00025
47. Peruma, A.S.A.: What the smell? An empirical investigation on the distribution and severity of test smells in open source Android applications. Master Thesis, Rochester Institute of Technology, Rochester, New York (2018)
48. Elsayed, E.K.; ElDahshan, K.A.; El-Sharawy, E.E.; Ghannam, N.E.: Reverse engineering approach for improving the quality of mobile applications. PeerJ Preprints **7**, e27633v1 (2019)
49. Almalki, K.S.: Bad Droid! An in-depth empirical study on the occurrence and impact of Android specific code smells. Thesis. Rochester Institute of Technology (2018)
50. El-Dahshan, K.A.; Elsayed, E.K.; Ghannam, N.E.: Comparative study for detecting mobile application's anti-patterns. In: Proceedings of the 2019 8th International Conference on Software and Information Engineering, pp. 1–8 (2019)
51. Habchi, S.; Hecht, G.; Rouvoy, R.; Moha, N.: Code smells in iOS apps: how do they compare to Android?. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, pp. 110–121 (2017)
52. Hecht, G.; Benomar, O.; Rouvoy, R.; Moha, N.; Duchien, L.: Tracking the software quality of android applications along their evolution (t). In: Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 236–247 (2015)
53. Mannan, U.A.; Ahmed, I.; Almurshed, R.A.M.; Dig, D.; Jensen, C.: Understanding code smells in android applications. In: Proceedings of the International Workshop on Mobile Software Engineering and Systems, pp. 225–234. ACM (2016)
54. Ghafari, M.; Gadient, P.; Nierstrasz, O.: Security smells in Android. In: Proceedings of IEEE 17th International Working Conference on Source Code Analysis and Manipulation, pp. 121–130 (2017)
55. Cruz, L.; Abreu, R.: Using automatic refactoring to improve energy efficiency of Android apps (2018). arXiv preprint arXiv:1803.05889
56. Abbes, M.; Khomh, F.; Guéhéneuc, Y.-G.; Antoniol, G.: An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Proceedings of the 15th European Conference on Software Maintenance and Reengineering, Oldenburg, Germany. IEEE Computer Society, pp. 181–190 (2011)
57. Oliveira, J.; Viggiato, M.; Santos, M.; Figueiredo, E.; Marques-Neto, H. An empirical study on the impact of android code smells on resource usage. In: International Conference on Software Engineering & Knowledge Engineering (SEKE) (2018)
58. Verdecchia, R.; Saez, R.A.; Procaccianti, G.; Lago, P.: Empirical evaluation of the energy impact of refactoring code smells. In: 5th International Conference on ICT for Sustainability, pp. 365–383 (2018)
59. Khomh, F.; Di Penta, M.; Guéhéneuc, Y.-G.; Antoniol, G.: An exploratory study of the impact of antipatterns on class changeand fault-proneness. Empir. Softw. Eng. **17**(3), 243–275 (2012)
60. Das, T.; Di Penta, M.; Malavolta, I.: A quantitative and qualitative investigation of performance-related commits in Android apps. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 443–447 (2016)
61. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D.: When and why your code starts to smell bad. In: Proceedings of the 37th International Conference on Software Engineering, vol. 1, pp. 403–414 (2015)
62. Palomba, F.; Bavota, G.; Di Penta, M.; Fasano, F.; Oliveto, R.; De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir. Softw. Eng. **23**(3), 1188–1221 (2018)
63. Gadient, P.; Ghafari, M.; Frischknecht, P.; Nierstrasz, O.: Security code smells in Android ICC. Empir. Softw. Eng. **24**, 3046–3076 (2018)
64. Olbrich, S.; Cruzes, D.S.; Basili, V.; Zazworka, N.: The evolution and impact of code smells: a case study of two open source systems. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ser. ESEM'09, pp. 390–400 (2009)
65. Cairo, A.; Carneiro, G.; Monteiro, M.: The impact of code smells on software bugs: a systematic literature review. Information **9**(11), 273 (2018)
66. Yamashita, A.; Counsell, S.: Code smells as system-level indicators of maintainability: an empirical study. J. Syst. Softw. **86**(10), 2639–2653 (2013)
67. Perez-Castillo, R.; Piattini, M.: Analyzing the harmful effect of god class refactoring on power consumption. IEEE Softw. **31**(3), 48–54 (2014)
68. Gottschalk, M.; Jelschen, J.; Winter, A.: Saving energy on mobile devices by refactoring. In: EnviroInfo. enviroinfo.eu, pp. 437–444 (2014)
69. Kim, D.K.: Towards performance-enhancing programming for Android application development. Int. J. Contents **13**(4), 39–46 (2017)
70. Rodriguez, A.; Longo, M.; Zunino, A.: Using bad smell-driven code refactorings in mobile applications to reduce battery usage.

In: Simposio Argentino de Ingeniería de Software (ASSE 2015)-JAIIO 44 (Rosario, 2015) (2015).
71. Fan, L.; Su, T.; Chen, S.; Meng, G.; Liu, Y.; Xu, L., Su, Z. Large-scale analysis of framework-specific exceptions in Android apps. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 408–419 (2018)
72. Habchi, S.; Rouvoy, R.; Moha, N.: On the survival of Android code smells in the wild. In: 6th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft) (2019)
73. Habchi, S.; Moha, N.; Rouvoy, R.: The rise of Android code smells: Who is to blame?. In: MSR 2019-Proceedings of the 16th International Conference on Mining Software Repositories (2019)
74. Malavolta, I.; Verdecchia, R.; Filipovic, B.; Bruntink, M.; Lago, P.:. How maintainability issues of Android apps evolve. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 334–344 (2018)
75. Pritam, N.; Khari, M.; Kumar, R.; Jha, S.; Priyadarshini, I.; Abdel-Basset, M.; Long, H.V.: Assessment of code smell for predicting class change proneness using machine learning. IEEE Access **7**, 37414–37425 (2019)
76. Johannes, D.; Khomh, F.; Antoniol, G.: A large-scale empirical study of code smells in JavaScript projects. Softw. Qual. J. **27**, 1271–1314 (2019)
77. Rodriguez, A.; Mateos, C.; Zunino, A.: Improving scientific application execution on android mobile devices via code refactorings. Softw. Pract. Exp. **47**(5), 763–796 (2017)
78. Morales, R.; Saborido, R.; Khomh, F.; Chicano, F.; Antoniol, G.: Anti-patterns and the energy efficiency of Android applications. IEEE Trans. Softw. Eng. (2016)
79. Kim, D.; Hong, J.E.; Yoon, I.; Lee, S.H.: Code refactoring techniques for reducing energy consumption in embedded computing environment. Cluster Comput. **21**(1), 1079–1095 (2018)
80. Li, W.; Jiang, Y.; Xu, C.; Liu, Y.; Ma, X.; Lü, J.: Characterizing and detecting inefficient image displaying issues in Android apps. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 355–365 (2019)
81. Cruz, L., Abreu, R.: Performance-based guidelines for energy efficient mobile applications. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 46–57 (2017)
82. Linares-Vásquez, M.: Supporting evolution and maintenance of Android apps. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 714–717 (2014)
83. Li, X.; Gallagher, J.P.: A source-level energy optimization framework for mobile applications. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 31–40 (2016)
84. Saifan, A.A.; Al-Rabadi, A.: Evaluating maintainability of android applications. In: 2017 8th International Conference on Information Technology (ICIT), pp. 518–523 (2017)
85. Afjehei, S.S.; Chen, T.H.P.; Tsantalis, N.: iPerfDetector: characterizing and detecting performance anti-patterns in iOS applications. Empir. Softw. Eng. 1–30 (2019)
86. Mumtaz, H.; Alshayeb, M.; Mahmood, S.; Niazi, M.: An empirical study to improve software security through the application of code refactoring. Inf. Softw. Technol. **96**, 112–125 (2018)
87. Linares-Vásquez, M.; Vendome, C.; Tufano, M.; Poshyvanyk, D.: How developers micro-optimize android apps. J. Syst. Softw. **130**, 1–23 (2017)
88. Gottschalk, M.; Josefiok, M.; Jelschen, J.; Winter, A.: Removing energy code smells with reengineering services. INFORMATIK (2012)
89. Meier, J.; Ostendorp, M.C.; Jelschen, J.; Winter, A.: Certifying energy efficiency of android applications. In: EnviroInfo, pp. 765–770 (2014)
90. Rani, A.; Chhabra, J.K.: Evolution of code smells over multiple versions of softwares: an empirical investigation. In: 2017 2nd International Conference for Convergence in Technology (I2CT), pp. 1093–1098 (2017)
91. Chatzigeorgiou, A.; Manakos, A.: Investigating the evolution of bad smells in object-oriented code. In: 2010 Seventh International Conference on the Quality of Information and Communications Technology, pp. 106–115 (2010)
92. Soh, Z.; Yamashita, A.; Khomh, F.; Guéhéneuc, Y.G.: Do code smells impact the effort of different maintenance programming activities?. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 393–402 (2016)
93. Tahmid, A.; Nahar, N.; Sakib, K.: Understanding the evolution of code smells by observing code smell clusters. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 4, pp. 8–11 (2016)
94. Li, D.; Halfond, W.G.: An investigation into energy-saving programming practices for android smartphone app development. In: Proceedings of the 3rd International Workshop on Green and Sustainable Software, pp. 46–53 (2014)
95. Gottschalk, M.; Jelschen, J.; Winter, A.: Refactorings for energy-efficiency. In: Advances and new trends in environmental and energy informatics, pp. 77–96. Springer, Cham (2016)
96. Habchi, S.; Blanc, X.; Rouvoy, R.: On adopting linters to deal with performance concerns in Android apps. In: Proceedings of Automated Software Engineering, pp. 6–16 (2018)
97. Chatzigeorgiou, A.; Manakos, A.: Investigating the evolution of code smells in object-oriented systems. Innov. Syst. Softw. Eng. **10**(1), 3–18 (2014)
98. Vidal, S.A.; Marcos, C.; Díaz-Pace, J.A.: An approach to prioritize code smells for refactoring. Autom. Softw. Eng. **23**(3), 501–532 (2016)
99. Sae-Lim, N.; Hayashi, S.; Saeki, M.: How do developers select and prioritize code smells? A preliminary study. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 484–488 (2017)
100. Palomba, F.; Tamburri, D.A.A.; Fontana, F.A.; Oliveto, R.; Zaidman, A.; Serebrenik, A.: Beyond technical aspects: how do community smells influence the intensity of code smells?. IEEE Trans. Softw. Eng. (2018)
101. Hozano, M.; Antunes, N.; Fonseca, B.; Costa, E.: Evaluating the Accuracy of machine learning algorithms on detecting code smells for different developers. In: ICEIS (2), pp. 474–482 (2017)
102. Liu, H.; Li, B.; Yang, Y.; Ma, W.; Jia, R.: Exploring the impact of code smells on fine-grained structural change-proneness. Int. J. Softw. Eng. Knowl. Eng. **28**(10), 1487–1516 (2018)
103. Palomba, F.; Zanoni, M.; Fontana, F.A.; De Lucia, A.; Oliveto, R.: Toward a smell-aware bug prediction model. IEEE Trans. Softw. Eng. **45**(2), 194–218 (2017)
104. Software Engineering Group: https://lahore.comsats.edu.pk/research/groups/SERC/Android-Bad-Smells.aspx
105. Haque, M.S.; Carver, J.; Atkison, T.: Causes, impacts, and detection approaches of code smell: a survey. In: Proceedings of the ACMSE 2018 Conference, p. 25 (2018)
106. Zhang, M.; Hall, T.; Baddoo, N.: Code bad smells: a review of current knowledge. J. Softw. Maint. Evol. Res. Pract. **23**(3), 179–202 (2011)
107. Gupta, A.; Suri, B.; Misra, S.: A systematic literature review: code bad smells in Java source code. In: International

Conference on Computational Science and Its Applications, pp. 665–682 (2017)
108. Kaur, A.; Dhiman, G.: A review on search-based tools and techniques to identify bad code smells in object-oriented systems. In: Harmony Search and Nature Inspired Optimization Algorithms, pp. 909–921 (2019).
109. Sharma, T.; Spinellis, D.: A survey on software smells. J. Syst. Softw. **138**, 158–173 (2018)
110. Azeem, M.I.; Palomba, F.; Shi, L.; Wang, Q.: Machine learning techniques for code smell detection: a systematic literature review and meta-analysis. Information and Software Technology (2019)
111. Azadi, U.; Fontana, F.A.; Zanoni, M.: Poster: machine learning based code smell detection through WekaNose. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 288–289 (2018)
112. Caram, F.L.; Rodrigues, B.R.D.O.; Campanelli, A.S.; Parreiras, F.S.: Machine learning techniques for code smells detection: a systematic mapping study. Int. J. Softw. Eng. Knowl. Eng. **29**(02), 285–316 (2019)
113. Alkharabsheh, K.; Crespo, Y.; Manso, E.; Taboada, J.A.: Software design smell detection: a systematic mapping study. Softw. Qual. J. **27**, 1069–1148 (2018)
114. de Paulo Sobrinho, E.V.; De Lucia, A.; de Almeida Maia, M.: A systematic literature review on bad smells—5 W's: which, when, what, who, where. IEEE Trans. Softw. Eng. 1–58 (2018)
115. Barbez, A.; Khomh, F.; Guéhéneuc, Y.G.: A machine-learning based ensemble method for anti-patterns detection (2019). arXiv preprint arXiv:1903.01899.
116. Saranya, G.; Nehemiah, H.K.; Kannan, A.; Nithya, V.: Model level code smell detection using egapso based on similarity measures. Alexandria Eng. J. **57**(3), 1631–1642 (2018)
117. Fernandes, E.; Oliveira, J.; Vale, G.; Paiva, T.; Figueiredo, E.: A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, p. 18. ACM (2016)
118. Din, J.; Al-Badareen, A.B.; Jusoh, Y.Y.: Antipatterns detection approaches in object-oriented design: a literature review. In: 2012 7th International Conference on Computing and Convergence Technology (ICCCT), pp. 926–931 (2012)
119. Kitchenham, B.; Pearl Brereton, O.; Budgen, D.; Turner, M.; Bailey, J.; Linkman, S.: Systematic literature reviews in software engineering–a systematic literature review. Inf. Softw. Technol. **51**(1), 7–15 (2009)
120. Ibrahim, R.; Ahmed, M.; Nayak, R.; Jamel, S.: Reducing redundancy of test cases generation using code smell detection and refactoring. J. King Saud Univ. Comput. Inf. Sci. (2018, in press)
121. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A.: On the impact of code smells on the energy consumption of mobile applications. Inf. Softw. Technol. **105**, 43–55 (2019)
122. Singh, S., & Kaur, S. (2017). A systematic literature review: Refactoring for disclosing code smells in object oriented software. Ain Shams Engineering Journal.
123. Kannangara, S.H.; Wijayanayake, W.M.J.I.: An empirical evaluation of impact of refactoring on internal and external measures of code quality (2015). arXiv preprint arXiv:1502.03526
124. Cedrim, D.; Sousa, L.; Garcia, A.; Gheyi, R.: Does refactoring improve software structural quality? A longitudinal study of 25 projects. In: Proceedings of the 30th Brazilian Symposium on Software Engineering, pp. 73–82 (2016)
125. Aniche, M.; Bavota, G.; Treude, C.; Gerosa, M.A.; van Deursen, A.: Code smells for model-view-controller architectures. Empir. Softw. Eng. **23**(4), 2121–2157 (2018)
126. Saboury, A.; Musavi, P.; Khomh, F., Antoniol, G.: An empirical study of code smells in javascript projects. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 294–305 (2017)
127. Prabowo, G.; Suryotrisongko, H.; Tjahyanto, A.: A tale of two development approach: empirical study on the maintainability and modularity of Android mobile application with anti-pattern and model-view-presenter design pattern. In: 2018 International Conference on Electrical Engineering and Informatics (ICELTICs) (44501), pp. 149–154 (2018)
128. Santos, J.A.M.; Rocha-Junior, J.B.; Prates, L.C.L.; do Nascimento, R.S.; Freitas, M.F.; de Mendonca, M.G.: A systematic review on the code smell effect. J. Syst. Softw. **144**, 450–477 (2018)
129. Lin, Y.; Okur, S.; Dig, D.: Study and refactoring of android asynchronous programming (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 224–235 (2015)
130. Hecht, G.; Moha, N.; Rouvoy, R.: An empirical study of the performance impacts of android code smells. In: Proceedings of the International Conference on Mobile Software Engineering and Systems, pp. 59–69 (2016)
131. Habchi, S.; Hecht, G.; Rouvoy, R.; Moha, N.: Code smells in iOS Apps: How do they compare to Android?. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 110–121 (2017)
132. Morales, R.; Saborido, R.; Khomh, F.; Chicano, F.; Antoniol, G.: Earmo: an energy-aware refactoring approach for mobile apps. IEEE Trans. Softw. Eng. **44**, 1176–1206 (2018)
133. Kessentini, M.; Ouni, A.: Detecting Android smells using multi-objective genetic programming. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, pp. 122–132 (2017)
134. Dennis, C.; Krutz, D.E.; Mkaouer, M.W.: P-lint: a permission smell detector for android applications. In: Proceedings of Mobile Software Engineering and Systems (MOBILESoft), pp. 219–220 (2017)
135. Paternò, F.; Schiavone, A.G.; Conti, A.: Customizable automatic detection of bad usability smells in mobile accessed web applications. In: Proceedings of the 19th International Conference on Human–Computer Interaction with Mobile Devices and Services, p. 42 (2017)
136. Rubin, J.; Henniche, A.N.; Moha, N.; Bouguessa, M.; Bousbia, N.: Sniffing Android code smells: an association rules mining-based approach. In: Proceedings of the 6th International Conference on Mobile Software Engineering and Systems, pp. 123–127 (2019)
137. Kessentini, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Ouni, A.: A cooperative parallel search-based software engineering approach for code-smells detection. IEEE Trans. Softw. Eng. **40**(9), 841–861 (2014)
138. Nuñez-Varela, A.S.; Pérez-Gonzalez, H.G.; Martínez-Perez, F.E.; Soubervielle-Montalvo, C.: Source code metrics: a systematic mapping study. J. Syst. Softw. **128**, 164–197 (2017)
139. Miceli, D.; et al.: Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In: Intl. Conf. on Software Maintenance. IEEE (2000)
140. Elish, M.O.; Rine, D.: Investigation of metrics for object-oriented design logical stability. In: Seventh European Conference on Software Maintenance and Reengineering. Proceedings. IEEE (2003)
141. Kaur, A.; Kaur, K.; Kaur, H.: An investigation of the accuracy of code and process metrics for defect prediction of mobile applications. In: 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), pp. 1–6 (2015)

142. Satrijandi, N.; Widyani, Y.: Efficiency measurement of java android code. In: 2014 International Conference on Data and Software Engineering (ICODSE), pp. 1–6 (2014)
143. Jošt, G.; Huber, J.; Heričko, M.: Using object oriented software metrics for mobile application development. In: 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications, pp. 17–27 (2013)
144. Kim, D.K.: Finding bad code smells with neural network models. Int. J. Electr. Comput. Eng. (2088-8708), **7**(6) (2017).
145. Mercaldo, F.; Di Sorbo, A.; Visaggio, C.A.; Cimitile, A.; Martinelli, F.: An exploratory study on the evolution of Android malware quality. J. Softw. Evol. Process **30**, e1978 (2018)
146. Rahman, A.; Pradhan, P.; Partho, A.; Williams, L.: Predicting Android application security and privacy risk with static code metrics. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 149–153 (2017)
147. Campbell, G.A.; Papapetrou, P.P.: SonarQube in Action, 1st edn. Manning Publications Co., Greenwich (2013)
148. Stojkovski, M.: Thresholds for software quality metrics in open source Android projects. Master Thesis, Norwegian University of Science and Technology (2017)
149. Bruggen. D. v.: Java Parser. http://javaparser.org/ (2017)