IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 6 March 2024, accepted 22 April 2024, date of publication 6 May 2024, date of current version 10 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3397055

APPLIED RESEARCH

Emerging Trends in Code Quality: Introducing Kotlin-Specific Bad Smell Detection Tool for Android Apps

RADINAL DWIKI NOVENDRA AND WIKAN DANAR SUNINDYO, (Member, IEEE)
School of Electrical Engineering and Informatics, Bandung Institute of Technology, Bandung, West Java 40132, Indonesia

Corresponding author: Radinal Dwiki Novendra (radinal.dwiki.novendra@gmail.com)

This work was supported by the School of Electrical Engineering and Informatics, Institut Teknologi Bandung, and the Indonesia Endowment Fund For Education (LPDP).

ABSTRACT The increasing demand for Android applications in line with technological evolution and the development of new features often leads to frequent updates and releases of applications. However, in this update cycle, developers sometimes make hasty changes that result in poor design choices, leading to the emergence of bad smell code. The impact of smells involves a decrease in quality, performance, and understanding, and impedes the software maintenance process. Some previously developed smell detection tools rely on JavaParser to parse source code into an Abstract Syntax Tree representation, meaning the extracted information is limited to Java programming language source code. Meanwhile, Google recommends Kotlin as the language for developing Android applications, and over 60% of professional Android developers use Kotlin. This research is using Program Structure Interface representation and applies a software metric-based approach as a method of smell detection and implements it into a Kotlin Android bad smell code detection tool. This approach was chosen because the characteristics of each smell can be represented with software metrics. The tool's evaluation results on five Kotlin Android projects showed an F-measure of 100% for Brain Class detection, 93.77% for God Class, and 85.71% for Brain Method. Comparing the tool's detection results with the iPlasma tool on the Quran project showed that the developed tool detected more smell occurrences. Usability testing results indicated that seven participants strongly agreed with the easy-to-understand detection results, easy-to-find presented functionality, clearly presented information, and the tool helped to perform metric calculation and smell detection more quickly and efficiently. The average rating given for each question was between 4.29 – 4.71.

INDEX TERMS Code smells, android, Kotlin.

I. INTRODUCTION

In recent years, mobile apps have dominated the software market, with 2.6 million Android apps on Google Play Store and 110 billion downloads in 2022 [1]. This demand necessitates high-quality app development. However, Android app updates are frequent [2], and developers may disregard standard practices, resulting in poor-quality code, known as 'bad smell' code [3].

'Bad smell' code reflects flaws in code structure and violations of design principles, significantly impacting software quality [4]. Previous studies have investigated the effects of smells on software, revealing that they not only reduce code quality, performance, and comprehensibility but also lead to increased complexity in software maintenance and a higher risk of bugs [5], [6], [7].

Manual detection of 'bad smell' code is arduous, time-consuming, and susceptible to errors, rendering it ineffective for large systems [8]. While tools like Paprika, aDoctor, and Android Bad Smell Detector [3], [9], [10] exist for Android 'bad smell' detection, they rely heavily on JavaParser, meaning they can only analyze Java-based code. Despite Google recommending Kotlin for Android app development in 2017 and its adoption by over 60% of professional

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

Android developers [11], there's a lack of Kotlin-supporting tools. By utilizing the Program Structure Interface (PSI) representation in Kotlin, which bears similarities to the Abstract Syntax Tree (AST) in general, it becomes highly feasible to perform smell detection on Kotlin programs.

The contributions of this paper are as follows. Firstly, we utilize the Program Structure Interface (PSI) representation employed by the smell detection algorithm, and then endeavor to implement it into a tool usable by practitioners. Secondly, we conduct experiments to measure the accuracy of the developed tool, comparing its performance with other tools, and conducting a survey with users regarding usability testing.

This paper is structured as follows. Section II provides background information on bad smell code and software metrics. Section III discusses related work. Section IV presents the method we propose for detecting bad smells and the experimental design. The experimental results are presented in Section V. Finally, we conclude this paper in Section VI.

II. BACKGROUND

This section explains the code smells that have been explored and investigated further in this paper. It also discusses software metrics, and the detection strategies typically used in identifying smells in object-oriented programming.

A. BAD SMELL CODE

Bad smell code is an indication of subpar code quality, which can potentially inconvenience developers during maintenance [12], the term ''smell'' was first introduced by Fowler in 1999 [4]. A smell can affect classes and methods; the characteristics of these smells indicate a negative impact on the quality of design elements discovered when considering design elements separately. In simple terms, these smells can be recognized by observing excessive size and complexity in classes and methods [13].

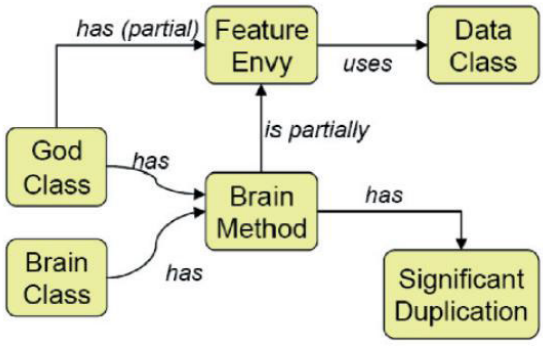


FIGURE 1. The relevance of bad smell [13].

Bad smells are interconnected, as shown in Figure 1. For instance, a Brain Class contains a Brain Method within it, and a God Class similarly possesses a Brain Method. The present study focuses explicitly on these three smells.

- **God Class** does too much work on its own, delegating only minor details to regular classes and using data from other classes. Naturally, this has a negative impact on system usability and comprehension [13].
- **Brain Method** differs from a God Class; this smell centralizes the functionality of its owning class. A good method ideally should have a level of complexity appropriate to its specific purpose [13].
- **Brain Class** differs from a Brain Method, with the smell appearing in the class. However, it's crucial to note that a Brain Class also differs from a God Class. At its simplest, this smell emerges in a class that isn't identified as a God Class, yet the class contains at least one Brain Method within it.

B. SOFTWARE METRICS

The most common approach to detecting software smells is using **software metrics**. This approach involves measuring various aspects of the software to obtain specific information, such as predicting defects in the software [14]. Examples of metrics in software include LOC, CYCLO, WMC, TCC, ATFD, MAXNESTING, and NOAV.

- **LOC (Lines of Code)** indicates the total number of lines defined in operations. Only lines of code that contain functionality, such as method lines of code, are counted [13].
- **CYCLO (Cyclomatic Number)** represents the potential number of program paths summed up from all operations within the system. This metric is the total of the McCabe Cyclomatic numbers for all operations and is used to measure the intrinsic functional complexity of the system [15].
- **WMC (Weight Method per Class)** indicates the number of methods implemented in a class or denotes the complexity value of methods implemented in a class, where the complexity of these methods is calculated using Cyclomatic Complexity [13], [16].

$$WMC = \sum_{i=1}^{n} ci$$

If the complexity of all methods is considered as a unit, then WMC = n, which is the number of methods.

- **TCC (Tight Class Cohesion)** indicates the number of methods directly connected through attribute access. TCC measures the cohesion among public methods in a class, expressing the ratio between directly connected methods and the maximum possible connections among methods in a class.

$$TCC(C) = \frac{NDC\ (C)}{NP(C)}$$

ND(C) represents the number of direct connections, while NP(C) denotes the maximum possible amount of connections among methods, both directly and indirectly, in a class. NP is obtained using the following formula, where N is the number of methods [17].

$$NP = \frac{N * (N-1)}{2}$$

Algorithm 1 Brain Method

```
1. FUNCTION isDetected(
   loc: integer,
   cyclo: integer,
   maxNest: integer,
   noav: integer
 ): boolean
2.    SET largeloc = loc > loc_th / 2.0
3.    SET manyBranches = cyclo / loc >= cyclo_loc_th
4.    SET deepNesting = maxNest >= max_nest_th
5.    SET manyVars = noav > noav_th
6.       RETURN largeloc AND manyBranches AND
      deepNesting AND manyVars
7. END FUNCTION
```

- **ATFD (Access to Foreign Data)** indicates the number of properties or attributes from other classes accessed either directly or through accessor methods by the class being measured. A higher ATFD typically corresponds to lower cohesion and higher coupling of the given class, signaling that a lower ATFD value is generally better [13]
- **MAXNESTING (Maximum Nesting)** represents the maximum level of nesting blocks or control structures present within a method or function. Generally, code blocks containing statements like if, else, else if, do, while, for, switch, catch, and others are part of nested loops [13].
- **NOAV (Number of Accessed Variables)** represents the count of variables accessed directly within a method or function. The term ''variables'' refers to all types of variables, including parameters, local variables, instance or object variables, and global variables [13].

C. DETECTION STRATEGY

Metrics are only capable of revealing somewhat obscure symptoms, akin to describing the symptoms of a disease without providing a deep understanding of the root issue. Marinescu proposed a detection strategy that involves the formulation of metric-based rules to capture design issues. This detection strategy utilizes threshold values to identify the presence of smell problems [18].

- **Brain Method Detection Strategy** is executed by calculating metrics like LOC, CYCLO, MAXNESTING, and NOAV on each method, this strategy can be observed through Algorithm 1 [13].
- **God Class Detection Strategy** is executed by calculating metrics like ATFD, WMC, and TCC for each class, this strategy can be observed through Algorithm 2 [13].
- **Brain Class Detection Strategy** is executed by calculating metrics such as Brain Method, LOC, WMC, and TCC for each class, this strategy can be viewed through Algorithm 3 [13].

D. PROGRAM STRUCTURE INTERFACE

The PSI (Program Structure Interface) can be utilized to view the structure as a representation of the contents of a file as a hierarchy of elements in a specific programming language. In this research, the PSI will be leveraged to extract metrics. Figure 2 displays how a code is represented in a PSI tree.

As shown in Figure 2, the PSI tree is a concrete syntax tree (CST) because it contains whitespace, and punctuation, and can be used to infer some semantic information from the code. With PSI, developers can perform several tasks such as traversing elements (for example, methods, comments,

Algorithm 2 God Class

```
1. FUNCTION isDetected(
   atfd: Integer,
   wmc: Integer,
   tcc: Double): Boolean
2.    SET usesMoreThanFew = atfd > thresholdATFD
3.    SET hasVeryHighComplexity =     wmc >= thresh-
oldWMC
4.    SET cohesionIsLow = tcc < thresholdTCC
5.    RETURN usesMoreThanFew
         AND hasVeryHighComplexity
         AND cohesionIsLow
6. END FUNCTION
```

Algorithm 3 Brain Class

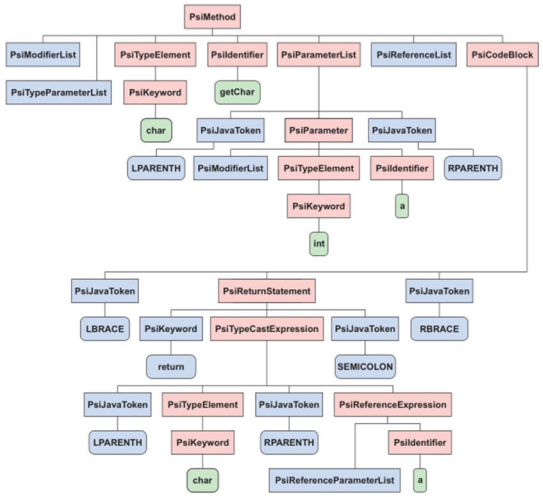
```
1. FUNCTION isDetected(
   bm: Integer,
   loc: Integer,
   wmc: Integer,
   cohesion: Double): Boolean
2.    moreThanOneBrain = bm > 1
3.    totalSizeIsHigh = loc >= tLOC
4.                              moreThanOneBrainAndHigh-
Size =          moreThanOneBrain AND totalSizeIsHigh
5.
6.    onlyOneBrain = bm == 1
7.    totalSizeIsVeryHigh = loc >= 2 * tLOC
8.    highComplexity = wmc >= 2 * tWMC
9.     oneBrainButVeryHighComplexity =          only-
OneBrain AND totalSizeIsVeryHigh AND
         highComplexity
10.
11.    veryHighComplexity = wmc >= tWMC
12.    lowCohesion = cohesion < tTCC
13.     complexAndNonCohesive =          veryHigh-
Complexity AND lowCohesion
14.
15.    isBrainClass =          (moreThanOneBrainAnd-
HighSize OR
          oneBrainButVeryHighComplexity) AND
          complexAndNonCohesive
16.
17.    RETURN isBrainClass
18. END FUNCTION
```

TABLE 1. Summary of related works.

STUDY	YEAR	LANGUAGE	EXAMINED APPLICATIONS	DETECTED BAD SMELLS	ACCURACY(%)
Marinescu [18]	2004	Java, C++	2 Desktop Apps	God Class God Method	Accuracy Rate 98% Accuracy Rate 75%
Hecht et al. [9]	2015	Java	15 Popular Apps from Google Play Store	Blob, Swiss Army Knife, Long Method, Complex Class	F-Measure 88.89%
Palomba et al. [10]	2017	Java	18 Android Apps	15 Android Code Smells	F-Measure 98%
Rasool et al. [3]	2020	Java	15 Popular Apps from Google Play Store	25 Android Code Smells	F-Measure 96%
Prestat et al. [19]	2024	Bytecode	538 APK from F-DROID	7 Android Code Smells	Precision 92.8% Recall 53.4%
This Study	2024	Kotlin	5 Popular Android Kotlin Apps from Github	God Class Brain Method Brain Class	F-Measure 93.77% F-Measure 85.71% F-Measure 100%

```
char getChar(int a) {
    return (char)a;
}
```

(a) An example code snippet.



(b) The snippet's PSI tree. `PsiWhiteSpace` nodes are omitted for readability.

FIGURE 2. Program structure interface (PSI) tree [25].

if-statements), invoking methods, or all statements from a specific code fragment, and so forth.

III. RELATED WORK

Several related studies have been used as references and supporting materials in conducting this research. For example, a systematic review focused on code smells in Android projects. This study demonstrates that although Android applications have a different structure than desktop applications, the variety and density of smells are similar, but their distribution differs [20]. Another systematic review study related to static analysis on Android applications indicates that most static analyses are performed to locate vulnerabilities within Android applications' security. However, the tools and datasets are often not published [21]. Furthermore, an empirical study comparing the quality of Android applications written in Kotlin and Java found that Kotlin indeed improves the quality. The most commonly encountered smells in Kotlin programs are Long Method, Complex Class, and Blob [22]. An empirical study has also been conducted to analyze the impact of code smells on resource utilization, such as CPU and Memory, in Android applications. The results show that refactoring the code leads to a significant increase in CPU usage, between 12.7% and 13.7%, and a memory enhancement up to 7.1% [23].

Several research studies have developed an Android smell detection tool named PAPRIKA. This tool identifies smells by generating Java source code from APKs. However, it has a characteristic of making it challenging to set dynamic thresholds [9]. Another study proposed a tool called ADOCTOR. This tool detects smells by exploiting the Abstract Syntax Tree (AST) in Java source code. The detection process is carried out by combining metrics and straightforward text comparison [10]. Recent research has also proposed an Android smell detection tool that integrates Java source code analysis and metrics. However, due to the tool's dependency on JavaParser, the supported language is Java. Moreover, if there is a syntax error in the code smell, it could be incorrectly classified as a false positive or false negative [3]. In addition to discussing static analysis tools, recent research also proposes dynamic tools for detecting behavioral code smells. However, as this research analyzes bytecode instead of source code, there is a limitation: it cannot determine the specific line of code where the instruction containing the smell occurs. [19].

Table 1 shows a summary of these papers, the code smells studied, the languages used, and the accuracy achieved. As indicated in the table, Marinescu [18] research and our study share some similarities in discussing the same smells (related to class and method). However, our research is conducted in a different environment, specifically for Android

Kotlin development, utilizing the Program Structure Interface, which is an extended version of the Abstract Syntax Tree.

IV. PROPOSED METHOD

In this paper, we present a Kotlin bad smell code detection approach using the Program Structure Interface. This section provides a detailed description of the method we propose as well as our experiment design.

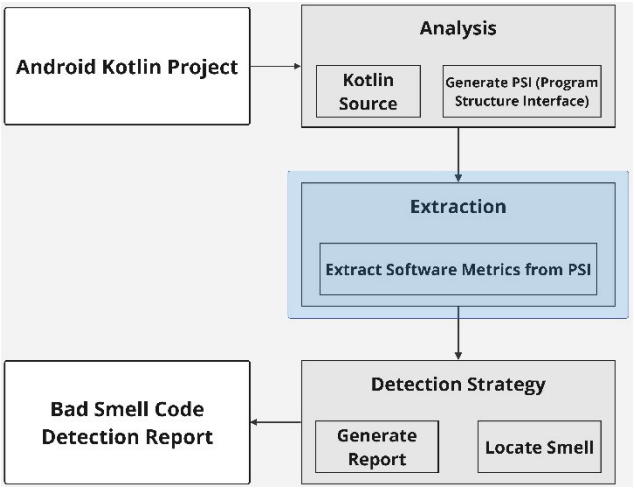


FIGURE 3. Architecture of detection approach.

A. DETECTION APPROACH

The detection of code bad smells is conducted using a software-metric-based approach. For the source code to be extracted into the required metrics, it must first be parsed. In this study, the Kotlin program used as input is converted into a PSI (Program Structure Interface). The flow of the tool in automatically detecting smells that will be created can be seen in Figure 3.

In general, there are three main processes involved: Analysis, Extraction, and Detection. The modified part of the proposed approach is the Extraction section. In this proposed approach, Extraction is performed using PSI, whereas the previous approach used AST.

- Analysis Process
  This starts with the source code parsing. In this stage, parsing is performed on the Android project that has been targeted. The parsing of the source code will then produce a PSI (Program Structure Interface).
- Extraction Process
  This is initiated by gathering class and method information from the PSI, and translating this data into metrics, exemplified in Figure 4. The CYCLO metric extraction initiates at a complexity counter of zero, which then increases based on encountered function types and expressions in Kotlin code. This includes NamedFunction, Binary Expression (with specific conditions), Continue or Break, If, Try, or Loop, and When expressions, alongside Call expressions for nesting functions with specific code content.

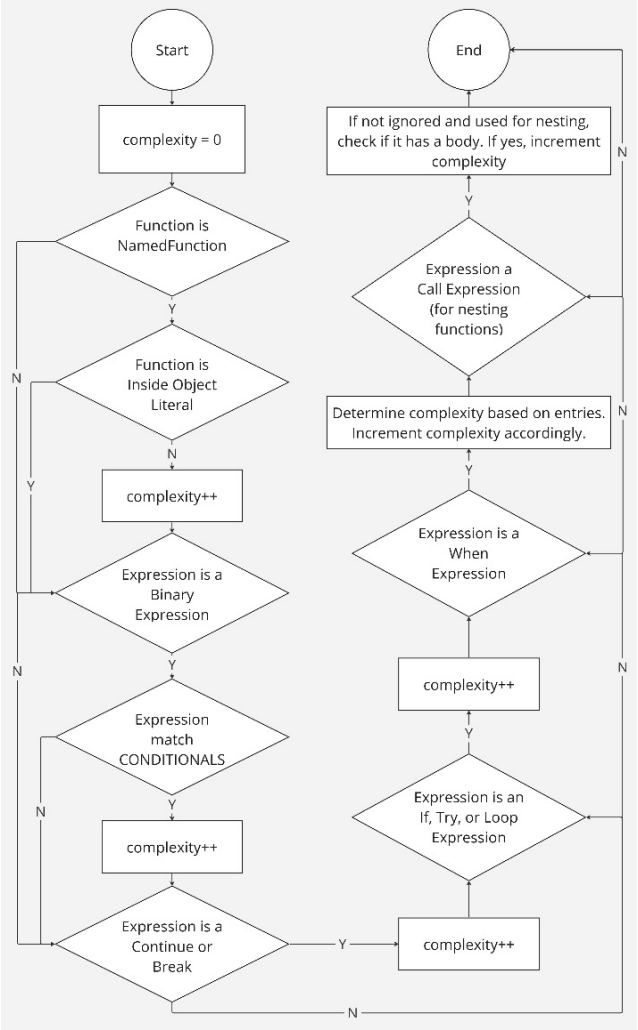


FIGURE 4. Flowchart of CYCLO metric extraction in PSI.

- Detection Process
  This is carried out using a detection strategy with the metrics obtained from the previous process. A collection of classes and methods that meet the criteria as smells will be compiled into a report.

After the three processes have been completed, a smell detection report is generated. The detection report presented contains information such as the names of classes or methods, location details, and the details of the software metric calculations.

B. EXPERIMENT DESIGN

In this study, experiments were conducted to measure the accuracy, effectiveness, performance, and quality of the developed tool. The observation variables that were intended to be measured in this experiment are as follows.

1. Evaluation of Detection Accuracy.
   This experiment aims to determine the precision of the detection results by conducting tests using various Kotlin Android projects (real-world apps) to evaluate whether the tool can detect smells in the source code that contain specific smells within them.
2. Comparative Evaluation of Detection Results with the iPlasma Tool.
   This experiment aims to compare the performance of code smell detection using the tool developed in this study with the tool used in previous research. The goal is to evaluate potential improvements or enhancements in the proposed detection approach.
3. Evaluation of Tool Usability.
   Testing will be conducted using a scoring method for the tool that has been tried by users. The scoring chosen by the users will have a range of 1 – 5, where this range represents strongly disagree – strongly agree with the intention of usage

C. CORPUS

In this study, 5 Android applications from four different categories were used. These applications were selected after reviewing several open-source projects available on GitHub, focusing on applications with at least 1 quantity of the studied smells and at least 30 classes, ensuring that the applications have different functions that could be tested. All applications were written using the Kotlin language. The selected applications were: *Bible,*[1] *Habitica,*[2] *Quran,*[3] *Rajin,*[4] *and Tachiyomi.*[5]

V. EXPERIMENTAL RESULTS

In this section, we present the steps taken for the implementation phase of the research. This includes experiments on measuring detection accuracy, processing time, and usability testing.

A. VALIDATION OBJECTS

Manual validation was carried out by 6 experts with experience in developing Android applications for 2–7 years. Beginning with the definition of each smell, the experts were tasked with identifying those smells across all projects. The results of this manual validation completed by the experts were then revisited by the author, ensuring the highest level of results were achieved. These findings are illustrated in Table 2.

B. ACCURACY EVALUATION OF DETECTION

The results of the first experiment conducted on the Android Bible project are summarized in Table 3.

The results of the second experiment conducted on the Habitica project are summarized in Table 4.

[1]https://github.com/yukuku/androidbible
[2]https://github.com/HabitRPG/habitica-android
[3]https://github.com/quran/quran_android
[4]https://github.com/23522046/IF5250_RAJIN
[5]https://github.com/tachiyomiorg/tachiyomi

TABLE 2. Manual validation by expert.

App Name	Number of Kt Files	Number of Brain Methods	Number of God Classes	Number of Brain Classes
Bible	13	3	5	1
Habitica	22	2	10	1
Quran	20	3	6	2
Rajin	27	0	1	0
Tachiyomi	18	3	9	0

TABLE 3. Precision, recall. and f-measure for android bible.

Smell	Precision	Recall	F-Measure
Brain Method	3/(3+0) = 100%	3/(3+0) = 100%	100.00%
God Class	5/(5+1) = 83%	5/(5+0) = 100%	90.91%
Brain Class	1/(1+0) = 100%	1/(1+0) = 100%	100.00%

TABLE 4. Precision, recall. and f-measure for habitica android.

Smell	Precision	Recall	F-Measure
Brain Method	2/(2+3) = 40%	2/(2+0) = 100%	57.14%
God Class	10/(10+2) = 83%	10/(10+0) = 100%	90.91%
Brain Class	1/(1+0) = 100%	1/(1+0) = 100%	100.00%

The results of the third experiment conducted on the Quran project are summarized in Table 5.

TABLE 5. Precision, recall. and f-measure for quran android.

Smell	Precision	Recall	F-Measure
Brain Method	3/(3+1) = 75%	3/(3+0) = 100%	85.71%
God Class	6/(6+1) = 86%	6/(6+0) = 100%	92.31%
Brain Class	2/(2+0) = 100%	2/(2+0) = 100%	100.00%

The results of the fourth experiment conducted on the Rajin App project are summarized in Table 6.

TABLE 6. Precision, recall. and f-measure for rajin app.

Smell	Precision	Recall	F-Measure
Brain Method	N/A	N/A	N/A
God Class	1/(1+0) = 100%	1/(1+0) = 100%	100.00%
Brain Class	N/A	N/A	N/A

The results of the fifth experiment conducted on the Android Tachiyomi project are summarized in Table 7.

TABLE 7. Precision, recall. and fmeasure for tachiyomi.

Smell	Precision	Recall	F-Measure
Brain Method	3/(3+0) = 100%	3/(3+0) = 100%	100.00%
God Class	9/(9+1) = 90%	9/(9+0) = 100%	94.74%
Brain Class	N/A	N/A	N/A

From the test results of 5 Android projects that have been conducted, it was found that the tool successfully detected 55 smell instances. God Class emerged as the most frequently detected smell, followed by Brain Method, and finally Brain Class. Based on the obtained results and from Figure 5, it's observed that the tool was capable of correctly identifying almost all smells across these five Android projects.

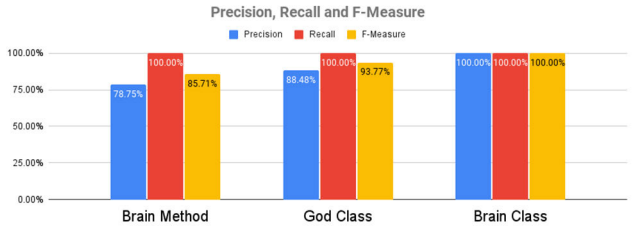


FIGURE 5. Accuracy detection evaluation results.

A manual analysis was performed to comprehend the reasons behind these results. From this analysis, several causes were discovered. Firstly, it appears that the tool detected some abstract classes as God Classes, such as the BaseActivity class in the Habitica project and PagerViewer in the Tachiyomi project. Upon closer inspection, these abstract classes do indeed possess greater size and functionality compared to other classes. This is due to the fact that these base classes are utilized (inherited) by other classes. However, according to expert judgment, such abstract classes do not qualify as God Classes, thus resulting in false positive detections by the tool.

Secondly, false positives were also found in the case of Brain Method, specifically in the deserialize method in the Habitica and Quran projects. Upon a more detailed examination, the deserialize method is found to perform only one function, i.e., to carry out deserialization on json objects. However, some objects such as those in the GroupSerialization, MemberSerialization, and UserDeserializer classes indeed possess more attributes in comparison to other classes like FAQArticleListDeserializer or ContentDeserializer, which also have the same method. According to expert judgment, this method does not fall under Brain Method, hence resulting in a false positive detection by the tool.

After identification of these issues from the validation results, it was revealed that these problems could be resolved by recalibrating the threshold of the software metrics used in the detection strategies for God Class and Brain Method. Adjusting the threshold values can be utilized as a means to enhance the tool's accuracy in smell detection because by altering these threshold values, the tool can recognize classes like BaseActivity and methods like deserialize as not being smells.

C. COMPARISON OF DETECTION PERFORMANCE WITH IPLASMA TOOLS

This testing was conducted by experimenting with detecting code smells from an Android Kotlin repository using the developed tool and then comparing the results with those obtained using another tool named iPlasma [24]. The repository selected for the testing was the Quran Android project.

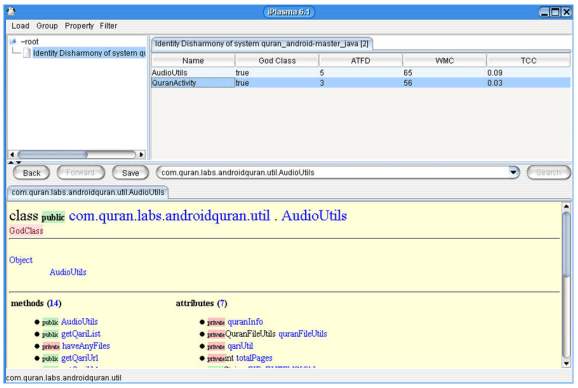


FIGURE 6. Detection results using iPlasma tool.

This project is an open-source project written in Kotlin. Since iPlasma only supports the Java programming language, the Quran Android project had to be converted to Java first with the help of a third party. Afterward, the iPlasma tool was run, and Figure 6 below shows the iPlasma interface while detecting code smells in the Java-converted Quran Android project.

It is observed that iPlasma only identified 2 God Classes. Subsequently, the developed tool was run to detect smells in the Quran Android Kotlin project, and Figure 7 below shows the interface when the proposed tool was executed.

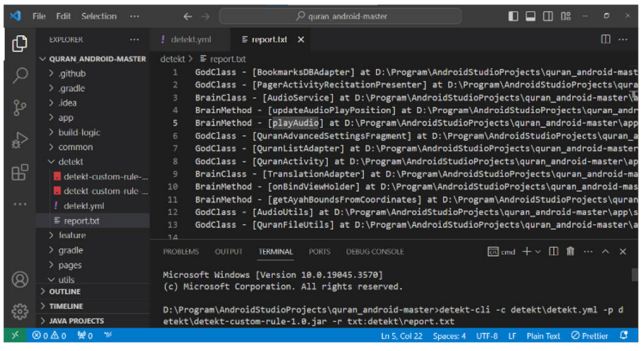


FIGURE 7. Detection results using the proposed tool.

From Figure 7, it's noted that the tool successfully identified 7 God Classes, 4 Brain Methods, and 2 Brain Classes. The comparison of smell detections by both tools can be seen in Table 8.

TABLE 8. Comparison of smell detection results.

SMELL	IPLASMA	PROPOSED TOOL
Brain Class	0	2
God Class	2	7
Brain Method	0	4

From Table 8, it is noted that iPlasma only detected 2 God Classes, namely the QuranActivity and AudioUtils classes. Whereas, the tool developed in this research was able to detect the existence of 7 God Classes, which are the BookmarksDBAdapter, PagerActivityRecitationPresenter, QuranActivity, QuranAdvancedSettingsFragment, QuranListAdapter, AudioUtils, and QuranFileUtils classes.

Furthermore, the tool was also successful in detecting other smells that weren't identified by iPlasma, namely Brain Class and Brain Method. There were two Brain Classes that were successfully detected by the tool, namely the AudioService and TranslationAdapter. Meanwhile, the Brain Methods that were identified amounted to four, derived from different classes, namely updateAudioPlayPosition and playAudio from the AudioService class, onBindViewHolder from the TranslationAdapter class, and getAyahBoundsFromCoordinates from the ImageAyahUtils class. The comparison graph of detection results between iPlasma and the tool can be seen in the following Figure 8.

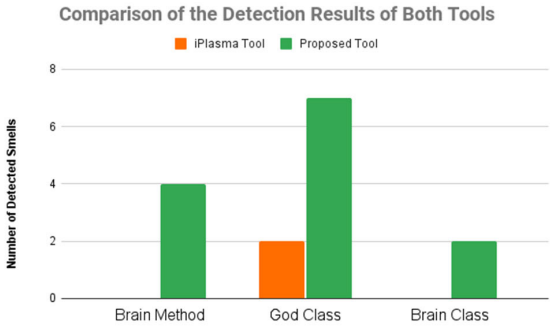


FIGURE 8. Comparison of detection results of both tools.

From the results obtained, it is known that there is a common smell detected by both tools, namely the God Class. Both tools similarly identified the AudioUtils and QuranActivity classes as God Classes. However, overall, the detection results using the tool developed show more smells were found.

D. USABILITY TESTING

This evaluation was conducted with 7 users experienced in developing Android applications. Each participant was asked to try using the tool on a provided Kotlin Android project. Participants were then asked to fill out a usability questionnaire. Figure 9 displays the results of the usability questionnaire for questions Q1 – Q5.

- The first question on the questionnaire pertained to the participant's understanding of the detection results displayed by the tool. The average rating given was 4.29, which indicates that participants agreed that the tool displayed the detection results in an understandable manner.
- The second question on the questionnaire addressed the functionalities provided by the tool. The average rating given was 4.43, indicating that participants agreed that the functionalities provided by the tool were easy to find.
- The third question on the questionnaire was about the clarity of information presented by the tool. The average rating given was 4.57, which shows that participants agreed that the information presented by the tool was clear.
- The fourth question on the questionnaire concerned the ease with which participants could calculate software metrics using the tool. The average rating given was 4.57, suggesting that participants agreed that with the tool, metric calculations were faster and more efficient.
- The final question on the questionnaire related to how easy it was for participants to detect smells in the Kotlin Android project using the tool. The average rating was 4.71, which indicates that participants agreed that with the tool, smell detection was quicker and more efficient.

From the usability testing results, it is known that almost all participants strongly agreed that the detection results by the tool were easy to understand, the functionalities presented were easy to find, the information presented was clear, and the tool helped in making the calculation of metrics and the detection of smells faster and more efficient.

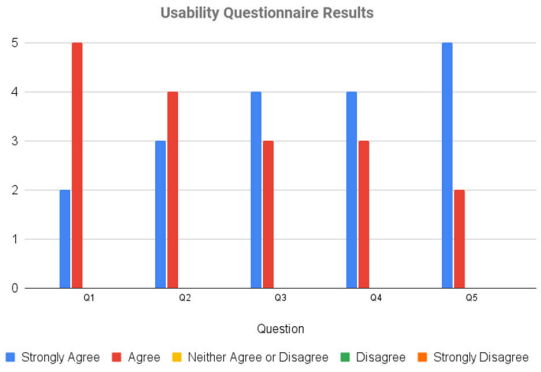


FIGURE 9. Usability testing results.

VI. CONCLUSION

By utilizing the Program Structure Interface representation and applying an approach based on software metrics, a Kotlin bad smell detection tool for Android was successfully developed. The results from the evaluation conducted on 5 Android projects show that the tool successfully detected a total of 55 smells, achieving an f-measure of 100% for Brain Class detection, 93.77% for God Class detection, and 85.71% for Brain Method detection. A manual analysis was carried out to understand the reasons behind these results, revealing that the tool tends to detect some abstract classes

as God Class, resulting in False Positives. The second reason is related to the Brain Method case, where the tool tends to detect methods that perform deserialization on objects with large sizes, leading to False Positives. It was also discovered from the results that this issue could be resolved by recalibrating the threshold of the software metrics used in the detection strategy for God Class and Brain Method. Adjusting the threshold values can serve as a means to enhance the tool's accuracy in smell detection because by changing these threshold values, the tool can recognize classes like BaseActivity and methods like deserialize as not being smells. Furthermore, the comparison of the tool's detection capabilities with iPlasma shows that the tool is able to detect more smells. The usability testing results indicate that out of 7 participants, nearly all strongly agree that the detection results from the tool are easy to understand, the functionalities provided are easy to find, the information presented is clear, and the tool helps in making metric calculation and smell detection faster and more efficient. The average value given for each question ranged from 4.29 – 4.71.

REFERENCES

[1] L. Ceci. (2009). *Number of Available Applications in the Google Play Store From December 2009 to March 2023*. Statista. Accessed: Mar. 12, 2023. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2] S. McIlroy, N. Ali, and A. E. Hassan, "Fresh apps: An empirical study of frequently-updated mobile apps in the Google play store," *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 1346–1370, Jun. 2016, doi: 10.1007/s10664-015-9388-2.

[3] G. Rasool and A. Ali, "Recovering Android bad smells from Android applications," *Arabian J. Sci. Eng.*, vol. 45, no. 4, pp. 3289–3315, Apr. 2020, doi: 10.1007/s13369-020-04365-1.

[4] M. Fowler, *Refactoring—Improving the Design of Existing Code* (Addison-Wesley Signature Series). Boston, MA, USA: Addison-Wesley Professional, 2018.

[5] T. Alkhaeir and B. Walter, "The effect of code smells on the relationship between design patterns and defects," *IEEE Access*, vol. 9, pp. 3360–3373, 2021, doi: 10.1109/ACCESS.2020.3047870.

[6] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proc. 16th Work. Conf. Reverse Eng.*, Oct. 2009, pp. 75–84, doi: 10.1109/WCRE.2009.28.

[7] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2009, pp. 390–400, doi: 10.1109/ESEM.2009.5314231.

[8] J. P. dos Reis, F. B. e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code smells detection and visualization: A systematic literature review," *Arch. Comput. Methods Eng.*, vol. 29, no. 1, pp. 47–94, Jan. 2022, doi: 10.1007/s11831-021-09566-x.

[9] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, "Detecting antipatterns in Android apps," in *Proc. 2nd ACM Int. Conf. Mobile Softw. Eng. Syst.*, May 2015, pp. 148–149, doi: 10.1109/MobileSoft.2015.38.

[10] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of Android-specific code smells: The aDoctor project," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 487–491, doi: 10.1109/SANER.2017.7884659.

[11] *Develop Android Apps With Kotlin*. Accessed: Apr. 5, 2023. [Online]. Available: https://developer.android.com/kotlin

[12] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, "Software design smell detection: A systematic mapping study," *Softw. Quality J.*, vol. 27, no. 3, pp. 1069–1148, Sep. 2019, doi: 10.1007/s11219-018-9424-8.

[13] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Berlin, Germany: Springer, 2006, doi: 10.1007/3-540-39538-5.

[14] F. M. Tua and W. Danar Sunindyo, "Software defect prediction using software metrics with Naïve Bayes and rule mining association methods," in *Proc. 5th Int. Conf. Sci. Technol. (ICST)*, vol. 1, Jul. 2019, pp. 1–5, doi: 10.1109/ICST47872.2019.9166448.

[15] *McCabe Object Oriented Tool User's Instructions*, McCabe & Associates, USA, 1994.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994, doi: 10.1109/32.295895.

[17] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," *ACM SIGSOFT Softw. Eng. Notes*, vol. 20, no. SI, pp. 259–262, Aug. 1995, doi: 10.1145/223427.211856.

[18] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, Sep. 2004, pp. 350–359, doi: 10.1109/ICSM.2004.1357820.

[19] D. Prestat, N. Moha, R. Villemaire, and F. Avellaneda, "DynAMICS: A tool-based method for the specification and dynamic detection of Android behavioral code smells," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, pp. 765–784, Apr. 2024, doi: 10.1109/TSE.2024.3363223.

[20] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in Android applications," in *Proc. IEEE/ACM Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*. New York, USA: ACM, May 2016, pp. 225–236, doi: 10.1145/2897073.2897094.

[21] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of Android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, Aug. 2017, doi: 10.1016/j.infsof.2017.04.001.

[22] B. Góis Mateus and M. Martinez, "An empirical study on quality of Android applications written in kotlin language," *Empirical Softw. Eng.*, vol. 24, no. 6, pp. 3356–3393, Dec. 2019, doi: 10.1007/s10664-019-09727-4.

[23] M. A. Alkandari, A. Kelkawi, and M. O. Elish, "An empirical investigation on the effect of code smells on resource usage of Android mobile applications," *IEEE Access*, vol. 9, pp. 61853–61863, 2021, doi: 10.1109/ACCESS.2021.3075040.

[24] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An integrated platform for quality assessment of object-oriented design," in *Proc. Int. Conf. Smart Multimedia*, 2005, pp. 1–4. [Online]. Available: https://api.semanticscholar.org/CorpusID:17455536

[25] JetBrains. *PSI Files*. Accessed: Sep. 13, 2023. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/psi-files.html

RADINAL DWIKI NOVENDRA received the master's degree in informatics from the Department of Informatics, School of Electrical Engineering and Informatics, Bandung Institute of Technology (ITB), in 2024. His work and research interests cover many subjects in knowledge and software engineering areas, including software quality, software processes, product improvement, requirements engineering, and software and data security. He is a Software Engineer on several national software development projects and is the author of a book on cross-platform mobile application development.

WIKAN DANAR SUNINDYO (Member, IEEE) received Sarjana Teknik degree (equivalence to Bachelor of Science) from Institut Teknologi Bandung, Indonesia, the M.Sc. degree in computational logic from Dresden University of Technology, Germany, and the Dr.techn. degree (equivalence to Ph.D.) from Vienna University of Technology, Austria. He is currently a Lecturer and a Researcher with Institut Teknologi Bandung. He studied computer science with emphasis on automation systems, quality assurance, data warehouse and semantics/ontology alignment and. His research work covers many subjects in knowledge and software engineering areas, including open data, big data, software quality, software process and product improvement, requirements engineering, and ICT for development. He is a Project Manager in several national development projects, an organizer of several international events and conferences, and a member of some national and international research projects. He is a member of Bandung Smart City Council.