International Journal on Advanced Science Engineering Information Technology

Vol.14 (2024) No. 1
ISSN: 2088-5334

An Empirical Evaluation on the Effect of Refactoring Code Smells Mobile Applications Android with ASATs on Resource Usage

Indira Syawanodya [a,*], Dian Anggraini [a], Fajar Muhammad Al-Hijri [a], Mochamad Iqbal Ardimansyah [a]

[a] *Department of Software Engineering, Universitas Pendidikan Indonesia, Bandung, 40625, Indonesia*
Corresponding author: [*]indira@upi.edu

***Abstract*— The Application is closely connected to mobile devices designed for many people and has maintenance, However, even maintenance can contain violations such as code smells that effect non-functional requirements, specifically the use of CPU and memory resources. When the software has a rapid use of resources, it gives rise to the phenomenon that the user may switch or uninstall the software. The solution to this phenomenon is to explore resource-related code smells and fix them by refactoring them. Developments to explore code smells came with ASATs, namely SonarQube, which 85,000 organizations are already using to speed up analyzing code in software. This topic is related to code smells, and the research objective is to analyze and compare the performance of the original versions and single or cumulative refactored versions of Android mobile software using the Design Research Methodology (DRM) approach. Code smells are represented based on the classification on SonarQube, namely Blocker, Critical, Major, and Minor, with code smells such as HashMap Usage, Member Ignoring Method, and Slow Loop. Aspects tested include Fixed Detection Ratio (FDR), improvement, CPU, and memory usage. Based on the results of the research, it shows the depreciation of code smells which is proven to significantly increase CPU performance in a single refactoring, namely Member Ignoring Method and Critical by 7.7% and 9.90%, respectively. Moreover, single refactoring offers developers advantages reducing high costs, diminished exertion, and truncated maintenance duration. However, the cumulative refactoring occasionally endeavors hold the potential be high improvements.**

***Keywords*— Software maintenance; refactoring; code smells; resource usage; Android.**

Manuscript received 8 Apr. 2023; revised 21 Aug. 2023; accepted 3 Oct. 2023. Date of publication 29 Feb. 2024.

I. INTRODUCTION

Technology is an integral section of people's lives, with application being a key component. Application is closely connected to mobile devices designed for many people [1]. Given the widespread use of application, it becomes a challenge for developers. Developers are responsible for the maintenance of applications; poor coding and implementation techniques can hinder long-term maintainability and lead to poor code quality [2], [3]. These issues arise due to a lack of awareness and lack of development experience or the need to develop applications rapidly under deadline pressure [4], [5]. Consequently, the developer requires maintenance program code on the application.

The maintenance of the application requires developers to review and modify program code manually. However, manual review can be time-consuming and inefficient [6]. The solution to that problem is using Automatic Static Analysis Tools (ASATs) that can automatically analyze program code and suggest corrective modifications [7]. ASATs are designed to help developers identify and understand deficiencies in program code [8]. SonarQube is a popular ASAT [9]. Despite the benefit of ASATs, during the maintenance of the application, developers may make bad decisions by ignoring rule violations that can cause conflicts and negative impacts on the application; the violations that can damage the maintenance of the application are code smells [10], which can affect on resource usage [4].

Resource usage should be considered a critical factor for the application's success, affecting performance and resource optimization [11]–[13]. The rapid consumption in applications can cause users to switch or uninstall the application from their device. However, the previous research on code smells related to resource usage has not fully explored the issue and has mainly used emulators [11], [14], [15]. It is important to note that behavior of an emulator differs from that of a physical device [1]. Therefore, there is a need to explore code smells that affect resource usage and effectively fix them to ensure optimal performance on physical devices as suggest by study [16]. This paper aims to analyze the effect

of refactoring code smells using ASATs on resource usage of the mobile application Android.

Hecht, et al. [4] have problems this study is on program code as code smells, that cause resource leaks and performance with metric frame time, number of delayed frames, memory usage, and number of garbage collection. There are three code smells i.e., HashMap Usage (HMU), Internal Getter/Setter (IGS), Member Ignoring Method (MIM) on two Android applications as corpus. The result of this study with refactoring MIM could be improved 12.4%, which means less frame delay. Furthermore, there was significant impact on refactoring HMU as much 3.6%.

Kwan Kim [14] discusses performance with metric CPU Time, problems this study is code smells, that cause performance degradation. There are seven code smells discusses i.e., Enhanced for Loop, Internal Getter/Setter, Local Variables, Avoiding Creating Unnecessary String Object, Use Static Final for Constants, Inefficient Data Structure, Avoid Using Recursive Methods on two Android applications. This study refactored the overall result CPU time of 579ms on Snake Game and 5.897ms on Bitmap Plasma. However, the study was incorrect in providing quantity code smells and not using tools to identification the code smells. Our study is not using IGS, because the code smells outdated and has no effect on performance [12].

Carette, et al. [13] assess code smells i.e., HashMap Usage (HMU), Internal Getter/Setter (IGS), and Member Ignoring Method (MIM) on five Android applications as a corpus. Three code smells cause the effect of leakage of battery energy consumption with metric average intensity and average voltage. The result of this study is that refactoring a single type such as MIM can reduce 3.86%, but refactoring cumulative reduce up to 4.83%.

Cruz and Abreu. [12] discuss problems related to the energy consumption of mobile devices using file changes and power measurement metrics. The study analyzed six applications, Loop – Habit Tracker, Writeily Pro, Talalarmo Alarm Clock, GnuCash, Acrylic Paint, and Simple Gallery, which are part of the F-Droid corpus. The study concluded that energy consumption decreased after refactoring ViewHolder (4.5%), DrawAllocation (1.5%), WakeLock (1.5%), ObsoleteLayoutParam (0.7%), and Recycle (0.7%). However, the study does not explore code smells.

Oliveira, et al. [1] present problem code smells on consume resource usage with metric CPU usage and memory usage on Android. There are three code smells discussed i.e., God Class, God Method, Feature Envy on nine Android applications as corpus. The results of this study can consume higher resource usage, one of them is that the Travel-Mate original version consumed 98.39 MB, after refactoring increased to 625.58 MB. In their study, have not inferential test to conclude the results are significant.

Verdecchia, et al. [17] discuss the problem of carbon emissions that reach a global scale in the information technology and communication sector. The problem requires maintenance to optimize energy efficiency, which involves aggressively fixing code smells using the Fisher-Yates Shuffle algorithm. The study used the average power consumption and time metrics on applications sourced from the GitHub corpus, namely CashManager, JTrac, and Spring-PetClinic. This study showed a decrease in energy consumption from 49.9% to 47.8% after refactoring. Additionally, cumulative refactoring of code smells can reduce performance by 6.8%, translating to a 10.7% decrease in energy consumption. Their study has no significant result and the metric used cannot be used to indicated impact of energy consumption. Our study will contribution inferential test to ensure and conclude the result.

Anwar, et al. [18] discuss problem energy consumption on Android applications with metric power measurement and refactoring using Fisher-Yates shuffle. There are five code smells discusses i.e., Long Method (LM), Feature Envy (FE), Type Checking (TC), Duplicated Code (DC), and God Class (GC) on three applications as corpus. This study showed a reduced energy consumption, on DC 10.8% and TC 10.5%. According to the author their result has significant result on DC and TC.

Alkandari, et al. [11] studied three code smells i.e., HashMap Usage (HMU), Member Ignoring Method (MIM), and Slow Loop (SL), on eight applications as a corpus. This study discusses problem resource usage in Android applications with metric CPU usage and memory usage. The result of this study performance improved on CPU usage 12.7% and 13.7% with single refactoring HMU and MIM. Furthermore, cumulative refactoring enhanced memory usage by 7.1%. Their result is not using a physical device and it is still questionable whether it has an effect. In our study we using physical device to ensure an impact.

Etemadi Someoliayi, et al. [19] another study discusses automatic refactoring code smells namely Sorald, this study discusses tools refactoring with deep analysis. This study related code smells raised by SonarQube.

Wibowo, et al. [15] discusses two code smells i.e., no-equal-then-else and prefer-conditional-expressions on one application. This study discusses problem code smells impact to performance in agricultural mobile applications. The result of this study, CPU usage increases 3.27% and memory usage decreases from 200M to 100M. In their study are not using physical device to ensure an effect and no significant result.

Previous studies have focused on non-functional energy has been extensively study and concluded the refactoring code smells can save energy consumption on applications, even though there are drawbacks. However, the study of code smells on non-functional resource usage remains insufficient. There exists a device gap between emulated and physical device. The fact behavior of applications on emulated and physical device is significantly different, even the emulated version fully mimics to properties and settings of the physical device [20]. Furthermore, a research gap regarding code smells exists between the identification of code smells based on previous studies and the characterization of code smells through the SonarQube tool, because code smells based SonarQube tool has not been studied on the resource usage in mobile applications. Therefore, our study focuses on the resource usage with metric CPU usage and memory usage, similar to the study by Oliveira et al. [1], Alkandari et al. [11], and Wibowo et al. [15]. For measures intensity ratio of code smells our study using FDR metric, is also similarly to the study conducted by Etemadi Someoliayi, et al. [19]. This study adopts three code smells i.e., HashMap Usage, Slow Loop, and Member Ignoring Method from conducted by Palomba, et. al. [21] and Alkandari, et. al. [11]. Our study

using code smells from SonarQube which not have been studied on resource usage Android i.e., Blocker, Critical Major, and Minor. We made ObreusDroid for aggregation of the resource usage and physical device to ensure the actually effect. Furthermore, our study will scientific contribution on software engineering related refactoring single and cumulative of mobile applications and tested using physical device and inferential statistic to conclude result significant to ensure the validity of the significance in line with previous studies [11].

II. Material and Method

This paper focused on refactoring code smells for mobile applications. Aggregation of the resource usage using manual testing which focused on Graphical User Interface (GUI) events only [22], [23]. Thus, our study used ObreusDroid which was designed and developed using bash scripting and Android Debug Bridge (ADB). Our study consists of six main steps as shown in Fig 1.

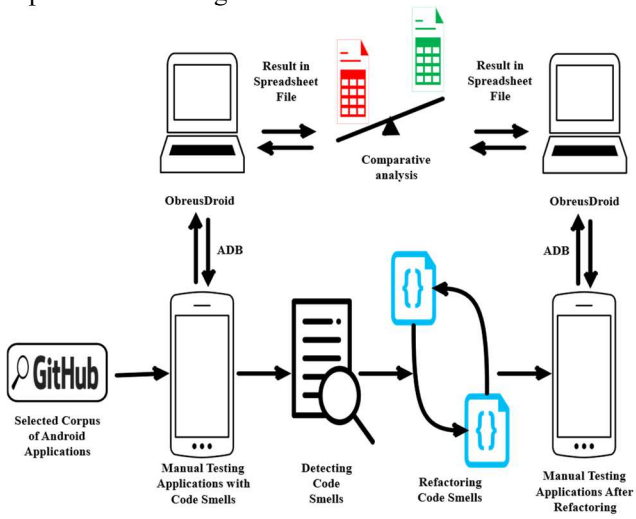

Fig. 1 Analysis Steps

A. Step 1: Selected Corpus of Android Applications

The Android applications selected were sourced from F-Droid which redirected to GitHub; we chose only mobile applications from a previous study by Anwar et al. We believed that paper because the applications have been researched, according to their author, the applications has been downloaded more than 10.000 on Google Play Store (GPS) with '4' review [18] as shown Table I and ease to find code smells. Then, we declared the configuration to be able to be analyzed by SonarQube. The configuration assigned to each application in the Android Gradle file is shown in the example Fig 2.

TABLE I
CHARACTERISTICS OF MOBILE APPLICATIONS ANDROID

Characteristics	Calculator	Todo-List	Openflood
Category	Tools	Tools	Puzzle
Line of Code (LoC)	5935	5466	948
Project age	6.8	3	3.1
Downloaded (GPS)	More than 1.000	More than 10.000	More than 10.000
Average user review	4.5	4	4.6

```
apply plugin: 'org.sonarqube'
dependencies {
…
classpath "org.sonarsource.scanner.gradle:sonarqube-gradle-plugin:3.5.0.2730"
}
sonarqube {
    properties {
        property "sonar.projectName", "[projectname]"
        property "sonar.projectKey", "[projectkey]"
        property "sonar.host.url", "http://localhost:9000"
        property "sonar.language", "java"
        property "sonar.sources", "src/main/java"
        property "sonar.java.sources", "src/main/java"
        property "sonar.sourceEncoding", "UTF-8"
        property "sonar.login", "[token]"
        property "sonar.scm.disabled" , "true"
    }
}

```

Fig. 2 Configuration of SonarQube

B. Step 2: Manual Testing Applications with Code Smells

After the previous step, we ran each application individually as much as three times. For several reasons, we made a test case, as shown in Table II, to collect data on resource usage, i.e., CPU and memory, through ObreusDroid with ADB. First, another research [21]–[23] uses the top commands to measure the CPU usage and memory usage; we believe the result is to evaluate each application. Second, we exported the result of resource usage to a CSV file.

C. Step 3: Detecting Code Smells

After manual testing in the previous step, we detected the code smells automatically and manually. We manually chose three code smells from the previous study, i.e., HMU, MIM, and SL, because three code smells are easy to refactor. Automatically, we use popular ASATs, specifically SonarQube [7], [24]–[26], to find other code smells that have not been studied in focused resource usage on Android applications, detailed code smells of SonarQube as shown in Table III. SonarQube is already used by over 85.000 organizations [9], [10].

TABLE II
TEST SCENARIOS OF APPLICATIONS

Android Applications	Scenarios	Approximate Duration (Seconds)
Calculator	1. Add up both numbers and press the result. 2. Subtraction on both numbers and hit results. 3. Multiply on both numbers and hit result. 4. Division on both numbers and hit result 5. Clear all history on the calculator 6. Launches advanced calculator and returns to basic calculator 7. Draw a chart 8. Zoom in and zoom out the chart	84

Android Applications	Scenarios	Approximate Duration (Seconds)
	9. Select the hex calculator 10. Add up both hex numbers and hit the result. 11. Subtraction on both hex numbers and hit results. 12. Multiply on both hex numbers and hit result. 13. Division on both hex numbers and hit result 14. Back to basic calculator 15. Open the calculator history list 16. Select a history 17. Close history list 18. Press a number, select an arithmetic operation, and press result to display errors in calculations 19. Switch to the advanced calculator and press the square root and pi keys, then delete the result 20. Switch to the basic calculator and select a number, then long press on the screen to cut and paste from the clipboard.	
Todo-List	1. View the application tutorial and navigate to completion 2. Go to settings, then turn on progress by Subtasks 3. Add three new lists by typing each list uniquely, namely “list1”, “list2”, and “list3” 4. Press the “+” icon on the bottom right-side button on the screen 5. Add three new tasks by typing each task uniquely, namely "Task1", "Task2", and "Task3," and setting the priority; each three task is given priority uniquely, namely "High," "Medium," and "Low." 6. Add deadlines and reminders of three tasks 7. Put the list of three different tasks 8. Enter three subtasks in the main task 9. Mark the subtask complete and see progress; on "Task1," mark one subtask; on "Task2," mark two subtasks; on "Task3," mark three subtasks 10. Send the main task to the recycler by deleting it 11. View tasks based on priority 12. View tasks by deadline 13. Open the calendar from the left menu 14. In the calendar, click a date to open the task 15. View tasks in the recycler 16. View and use the help options 17. View info about the app 18. Search for tasks by one of the names	188
Openflood	1. Launch activity info 2. Play the game and lose 3. Close the "game over" pop-up by selecting Start a new game 4. Launch settings and change the board size to 12x12 5. Press all the color buttons in the game 6. Launch settings and choose color blind mode (blind mode) 7. Press all color buttons in the game on color blind mode (blind mode) 8. Launch settings and select the old color scheme 9. Press all color buttons in-game on the old schema color 10. Launch settings and change the board size to 24x24	32

TABLE III
DETAILED CODE SMELLS OF SONARQUBE

Key rule	Classifications	Descriptions
S3516	Blocker	The return value on the method doesn't be invariant
S2387	Blocker	Child class fields cannot have the same naming attribute as the parent class.
S1192	Critical	String literals cannot be duplicated
S1186	Critical	The method cannot be empty
S131	Critical	The switch statement must have a “default” clause
S2093	Critical	Try with resources must be used
S3973	Critical	The single conditional must fit the indentation
S115	Critical	Constants must obey naming conventions
S1604	Major	A class containing one method must be a lambda
S1144	Major	Unused “private” method must be removed
S1161	Major	“@Override” must be used on overridden methods and implementations
S1172	Major	Parameters in methods that are not used should be removed
S125	Major	Sections of code should not be commented
S1066	Major	The collapsible if statements must be concatenated
S5993	Major	The "abstract" class constructor cannot be declared "public."
S1068	Major	The unused “private” field should be deleted
S108	Major	Nested code cannot be left blank
S1854	Major	Values that are stored in variables but not used must be deleted
S4165	Major	Variables containing hold values cannot be redundant
S2589	Major	Boolean expressions cannot be arbitrary
S1117	Major	Local variables cannot duplicate class fields
S3740	Major	Generic data types may not be used
S1301	Minor	Changed the "switch" statement to an "if" statement to improve readability
S1874	Minor	The code "@Deprecated" is not required

Key rule	Classifications	Descriptions
S2293	Minor	The diamond (“<>”) operation must be used
S1596	Minor	“Collections.EMPTY_LIST”, “EMPTY_MAP”, and “EMPTY_SET” must not be used
S1450	Minor	Private local variables outside the method must be local variables within the method.
S1155	Minor	Collection.isEmpty() should be used to test for empty values
S1643	Minor	Strings cannot be concatenated using "+" into one
S1128	Minor	Unnecessary imports should be removed
S3400	Minor	Methods cannot return constant values
S1905	Minor	Redundancy casts should not be used
S1659	Minor	Variables cannot be declared on a single line
S3008	Minor	Non-final static naming must comply with naming conventions
S1125	Minor	Boolean literals cannot be redundant
S1488	Minor	Local variables must be undeclared and must be returned.
S1124	Minor	Modifiers must be declared in order
S1116	Minor	Blank statements must be deleted or filled in
S1126	Minor	Boolean expression returns cannot be wrapped in an if-else statement
S1104	Minor	Variables in class fields must not have public accessibility
S1319	Minor	The declaration must use a Java collection interface such as "List" rather than its implementation-specific class like “LinkedList”
S1197	Minor	The array pointer “[]” must be a data type, not a variable.
S1153	Minor	String.valueOf() cannot be added to a String
S1481	Minor	Unused variables must be deleted
S1640	Minor	Maps with keys are enum values that must be changed with EnumMap
S3626	Minor	Jump statements must not be redundant
S4201	Minor	A blank value check should not be used with "instanceof."
S117	Minor	Local variable and parameter names must conform to convention
S116	Minor	The naming of class fields must be according to the convention
S1170	Minor	Constants on initialized fields should be declared “static final” rather than just “final.”

D. Step 4: Refactoring Code Smells

After manual testing in previous step, we manually refactoring code smells individually and cumulatively. The refactoring purpose is to improve internal quality applications [30]–[34]. Furthermore, we separated to be another edition, as follows:

$\mathbf{E}_{\text{Original}}$ - Code smells in the original edition are not refactored.
$\mathbf{E}_{\text{HMU}}$ - Code smells HashMap Usage in this edition has been refactored.
$\mathbf{E}_{\text{SL}}$ - Code smells Slow Loop in this edition has been refactored.
$\mathbf{E}_{\text{MIM}}$ - Code smells Member Ignoring Method in this edition has been refactored.
$\mathbf{E}_{\text{Blocker}}$ - Code smells Blocker classification in this edition has been refactored.
$\mathbf{E}_{\text{Critical}}$ - Code smells Critical classification in this edition has been refactored.
$\mathbf{E}_{\text{Major}}$ - Code smells Major classification in this edition has been refactored.
$\mathbf{E}_{\text{Minor}}$ - Code smells Minor classification in this edition has been refactored.
$\mathbf{E}_{\text{All}}$ - All code smells in this edition have been refactored.

E. Step 5: Manual Testing Applications After Refactoring

After refactoring code smells manually, we run each application, which has been done refactoring individually and cumulatively with manual testing similar to previous step 2. We run each application 24 times in appropriate editions except $\mathbf{E}_{\text{ori}}$. The result of resource usage collected and evaluated related consumption.

F. Step 6: Comparative Analysis

After completing all the previous steps, we compared the resource consumption of Android applications before and after refactoring. We measure the average CPU [1], [11] and memory usage [1], [11] as well as the improvement [12]. Furthermore, we conducted inferential statistical analysis to conclude the results. The formulation of measure as shown below:

$$FDR = \frac{|All\ Detected\ Before\ Repair| - |All\ Detected\ After\ Repair|}{|All\ Detected\ Before\ Repair|} \quad (1)$$

$$Improvement = \frac{x_{refactored} - x_{original}}{x_{original}} \quad (2)$$

III. Result and Discussion

The measure results of resource usage were using three (3) applications with different categories and benchmark LoC, which has been described in Table I.

Experimental Setup. An Android physical device is needed to test a mobile application. This study used the device specifically Xiaomi Redmi S2 (YSL) running Android 10 operating system with processor 2 GHz octa-core Qualcomm MSM8953 Snapdragon 625 and 3GB RAM. Furthermore, experiments for monitoring used with Windows 10. The PC has an i3 7th Gen Intel processor with 12GB of RAM. The analysis of the result in this study is measure FDR between the original edition and refactored edition, evaluate CPU usage, and evaluated memory usage.

A. Intensity of Refactoring Code Smells

The analysis of results in this study is measure FDR between the original edition and the refactored edition. The code smells of all Android applications are covered and the whole code smells was detected automatically and manually. Information on all the intensity of code smells is shown in Table IV without HMU, SL, MIM because it does not have a key rule and we detailed result intensity refactored.

TABLE IV
RESULT CODE SMELLS OF SONARQUBE REFACTORED

Applications	Key Rules	Refactoring Code Smells	Code Smells Detected	FDR
Calculator	Java: S3516	1	1	100%
	Java: S1186	18	18	100%
	Java: S131	10	17	59%
	Java: S1604	32	32	100%
	Java: S1144	2	2	100%
	Java: S1161	6	6	100%
	Java: S1172	5	6	83%
	Java: S125	1	1	100%
	Java: S1066	3	5	60%
	Java: S5993	2	2	100%
	Java: S2589	1	1	100%
	Java: S1068	2	2	100%
	Java: S108	1	1	100%
	Java: S1301	9	9	100%
	Java: S1874	10	20	50%
	Java: S2293	8	8	100%
	Java: S1596	1	1	100%
	Java: S1450	2	2	100%
	Java: S1155	1	2	50%
	Java: S1643	3	3	100%
	Java: S1640	1	1	100%
	Java: S1128	1	1	100%
	Java: S1124	1	3	33%
	Java: S3400	2	2	100%
	Java: S1905	2	2	100%
	Java: S1659	1	1	100%
	Java: S3008	1	1	100%
	Java: S1125	2	2	100%
	Java: S1488	2	2	100%
	Java: S117	2	2	100%
Total		133	157	85%
Todo-List	Java: S2387	6	6	100%
	Java: S1192	12	12	100%
	Java: S2093	1	1	100%
	Java: S1186	5	5	100%
	Java: S131	8	8	100%
	Java: S3973	2	2	100%
	Java: S115	1	1	100%
	Java: S5993	2	2	100%
	Java: S1068	20	20	100%
	Java: S125	38	38	100%
	Java: S1161	10	10	100%
	Java: S108	5	5	100%
	Java: S1854	10	10	100%
	Java: S1604	51	51	100%
	Java: S1066	2	3	67%
	Java: S4165	1	1	100%
	Java: S2589	2	2	100%
	Java: S1117	7	7	100%
	Java: S3740	1	1	100%
	Java: S1144	1	1	100%
	Java: S1172	2	2	100%
	Java: S1128	28	28	100%
	Kotlin: S1128	7	7	100%
	Java: S1874	6	83	7%
	Java: S1116	2	2	100%
	Java: S1155	4	6	67%
	Java: S2293	21	21	100%
	Java: S1319	10	17	59%
	Java: S1125	2	2	100%
	Java: S1126	4	4	100%
	Java: S1104	20	23	87%
	Java: S1197	9	9	100%
	Java: S1153	1	1	100%

Applications	Key Rules	Refactoring Code Smells	Code Smells Detected	FDR
	Java: S116	1	2	50%
	Java: S1659	6	6	100%
	Java: S1450	11	14	79%
	Java: S1481	8	8	100%
	Java: S1301	8	8	100%
	Java: S1640	1	1	100%
	Java: S3626	2	2	100%
	Java: S1488	2	2	100%
	Java: S1124	2	2	100%
	Java: S4201	1	1	100%
	Java: S117	2	2	100%
Total		345	439	78%
Openflood	Java: S1192	8	8	100%
	Java: S131	1	1	100%
	Java: S1161	2	2	100%
	Java: S1604	19	19	100%
	Java: S2589	1	1	100%
	Java: S1066	1	1	100%
	Java: S3626	18	18	100%
	Java: S1197	4	4	100%
	Java: S1643	1	1	100%
	Java: S1905	1	1	100%
	Java: S2293	2	2	100%
	Java: S1659	4	4	100%
	Java: S116	1	1	100%
	Java: S1170	1	1	100%
	Java: S1128	3	3	100%
	Java: S1301	1	1	100%
Total		68	68	100%

Code smells in the Calculator have been refactored 85%, the code smells of Todo-List have been refactored 78%, and code smells of Openflood have been refactored 100%.

B. Interpretation CPU Usage

In Figs. 3, 4, and 5 we visualized CPU usage of all editions of applications. CPU usage interpretation has different result at any moment run. In the Table V. We present average and standard deviation of the CPU usage of nine editions applications, which then measured relative change to see improvement. The negative value indicated that the result of refactoring caused improvement in performance, and the value positive indicated that the result of refactoring caused poor performance.

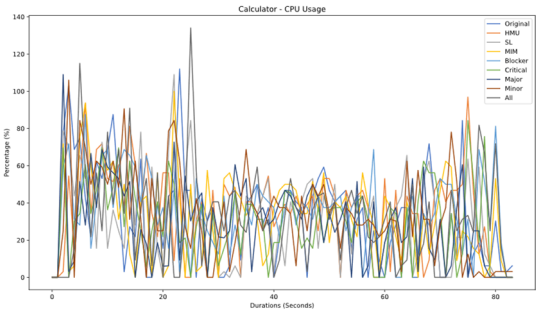

Fig. 3 CPU usage of Calculator

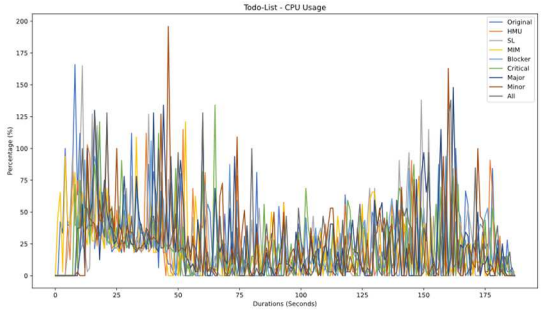

Fig. 4 CPU usage of Todo-List

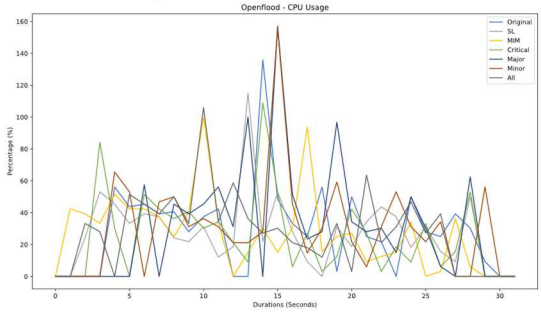

Fig. 5 CPU usage of Openflood

Refactoring HashMap Usage, one of the two applications shows an improvement in CPU performance, with an average of 4.14%. For Slow Loop refactoring, the three pieces of applications improved CPU performance, by an average of 4.39%. In the Member Ignoring Method, two of the three applications generate improvement in CPU performance, with an average of 7.87%. In Blocker, refactoring of one of the two applications resulted in CPU performance, averaging 4.75%. Furthermore, Critical refactoring of the three of applications

improved CPU, with an average of 9.90%. Major refactoring of one in three pieces of applications resulted in CPU performance, with an average of 0.28%. At the same time, Minor refactoring of one in three applications results in CPU performance, with an average of 0.90%. Finally, two of the three applications resulted in CPU performance in cumulative refactoring, with an average of 3.96%.

TABLE V
PERCENTAGE DIFFERENT EDITIONS OF AVERAGE CPU USAGE

%	Calculator			Todo-List			Openflood		
Editions	**Average**	**Standard Deviation**	**Improvement**	**Average**	**Standard Deviation**	**Improvement**	**Average**	**Standard Deviation**	**Improvement**
E_{Ori}	32,02%	26,38%	0%	26,18%	27,72%	0%	28.56%	27.79%	0%
E_{HMU}	34,74%	23,55%	+8,49%	21,79%	25,06%	-16,77%	-	-	-
E_{SL}	31,15%	25,90%	-2,72%	25,06%	30,83%	-4,28%	26,80%	22,85%	-6,16%
E_{MIM}	32,37%	23,56%	+1,09%	21,33%	23,14%	-18,53%	26,80%	24,16%	-6,16%
$E_{Blocker}$	34,58%	21,22%	+8,00%	21,60%	24,30%	-17,49%	-	-	-
$E_{Critical}$	28,97%	21,55%	-9,53%	23,44%	26,18%	-10,47%	25,79%	25,98%	-9,70%
E_{Major}	28,61%	22,48%	-10,65%	26,61%	30,87%	+1,64%	30,89%	36,38%	+8,16%
E_{Minor}	33,21%	24,40%	+3,72%	23,58%	29,97%	-9,93%	29,56%	21,10%	+3,50%
E_{All}	32,17%	26,18%	+0,47%	23,61%	28,26%	-9,82%	27,84%	23,79%	-2,52%

Based on the statistical analysis in Table VI, the refactoring of code smells significantly impacts Android applications. In previous studies the results of a p-value of less than 0.1 certain percentage of 90%, has a significant effect [12] or a p-value of less than equal to 0.1 certain percentage 90%. These results show that statistically refactoring Member Ignoring Method and Critical can improve with refactoring singly, with an average of 7.87% and 9.90% in Table VI. In the results of the p-value with 0.074 belonging to the Member Ignoring Method and 0.109 belonging to Critical, even though Critical is slightly more than 0.1 it is enough to produce statistically significant results present 80%.

TABLE VI
WILCOXON SIGNED-RANK INFERENTIAL TEST OF CPU USAGE

	E_{Ori}	E_{HMU}	E_{SL}	E_{MIM}	$E_{Blocker}$	$E_{Critical}$	E_{Major}	E_{Minor}
E_{HMU}	0.219	-	-	-	-	-	-	-
E_{SL}	0.284	0.757	-	-	-	-	-	-
E_{MIM}	**0.074**	0.706	0.448	-	-	-	-	-
$E_{Blocker}$	0.529	0.981	0.963	0.620	-	-	-	-
$E_{Critical}$	**0.109**	0.727	0.536	0.897	0.503	-	-	-
E_{Major}	0.745	0.841	0.777	0.394	0.596	**0.044**	-	-
E_{Minor}	0.346	0.854	0.663	0.556	0.852	0.336	0.603	-
E_{All}	0.449	0.823	0.905	0.410	0.824	0.483	0.925	0.762

C. Interpretation Memory Usage

Fig. 6, Fig. 7, and Fig. 8, we visualized memory usage of all editions in the same applications. Memory usage interpretation has different results at any moment run. In the Table VII. We present the average and standard deviation of the memory usage of nine editions applications, which then measured relative change to see improvement. Improvement of memory usage has improved variative similar CPU usage.

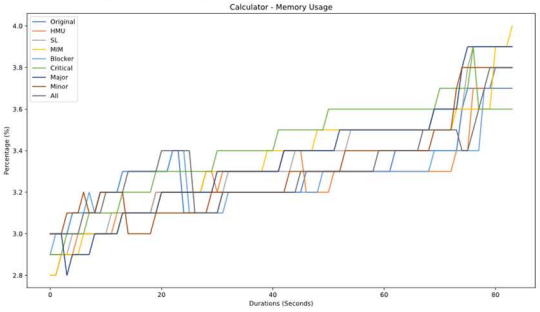

Fig. 6 Memory Usage of Calculator

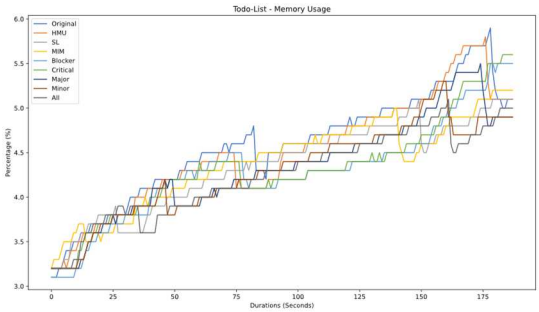

Fig. 7 Memory usage of Todo-List

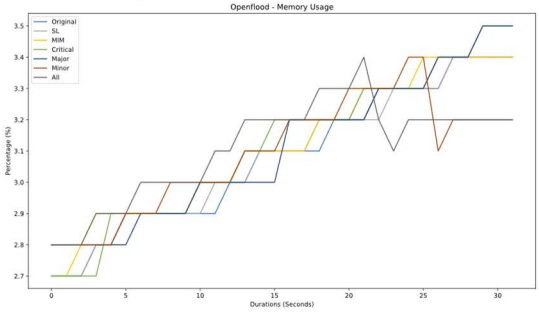

Fig. 8 Memory usage of Openflood

Refactoring HashMap Usage, the two applications show an improvement in memory performance, with an average of

1.48%. For the Slow Loop refactoring, one in three pieces of applications resulted in an improvement in memory performance by an average of 1.17%. In the Member Ignoring Method, one out of three applications result in an improvement in memory performance, with an average of 0.59%. The Blocker refactoring of the two applications results in memory performance, averaging 3.52%. Subsequent Critical refactoring of one in three pieces of applications resulted in a memory improvement by an average of 0.25%. Major refactoring on two of the three applications resulted in memory performance, averaging 1.10%. Minor refactoring on one of the three applications results in memory performance, with an average of 1.94%. Finally, one in three applications results in memory performance on refactoring seven code smells, with an average of 2.09%.

TABLE VII
PERCENTAGE DIFFERENT EDITIONS OF AVERAGE MEMORY USAGE

%	Calculator			Todo-List			Openflood		
Editions	**Average**	**Standard Deviation**	**Improvement**	**Average**	**Standard Deviation**	**Improvement**	**Average**	**Standard Deviation**	**Improvement**
E_{Ori}	3,30%	0,19%	0%	4,57%	0,60%	0%	3,08%	0,21%	0%
E_{HMU}	3,26%	0,19%	-1,21%	4,49%	0,62%	-1,75%	-	-	-
E_{SL}	3,35%	0,26%	+1,52%	4,34%	0,50%	-5,03%	3,08%	0,21%	0%
E_{MIM}	3,34%	0,25%	+1,21%	4,39%	0,49%	-3,94%	3,11%	0,21%	+0,97%
$E_{Blocker}$	3,27%	0,15%	-0,91%	4,29%	0,56%	-6,13%	-	-	-
$E_{Critical}$	3,41%	0,22%	+3,33%	4,31%	0,55%	-5,69%	3,13%	0,23%	+1,62%
E_{Major}	3,35%	0,27%	+1,52%	4,32%	0,56%	-5,47%	3,10%	0,23%	+0,65%
E_{Minor}	3,30%	0,23%	0%	4,29%	0,51%	-6,13%	3,09%	0,17%	+0,32%
E_{All}	3,31%	0,18%	+0,30%	4,24%	0,47%	-7,22%	3,10%	0,15%	+0,65%

Based on the statistical analysis in Table VIII, the refactoring code smells has a significant effect on Android applications. These results show that statistically refactoring the seven code smells singly or cumulatively can significantly improvement memory usage. So, it was concluded that refactoring HMU, SL, MIM, Blocker, Critical, Major, and Minor can result in an improvement in memory performance. Despite one of application (openflood) is poor memory after refactoring in Table VII. This concludes refactoring singly or cumulatively significant, but not whole improvement.

TABLE VIII
WILCOXON SIGNED-RANK INFERENTIAL TEST OF CPU USAGE

	E_{Ori}	E_{HMU}	E_{SL}	E_{MIM}	$E_{Blocker}$	$E_{Critical}$	E_{Major}	E_{Minor}
E_{HMU}	**0.000**	-	-	-	-	-	-	-
E_{SL}	**0.000**	**0.000**	-	-	-	-	-	-
E_{MIM}	**0.000**	0.359	**0.000**	-	-	-	-	-
$E_{Blocker}$	**0.000**	**0.000**	**0.000**	**0.000**	-	-	-	-
$E_{Critical}$	**0.000**	**0.000**	0.819	0.344	**0.000**	-	-	-
E_{Major}	**0.000**	**0.000**	**0.027**	**0.001**	**0.000**	0.664	-	-
E_{Minor}	**0.000**	**0.000**	**0.000**	**0.000**	0.353	**0.000**	**0.000**	-
E_{All}	**0.000**	**0.000**	**0.000**	**0.000**	0.544	**0.000**	**0.000**	**0.000**

IV. CONCLUSION

Based on the result of our study, we concluded that refactoring with ASATs can help detect code smells and refactoring code smells with percentage on each application i.e., Calculator 85%, Todo-List 78%, and Openflood 100%. The test results each application obtained on real device with test case represent that refactoring Member Ignoring Method and Critical results in an improvement performance on CPU usage with an average of 7.87% and 9.90%, there is an improvement in memory performance in refactoring Blocker and cumulative refactoring on seven code smells with an average of 3.52% and 2.09%, this signifies code smells classification as “Blocker” and “Critical” have an impact on the improvement performance CPU usage and memory usage other than found in previous studies. The statistical results obtained can significantly prioritize that refactoring code smells is crucial to adjusting especially in the use of resources or the performance for multitasking. In addition, single refactoring offers developers advantages reducing high cost, diminished exertion, and truncated maintenance duration. However, the cumulative refactoring occasionally endeavors hold the potential be high improvements. Our study has a limitation concerning the relatively small size of the corpus, testing each application manually. In the future work, we intend to develop automatically tools with setting specific code smells to improvement resource usage which combine code smells has been studied, impacting either CPU usage exclusively (such as Critical, MIM, SL, and Cumulative improvements for CPU usage) or memory usage exclusively (including HMU, Blocker, and Cumulative improvements for memory usage) and will be implemented on large size of the corpus and each application will testing automatically.

REFERENCES

[1] J. Oliveira, M. Viggiato, M. Santos, E. Figueiredo, and H. Marques-Neto, “An Empirical Study on the Impact of Android Code Smells on Resource Usage,” Jul. 2018, pp. 314–359, doi: 10.18293/SEKE2018-157.

[2] S. Habchi, N. Moha, and R. Rouvoy, “Android code smells: From introduction to refactoring,” *J. Syst. Softw.*, vol. 177, p. 110964, 2021, doi:10.1016/j.jss.2021.110964.

[3] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, “Understanding code smells in android applications,” *Proc. - Int.*

Conf. Mob. Softw. Eng. Syst. MOBILESoft 2016, pp. 225–236, 2016, doi: 10.1145/2897073.2897094.

[4] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of Android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, May 2016, pp. 59–69, doi:10.1145/2897073.2897100.

[5] G. Rasool and Z. Arshad, "A Lightweight Approach for Detection of Code Smells," *Arab. J. Sci. Eng.*, vol. 42, no. 2, pp. 483–506, 2017, doi:10.1007/s13369-016-2238-8.

[6] D. Stefanović, D. Nikolić, D. Dakić, I. Spasojević, and S. Ristić, "Static code analysis tools: A systematic literature review," *Ann. DAAAM Proc. Int. DAAAM Symp.*, vol. 31, no. 1, pp. 565–573, 2020, doi: 10.2507/31st.daaam.proceedings.078.

[7] D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto, "SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings," *J. Syst. Softw.*, vol. 168, p. 110671, 2020. doi:10.1016/j.jss.2020.110671.

[8] D. Nikolic, D. Stefanovic, D. Dakic, S. Sladojevic, and S. Ristic, "Analysis of the Tools for Static Code Analysis," *2021 20th Int. Symp. INFOTEH-JAHORINA, INFOTEH 2021 - Proc.*, no. March, pp. 17–19, 2021, doi:10.1109/infoteh51037.2021.9400688.

[9] J. Wang, Y. Huang, S. Wang, and Q. Wang, "Find Bugs in Static Bug Finders," *IEEE Int. Conf. Progr. Compr.*, vol. 2022-March, pp. 516–527, 2022, doi:10.1145/3377811.3380380.

[10] D. Marcilio, R. Bonifacio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," *IEEE Int. Conf. Progr. Compr.*, vol. 2019-May, pp. 209–219, 2019, doi:10.1109/ICPC.2019.00040.

[11] 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.

[12] L. Cruz and R. Abreu, "Performance-Based Guidelines for Energy Efficient Mobile Applications," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2017, pp. 46–57, doi:10.1109/MobileSoft.2017.19.

[13] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of Android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2017, pp. 115–126, doi:10.1109/saner.2017.7884614.

[14] D. Kwan Kim, "Towards Performance-Enhancing Programming for Android Application Development," *Int. J. Contents*, vol. 13, no. 4, pp. 39–46, 2017, doi:10.5392/IJoC.2017.13.4.039.

[15] A. Wibowo, A. R. Chrismanto, M. N. A. Rini, and L. Chrisantyo, "Mobile Application Performance Improvement with the Implementation of Code Refactor Based on Code Smells Identification: Dutataniku Agriculture Mobile App Case Study," in *2022 Seventh International Conference on Informatics and Computing (ICIC)*, Dec. 2022, pp. 1–7, doi: 10.1109/ICIC56845.2022.10006961.

[16] Q. Xu, J. C. Davis, Y. C. Hu, and A. Jindal, "An Empirical Study on the Impact of Deep Parameters on Mobile App Energy Usage," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2022, pp. 844–855, doi: 10.1109/SANER53432.2022.00103.

[17] R. Verdecchia, R. Aparicio Saez, G. Procaccianti, and P. Lago, "Empirical Evaluation of the Energy Impact of Refactoring Code Smells," 2018, pp. 365–345, doi: 10.29007/dz83.

[18] H. Anwar, D. Pfahl, and S. N. Srirama, "Evaluating the Impact of Code Smell Refactoring on the Energy Consumption of Android Applications," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2019, pp. 82–86, doi: 10.1109/SEAA.2019.00021.

[19] K. Etemadi Someoliayi *et al.*, "Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations," *IEEE Trans. Dependable Secur. Comput.*, pp. 1–17, 2022, doi:10.1109/TDSC.2022.3167316.

[20] A. Guerra-Manzanares and M. Välbe, "Cross-device behavioral consistency: Benchmarking and implications for effective android malware detection," *Mach. Learn. with Appl.*, vol. 9, p. 100357, Sep. 2022, doi: 10.1016/j.mlwa.2022.100357.

[21] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of Android-specific code smells: The aDoctor project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2017, pp. 487–491, doi: 10.1109/saner.2017.7884659.

[22] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," *ACM SIGPLAN Not.*, vol. 48, no. 10, pp. 623–639, 2013, doi: 10.1145/2544173.2509552.

[23] M. Nass, E. Alegroth, and R. Feldt, "Augmented testing: Industry feedback to shape a new testing technology," *Proc. - 2019 IEEE 12th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2019*, pp. 176–183, 2019, doi: 10.1109/ICSTW.2019.00048.

[24] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's Escape: Breaking the Resource Rein of Linux Control Groups," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, Nov. 2019, pp. 1073–1086, doi:10.1145/3319535.3354227.

[25] J. Lee, A. V. Raja, and D. Gao, "SplitSecond: Flexible Privilege Separation of Android Apps," in *2019 17th International Conference on Privacy, Security and Trust (PST)*, Aug. 2019, pp. 1–10, doi:10.1109/PST47121.2019.8949067.

[26] E. Wen, J. Cao, J. Shen, and X. Liu, "Fraus: Launching Cost-efficient and Scalable Mobile Click Fraud Has Never Been So Easy," in *2018 IEEE Conference on Communications and Network Security (CNS)*, May 2018, pp. 1–9, doi: 10.1109/CNS.2018.8433126.

[27] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study," *J. Syst. Softw.*, vol. 170, p. 110750, Dec. 2020, doi:10.1016/j.jss.2020.110750.

[28] P. H. De Andrade Gomes, R. E. Garcia, G. Spadon, D. M. Eler, C. Olivete, and R. C. M. Correia, "Teaching software quality via source code inspection tool," *Proc. - Front. Educ. Conf. FIE*, vol. 2017-Octob, pp. 1–8, 2017, doi: 10.1109/FIE.2017.8190658.

[29] D. Stefanović, D. Nikolić, S. Havzi, T. Lolić, and D. Dakić, "Identification of strategies over tools for static code analysis," *Identif. Strateg. over tools static code Anal.*, vol. 1163, no. 012012, pp. 1–9, 2021.

[30] I. Blasquez and H. Leblanc, "Experience in learning test-driven development: space invaders project-driven," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, Jul. 2018, pp. 111–116, doi: 10.1145/3197091.3197132.

[31] O. Hamdi, A. Ouni, E. A. AlOmar, M. O Cinneide, and M. W. Mkaouer, "An Empirical Study on the Impact of Refactoring on Quality Metrics in Android Applications," in *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, May 2021, pp. 28–39, doi:10.1109/MobileSoft52590.2021.00010.

[32] T. Oo, H. Liu, and B. Nyirongo, "Dynamic Ranking of Refactoring Menu Items for Integrated Development Environment," *IEEE Access*, vol. 6, pp. 76025–76035, 2018, doi: 10.1109/ACCESS.2018.2883769.

[33] N. Pombo and C. Martins, "Test driven development in action: Case study of a cross-platform web application," *EUROCON 2021 - 19th IEEE Int. Conf. Smart Technol. Proc.*, no. July, pp. 352–356, 2021, doi: 10.1109/eurocon52738.2021.9535554.

[34] S. Romano, F. Zampetti, M. T. Baldassarre, M. Di Penta, and G. Scanniello, "Do Static Analysis Tools Affect Software Quality when Using Test-driven Development?," *Int. Symp. Empir. Softw. Eng. Meas.*, pp. 80–91, 2022, doi:10.1145/3544902.3546233.