IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 5, 2021, accepted April 15, 2021, date of publication April 22, 2021, date of current version April 29, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3075040

An Empirical Investigation on the Effect of Code Smells on Resource Usage of Android Mobile Applications

MOHAMMAD A. ALKANDARI[1], ALI KELKAWI[1], AND MAHMOUD O. ELISH[2], (Senior Member, IEEE)
[1]Department of Computer Engineering, Kuwait University, Kuwait City 13060, Kuwait
[2]Department of Computer Science, Gulf University for Science and Technology, Hawally 32093, Kuwait
Corresponding author: Mohammad A. Alkandari (m.kandari@ku.edu.kw)

ABSTRACT Code smells refer to suboptimal coding practices which impact software quality and software non-functional requirements such as performance, maintainability, and resource usage. Although desktop application code smells have been extensively studied in the literature, mobile applications are relatively new in nature, and the effect of code smells is only recently being studied on mobile devices. This paper investigates the effect of code refactoring on enhancing both CPU usage and Memory usage. It presents a study of three code smells: HashMap Usage, Member Ignoring Method and Slow Loop, and eight open-source applications were selected from Github for testing purposes. The three aforementioned code smells were refactored individually and cumulatively to study their effects on a mobile phone's resource usage, with CPU usage and memory usage as the metrics of choice. The resource usage of five different versions of eight different mobile applications were measured to find the optimal refactoring strategy. The results obtained suggest that refactoring HashMap Usage and Member Ignoring Methods yielded significantly an average improvement in CPU usage of 12.7% and 13.7% respectively, while the refactoring of all three code smells yielded an improvement of up to 7.1% in memory usage. This research shows that certain refactoring methods have significant impacts on improving both the CPU usage and Memory usage. These statistically significant results can be used as the basis of guidelines to assist in writing codes which utilize smartphones' resources more efficiently and enhance their quality.

INDEX TERMS Code smells, Android, resource usage.

I. INTRODUCTION

Mobile applications have come to dominate the software industry over the past decade, with over 5 million applications available on the Apple App Store and Google Play Store combined. It is expected that the number of application downloads worldwide will exceed 250 billion downloads by the year of 2022 [1]. This exponential rise in numbers calls for closer evaluation of coding practices and their effects on the non-functional requirements of mobile applications such as performance, energy consumption, maintainability, and security.

Poor coding practices, also known as code smells [12], have been extensively studied in the literature when it comes to desktop applications, with papers addressing the correlation between different code smells and an application's maintainability [2] and its quality [3]. However, due to their recent growth, research has not been able to stay up to date with the large number of mobile applications being developed. Several articles have been published to address a small number of code smells' effect on a mobile application's performance [4], energy consumption [5], security [6] and resource usage [7]. However, there remain several combinations of code smells whose effects on mobile applications and their non-functional requirements which have not been explored.

Resource usage is an important factor for mobile applications due to the limited resources available. Therefore, identifying reasons that an application is hogging resources can pave the path for better coding practices to be used to promote the usage of less CPU and device memory. Although there have been several studies conducted on the effect of code smells on the energy consumption of Android

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

applications [8]–[10], [13], code smells related to resource usage have not been as extensively studied.

The objective of this paper is to empirically investigate the effect of some code smells on resource usage of Android mobile applications. It contributes statistically significant results and empirical insights that can be used as the basis of guidelines to assist in writing code which utilizes a smartphone's resources more efficiently.

The rest of the paper is organized as follows: Section II discusses background related to code smells and performance metrics. Section III reviews related works. Section IV discusses the research methodology and the empirical study conducted to investigate the effects of code smells on a mobile application's memory and CPU usage. Section V discusses the evaluation and the results of the conducted empirical study. Finally, Section VI concludes the paper and provides directions for future work.

II. BACKGROUND

This section describes the code smells which have been studied and investigated further in this paper. It also discusses the metrics typically used to measure an Android application's resource usage.

A. CODE SMELLS

As discussed previously, code smells are considered poor coding practices, and have been a topic of research in the world of software due to the importance of optimizing the performance of mobile applications by improving energy consumption, resource usage and UI performance to name a few. Initially studied in relation to desktop applications, mobile specific code smells have been introduced and further studied [11]. Although code smells on both platforms have studied bad coding practices in object oriented programming in desktop applications, mobile code smells have come to light due to the negative impact they may have on a smartphone's limited resources and energy.

Of the Android-specific code smells discussed in [14], three are selected to study their relations to a mobile application's resource usage due to: (i) their commonality in Android applications as discussed in [15]; (ii) their simplicity to be detected and to be easily refactored; (iii) their commonality in a programming environment; and (iv) their effects not being investigated with respect to resource usage.

- **Member Ignoring Method (MIM)**: This refers to the mis-classification of a method (static or dynamic). It is recommended to have a static method when feasible due to the increase in speed from dynamic to static methods of 15-20%. Figure 1 shows this code smell before and after refactoring.
- **Slow Loop (SL)**: This refers to the standard for loop being slower than the for-each loop. Therefore, it is suggested to replace the former with the latter whenever possible to improve the efficiency of the application. Figure 2 shows this code smell before and after refactoring.

```
private String
stringOrBlank(final String s) {
  return s != null ? s : "";
}
```

(a) Before refactoring

```
private static String
stringOrBlank(final String s) {
  return s != null ? s : "";
}
```

(b) After refactoring

FIGURE 1. Member Ignoring Method refactoring strategy.

```
for (int p = 0; p != points.size(); ++p) {
  int distance =
GeoHelper.distanceBetween(points.get(p), (location));
  if (distance > minDistance)
    continue;

  minDistance = distance;
  minIndex = p;
}
```

(a) Before refactoring

```
int p = 0;
for(IGeoPoint point : points) {
  int distance = GeoHelper.distanceBetween(point,
(location));
  if (distance > minDistance)
    continue;

  minDistance = distance;
  minIndex = p;

  p++;
}
```

(b) After refactoring

FIGURE 2. Slow Loop refactoring strategy.

```
private final Map<Integer, String>
purpDescriptions = new HashMap<>();
```

(a) Before refactoring

```
private final Map<Integer, String>
purpDescriptions = new ArrayMap<>();
```

(b) After refactoring

FIGURE 3. HashMap Usage refactoring strategy.

- **HashMap Usage (HMU)**: As the name implies, this refers to the usage of HashMaps in an Android application. Theoretically, the Android framework prefers the use of ArrayMap and SimpleArrayMap due to their better memory-efficiency. Figure 3 shows this code smell before and after refactoring.

B. RESOURCE USAGE METRICS

High resource usage in a mobile application inevitably leads to poor performance and decreased battery life, thus it is

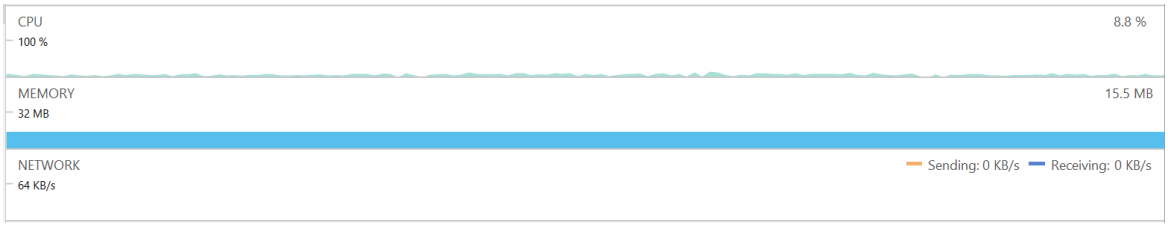


FIGURE 4. Android Studio profiler.

imperative to find ways of optimizing resource usage to ensure that the user experience is improved. For the purpose of this study, resource usage will refer to CPU and memory usage. Their relevance is as follows:

- **CPU Usage**: This is a measure of how much work is being performed by the smart phone's processor. Applications with high CPU usage are a cause for concern, as it has a negative effect on battery life and device speed.
- **Memory Usage**: This is a measure of how much memory is being consumed by a mobile application. Although the sizes of Random Access Memory (RAM) in smart phones is continuously increasing with each iteration of new releases, it is still important to try to optimize the amount of memory consumed.

In evaluating the aforementioned resource usage metrics, Android Studio was used as the tool of choice, as it offers an extensive profiler tool which visually shows CPU and memory usage at any given time as shown in Figure 4. Further, it offers seamless integration with open-source Android applications, the majority of which are developed using this tool.

III. RELATED WORKS

Hecht *et al.* [4] conduct an empirical study on the effects of code smells on the performance of Android mobile applications in terms of memory usage and UI performance. The study discusses the individual and combined effects of the three following code smells on two different applications: Internal Getter/Setter, Member Ignoring Method, and HashMap Usage. The authors test the applications using two metrics for UI performance: frame time and number of delayed frames, which pertain to the time taken for an Android application to draw one frame on the screen. A delay in the drawing of these frames would lead to a reduction in User Experience. The paper also discusses two metrics for memory efficiency: memory usage and number of garbage collection calls, which pertain to how well an application manages its memory usage and the resources allocated to it. Using PAPRIKA Toolkit to detect the code smells and a process of manual refactoring of them both individually and combined, the authors found an overall improvement of 12.6% in UI performance by refactoring Member Ignoring Methods and up to 3.6% improvement in memory performance with all three code smells refactored.

Pritam *et al.* [21] Another research study, that was published recently in this area, addressed code smells with respect to software quality. This study targeted the assessment of code smell for predicting class change proneness and discover errors using machine learning algorithms.

Morales *et al.* [17] study the effects of anti-patterns (code smells) on the energy efficiency of 20 Android applications downloaded from F-Droid. Although the paper looks at 8 different code smells, only the three Android-specific ones are considered for this paper, which are Binding Resources too early, HashMap Usage and Private getters and setters. Binding Resources too early refers to the initialization of energy consuming resources such as GPS or Wi-Fi before they need to be used. The authors use several tools to conduct this case study: ReCon for the detection of code smells, HiroMacro for the generation of usage scenarios, Monkeyrunner to automatically run the generated usage scenarios, Android Studio to refactor the code smells, and the digital oscilloscope TiePie Handyscope HS5 to measure the application's energy consumption. The results obtained during the study showed that although the improvement in energy efficiency by refactoring code smells depends on the context in which the application runs, applications with code smells consumed more energy than their refactored counterparts.

Carette *et al.* [9] discuss the effects of the three aforementioned code smells on the energy consumption of five open-source Android applications. The authors use a combination of two tools, PAPRIKA and NAGA VIPER, dubbing it as the HOT-PEPPER approach. In this paper, the PAPRIKA toolkit is slightly modified to detect as well as refactor the code smells. NAGA VIPER is used to measure and compare the energy metrics of each Android application, by feeding it both the original Android Package (APK) as well as the refactored one. Based on the results obtained, the most significant improvement in energy consumption was found to be 4.83%, showing that more research needs to be invested in the reduction of code smells to improve application performance.

Cruz and Abreu [18] explore the effects of code smells on an Android application's energy efficiency. The authors attempt to outline performance-based development guidelines to ensure that applications consume as little energy as possible. The paper conducts tests on 6 open-source Android applications, observing 8 different code smells (some of which are mentioned in Table 4) using Lint, a tool provided by the Android Software Development Kit (SDK) to

TABLE 1. Summary of related works.

Study	Year	Source of Systems	# of Android Apps Tested	Tool(s) Used	Code Smells	Non-Functional Requirements	Metrics
Hecht et al. [4]	2016	Github	2	Paprika Toolkit	1. Internal Getter/Setter 2. Member Ignoring Method 3. HashMap Usage	Performance	Frame Time Number of delayed frames Memory Usage Number of garbage collection calls
Morales et al. [17]	2016	F-Droid	20	ReCon	1. Binding Resources too early 2. HashMap Usage 3. Private getters & setters	Energy Consumption	Current Voltage Time
Carette et al. [9]	2017	F-Droid Github SourceForge	5	Paprika Toolkit Naga Viper	1. Internal Getter/Setter 2. Member Ignoring Method 3. HashMap Usage	Energy Consumption	Average Intensity Average Voltage
Cruz et al. [18]	2017	F-Droid	6	lint	1. ViewHolder 2. DrawAllocation 3. WakeLock and 5 others	Maintainability	File changes
Oliveira et al. [7]	2018	Github	9	JDeodorant	1. God Class 2. God Method 3. Feature Envy	Resource Usage	CPU Usage Memory Usage
Palomba et al. [19]	2019	F-Droid	60	aDoctor Petra	1. Inefficient Data Structure 2. Internal Setter 3. Member Ignoring Method and 6 others	Energy Consumption	Consumed Energy Current Intensity Device Voltage
This Study	2020	Github	8	aDoctor	1. Member Ignoring Method 2. Slow Loop 3. HashMap Usage	Resource Usage	CPU Usage Memory Usage

detect problems with code structure. It was found that applications with the code smells ViewHolder, DrawAllocation, WakeLock, ObsoleteLayoutParam and Recycle are more likely to have improvements in terms of energy efficiency. Through the studies conducted, the authors also found that the mobile device's battery life could last up to one hour longer by using energy-aware coding practices.

Oliveira *et al.* [7] also assess the effects of an Android application's resource usage in terms of CPU and memory with three different code smells: God Class, God Method and Feature Envy. God Class refers to a class which has too many methods and attributes and does not make use of all of them. God Method describes a method that is long and handles more than one task. Feature Envy refers to a method which uses more attributes from another class than its own class. The authors use JDeodorant as the tool of choice due to its ability to detect and refactor these code smells. The results of this empirical study showed that refactoring of Desktop code smells on Android applications may not necessarily yield better results in terms of CPU usage, with the refactoring of God Method in one application increasing CPU consumption by 47%, while the refactoring of all three code smells reduced memory consumption by up to 8.4%.

Palomba *et al.* [19] study the effects of nine Android-specific code smells (some of which are mentioned in Table 4) on the energy consumption of 60 Android applications. These smells are detected and refactored by aDoctor, a tool which was developed by the authors. The energy consumption of the Android applications was tested by PETRA, a software-based tool which estimates the energy profile of mobile apps. This empirical study concluded that four code smells (Internal Setter, Leaking Thread, Member Ignoring Method, and Slow Loop) consume up to 87 times more energy than methods affected by other code smells.

Table 1 shows a summary of these papers, the code smells studied and the metrics used. As demonstrated in the table, Oliveira *et al.* [7] research and our study have some similarities using the same Metrics (CPU usage and Memory usage) as well as the same non-functional requirements (Resource Usage). However, our study investigates completely different code smells which are: Member Ignoring Method, Slow Loop, and HashMap Usage.

IV. RESEARCH METHODOLOGY AND STUDY DESIGN

This section describes the research methodology and the research questions this study aims to answer, and proceeds to describe the process and tools used to conduct the study.

A. RESEARCH METHODOLOGY

The research methodology of this study starts with understanding the literature behind code smells and their effects on software non-functional requirements such as performance, maintainability, and resource usage. Then, examine and choose a sufficient number of existing Android Applications from various categories with at least 3 counts of the code smells and at least 10 classes in order to ensure that each application has different functionalities which can be tested. After that, analyze all code smells using a particular tool that has both a Graphical User Interface, and command-line support which help detect the code smells. Then, manually refactor all detected code smells with respect to Member Ignoring Method, Slow Loop, and HashMap Usage. After that, manually run some test scenarios generated particularly to test the different functionalities of each application and

collect the average CPU and Memory usage for each run. Then, apply statistical tests and analyze the collected data, and draw conclusions from the obtained results. More information and specific details regarding the research methodology and procedure will be provided in the next subsections.

B. RESEARCH QUESTIONS

The purpose of this study is to discover whether refactoring certain code smells can improve the resource usage capabilities of a mobile application. Based on this goal, we first collect a corpus of open-source Android applications. The applications are tested for the aforementioned code smells using aDoctor [14]. Each application is then tested based on a generated test scenario on the original application as well as when the code smells are refactored both individually and cumulatively. The results collected are then analyzed statistically to extract an informative conclusion. The following research questions are formulated to guide the study forward:

- **RQ1**: Does refactoring specific code smells reduce a mobile application's CPU usage?
- **RQ2**: Does refactoring specific code smells reduce a mobile application's memory usage?

C. CORPUS

A total of eight Android applications from six different categories were used in this study. These applications were selected after inspecting several open-source projects available on Github, with a focus on applications with at least 3 counts of the code smells being studied and at least 10 classes, ensuring that the application has different functionalities which can be tested. It is worth noting that all applications are written using Java language. Despite the recent popularity of Kotlin language in creating Android mobile applications, code smells of Android applications written in Java have been more thoroughly defined and there is a larger corpus of open-source Android written in Java available for testing. The applications selected were: *CycleStreets,*[1] *Loop Habit Tracker,*[2] *Travel Mate,*[3] *NSIT-Connect,*[4] *GNUCash,*[5] *OmniNotes,*[6] *EasyXKCD*[7] and *Memory Game.*[8]

D. PROCEDURE

1) CODE SMELL DETECTION

Once the open-source code of the application is downloaded, aDoctor [14] is used as the tool for detecting Android code smells. This tool was developed to detect 15 Android-specific code smells, and offers both a Graphical User Interface and command-line support to analyze an application's source code. Upon testing, it was found that aDoctor failed to detect HashMap Usage code smell in several projects. Therefore, this code smell was detected through a simple search using Android Studio's utilities.

2) CODE SMELL REFACTORING

While there are several custom tools that have been developed in the literature for automatic refactoring, we adopt Palomba *et al.* [19] approach in using manual refactoring. This is largely due to the simple nature of the code smells being studied, therefore manual refactoring is not seen to be an expensive process, while also making sure that each refactoring detected is one that is correct and necessary for refactoring. The code smells detected are manually refactored according to the following:

- **Member Ignoring Method:** Non-static methods are converted to static methods if they have no need to call instance variables in a given class.
- **Slow Loop:** For loops are converted to for each loops. In some instances, the usage of for each loops is not possible as the loop is meant to iterate over an object which does not implement the Iterable interface.
- **HashMap Usage:** HashMaps are replaced with ArrayMaps where applicable. In some instances, the conversion of HashMap to ArrayMap causes compatibility errors in the program.

3) COLLECTION OF RESULTS

Similar to the refactoring process, each application is manually run on the test scenarios outlined in Table 2, generated to test the different functionalities of a given application. The choice of manual testing is made to ensure that the CPU and memory usage results can be collected without the interference of third-party automation tools. Five individual runs are made and the average CPU and memory usage for each run is collected using the Android Studio profiler. Although the testing process is manual, careful care is taken to ensure that each of the five runs is identical in steps and execution of the test scenario. The applications are first loaded using Android Studio, and tested on the test scenarios on an emulated version of the Google Pixel 2 device running Android Oreo operating system. It has a 1.9 GHz octa-core Qualcomm Snapdragon 835 processor and 4 GB RAM. This version of the application will be referred to as $\mathbf{V}_{orig}$. Four more versions of each application were generated:

- $\mathbf{V}_{HMU}$ - This version has the HashMap Usage code smell refactored.
- $\mathbf{V}_{MIM}$ - This version has the Member Ignoring Methods code smell refactored.
- $\mathbf{V}_{SL}$ - This version has the Slow Loop code smell refactored.
- $\mathbf{V}_{ALL}$ - This version has all three code smells refactored.

Each of the five versions were manually run five separate times on the same user test scenarios which cover the applications' functionalities, and the average readings of CPU and memory usage were aggregated and tabulated for comparison.

[1] https://github.com/cyclestreets/android
[2] https://github.com/iSoron/uhabits
[3] https://github.com/project-travel-mate/Travel-Mate
[4] https://github.com/NSITonline/NSIT-Connect
[5] https://github.com/codinguser/gnucash-android
[6] https://github.com/federicoiosue/omni-notes
[7] https://github.com/tom-anders/easy_xkcd
[8] https://github.com/sromku/memory-game

TABLE 2. Description and duration of testing scenarios.

App	Category	Scenario(s)	Approximate Duration (sec)
CycleStreets	Health & Fitness	1. Browse map and plan route 2. Visit itinerary and scroll to the bottom 3. Visit photomap and browse two different photos	60
Loop Habit Tracker	Productivity	1. Register new habit 2. Mark habit as done for past 3 days 3. View habit overview page 4. Delete habit	40
Travel Mate	Travel	1. View list of destinations 2. Choose destination 3. View fun facts, monuments, restaurants and shopping places	80
NSIT-Connect	Education	1. View document on homepage 2. Open locations tab and view Cafes 3. Open coderadar tab and view first upcoming competition 4. Open Professors tab and view first professor's information	65
GNUCash	Finance	1. Create test account 2. Add charge to test account 3. Add expense to expenses account 4. Add charge to expenses account 5. Generate pie chart for expenses account 6. Generate bar chart for expenses account 7. Delete test account 8. Delete all charges created steps 3 and 4	100
OmniNotes	Productivity	1. Create one regular note 2. Create one note with checklist with two items 3. Sort notes by creation date 4. Delete first note 5. Mark both tasks as done on second note 6. Delete second note 7. Remove both notes from trash permanently	80
EasyXKCD	Entertainment	1. Browse five recent comics 2. Go to "What if?" tab 3. Click on two most recent articles and add them to favorites 4. Go to "Favorites" tab and remove articles 5. Browse random article using randomization feature	110
Memory Game	Entertainment	1. Play Animals memory game on easy mode	60

V. EMPIRICAL STUDY RESULTS

This section discusses the results obtained from the conducted empirical study, and answers the research questions.

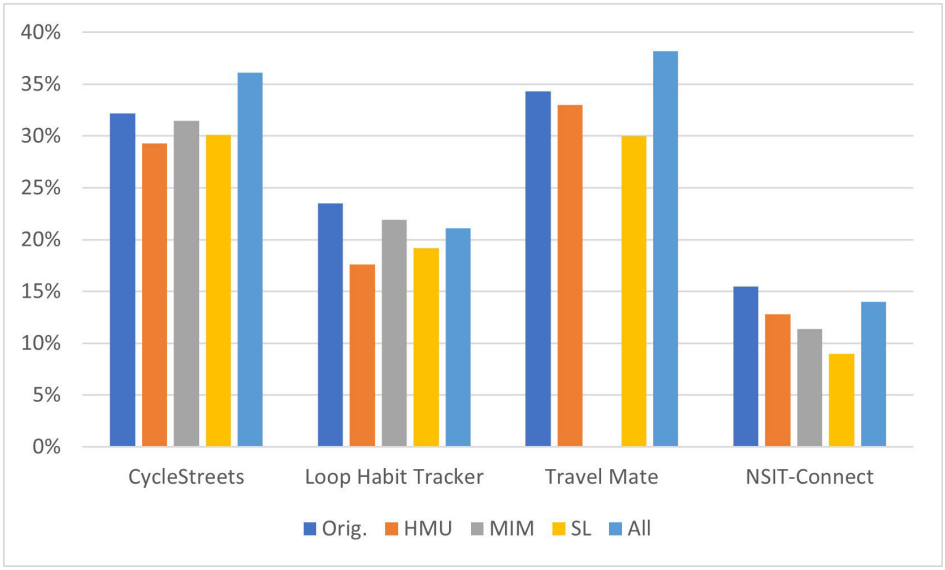
A. OVERVIEW

The results collected for average CPU usage and average memory usage are shown in Figure 5 and Figure 6 respectively. As discussed previously, a given Android application is tested after each individual code smell is refactored, and after all three code smells are refactored in combination. This approach is used in previous works [4], [9] and helps distinguish and focus on the effect of refactoring individual code smells as well as on the effect of refactoring multiple code smells at once.

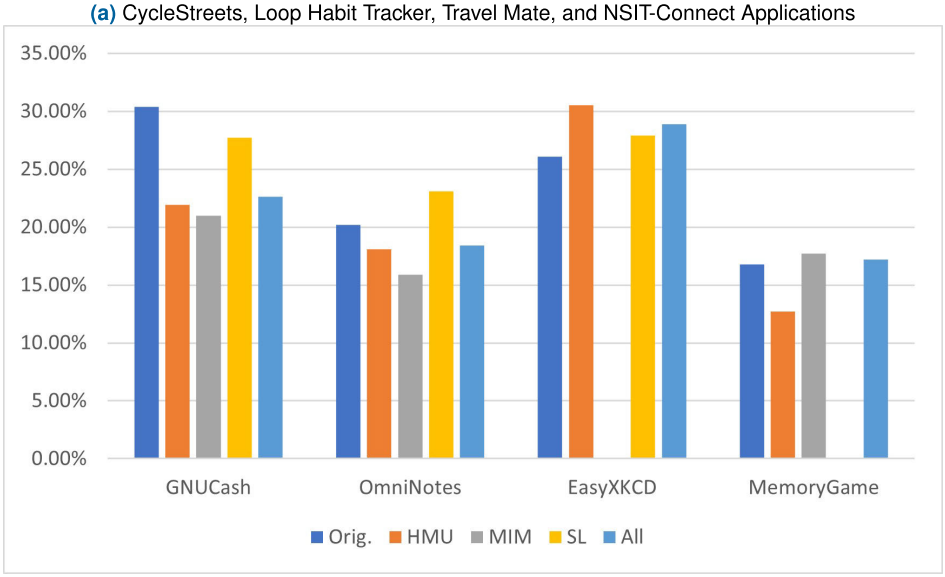
In looking at the initial results in Figure 5, we can see that the refactoring of indiviual code smells HashMap Usage, Slow Loop, Memory Ignoring Method often results in lower average CPU usage. However, the combined refactoring of these code smells may result in an increased average CPU usage in some cases. For memory usage in Figure 6, it is easier to see that the refactoring of individual code smells as well

TABLE 3. Number of code smells corrected.

	HMU	*MIM*	*SL*	*ALL*
CycleStreets	8	5	4	17
Loop Habit Tracker	4	1	3	8
Travel Mate	4	0	9	13
NSIT-Connect	5	2	3	10
GNUCash	14	3	2	19
OmniNotes	15	14	3	32
EasyXKCD	4	0	7	11
MemoryGame	3	7	0	10



(a) CycleStreets, Loop Habit Tracker, Travel Mate, and NSIT-Connect Applications



(b) GNUCash, OmniNotes, EasyXKCD, and MemoryGame Applications

FIGURE 5. CPU usage of all versions.

as the refactoring of all three code smells collectively almost always results in decreased average memory usage. These results are tabulated and will be further analyzed with respect to the number of code smells refactored shown in Table 3.

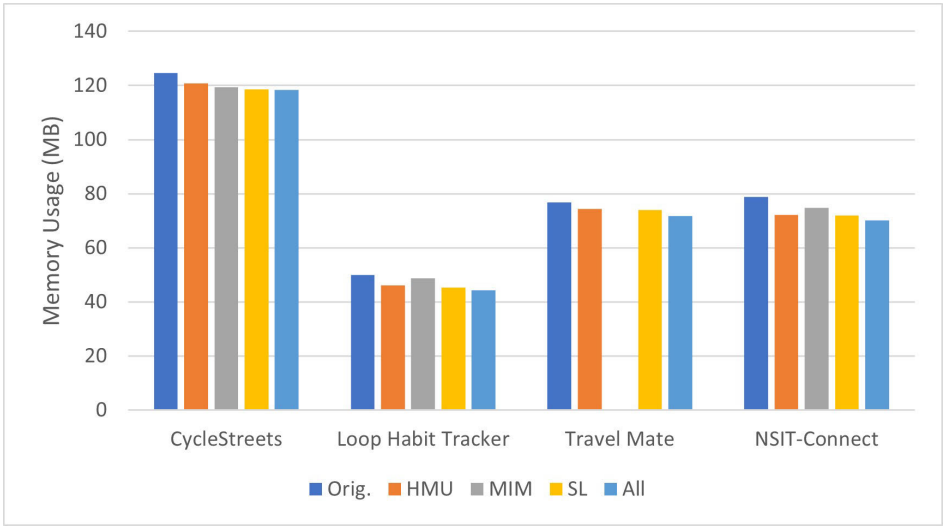
B. RESULTS AND DISCUSSION

As mentioned earlier, the objective of this study is to investigate the effect of code refactoring on CPU usage and Memory usage. Therefore, this section analyzes the data and answers the main questions of this research:

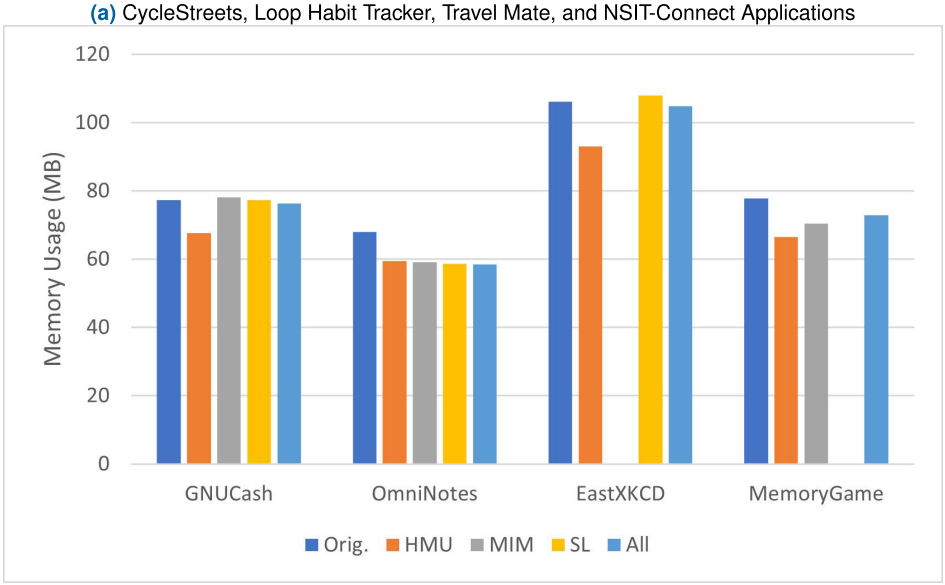
1) RQ1: DOES REFACTORING SPECIFIC CODE SMELLS REDUCE A MOBILE APPLICATION's CPU USAGE?

Figure 5 shows the CPU usage of the original applications along with the four derived versions of the same application. It is worth noting that only the application's CPU usage is taken into consideration, and not that of the smartphone's background usage as well. Although the same usage scenario is used across all versions, CPU usage readings often differ everytime the application is run. Therefore, Table 4 is used to show the average CPU usage for each version of the applications and the differences with $\mathbf{V}_{orig.}$ for comparison purposes.

In Table 4, we present the average and standard deviation of the CPU consumption of the five versions of each application, as well as the improvement, with a negative figure indicating that the refactoring results in an improvement in performance, and a positive figure indicating that the refactoring results in a worse performance. For the refactoring of HashMap Usage, seven of the eight applications tested showed an improvement in CPU performance, with the average improvement being



(a) CycleStreets, Loop Habit Tracker, Travel Mate, and NSIT-Connect Applications



(b) GNUCash, OmniNotes, EasyXKCD, and MemoryGame Applications

FIGURE 6. Memory usage of all versions.

12.65%. For the refactoring of Member Ignoring Method, five of the six applications tested resulted in an improvement in CPU performance, with the average improvement being 13.71%. In refactoring the Slow Loop code smell, five of the seven applications tested resulted in an improvement in CPU performance, with an average improvement of 9.56%. Finally, refactoring the three code smells resulted in an improvement in CPU performance in only four of the eight applications tested. However, the average improvement is 2.23%. This result indicates that code smells are not independent in improving or worsening the performance of Android applications and refactoring multiple code smells does not necessarily translate to an improvement in performance, and may even negatively impact the application's performance.

In Table 5, we apply the Wilcoxon signed-rank statistical test to the CPU usage results to analyze if refactoring of code smells has a statistically significant impact on Android applications. Results with p-value of less than 0.1 show 90% confidence level that the impact is significant. As shown in the results, the refactoring of HashMap Usage and Member Ignoring Method code smells individually results in a significant improvement in average CPU usage, which is reflected in the average improvements of 12.65% and 13.71% respectively. From these results, we can conclude that the refactoring of HMU and MIM code smells can have a statistically significant impact on the CPU usage of Android applications. Further, the data illustrates that HashMap Usage with a p-value of 0.093 or Member Ignoring Method with a p-value of 0.075 is sufficient to obtain statistically significant

TABLE 4. Percentage differences of average CPU usage.

	CycleStreets			*Loop Habit Tracker*			*Travel Mate*			*NSIT-Connect*		
	Average	**Std. Deviation**	**Improvement**	**Average**	**Std. Deviation**	**Improvement**	**Average**	**Std. Deviation**	**Improvement**	**Average**	**Std. Deviation**	**Improvement**
$\mathbf{V}_{orig}$	32.2%	31.1%	0%	23.5%	23.8%	0%	34.3%	21.4%	0%	15.5%	21.6%	0%
$\mathbf{V}_{HMU}$	29.3%	27.2%	-9.0%	17.6%	21.8%	-25.1%	33.0%	22.7%	-3.8%	12.8%	20.3%	-17.8%
$\mathbf{V}_{MIM}$	31.5%	22.9%	-2.2%	21.9%	26.4%	-6.8%	–	–	–	11.4%	15.9%	-26.5%
$\mathbf{V}_{SL}$	30.1%	28.8%	-6.5%	19.2%	24.1%	-18.3%	30.0%	22.4%	-12.5%	9.0%	20.1%	-41.9%
$\mathbf{V}_{ALL}$	36.1%	24.9%	-12.1%	21.1%	19.6%	-10.2%	38.2%	14.9%	+11.4%	14.0%	20.1%	-9.7%
	GNUCash			*OmniNotes*			*EasyXKCD*			*Memory Game*		
	Average	**Std. Deviation**	**Improvement**	**Average**	**Std. Deviation**	**Improvement**	**Average**	**Std. Deviation**	**Improvement**	**Average**	**Std. Deviation**	**Improvement**
$\mathbf{V}_{orig}$	30.4%	24.9%	0%	20.2%	22.9%	0%	26.1%	20.9%	0%	16.8%	14.7%	0%
$\mathbf{V}_{HMU}$	21.9%	22.1%	-28.0%	18.1%	17.5%	-10.4%	30.5%	18.5%	+16.9%	12.7%	9.2%	-24.4%
$\mathbf{V}_{MIM}$	21.0%	15.3%	-30.9%	15.9%	17.9%	-21.3%	–	–	–	17.7%	17.4%	+5.4%
$\mathbf{V}_{SL}$	27.7%	21.1%	-8.9%	23.1%	23.0%	+14.4%	27.9%	18.7%	+6.9%	–	–	–
$\mathbf{V}_{ALL}$	22.6%	21.4%	-25.7%	18.4%	20.3%	-8.9%	28.9%	17.4%	+10.7%	17.2%	18.5%	+2.38%

TABLE 5. Wilcoxon statistical test results for CPU usage.

	Orig.	**HMU**	**MIM**	**SL**
HMU	**0.093**	-	-	-
MIM	**0.075**	0.400	-	-
SL	0.128	0.735	0.686	-
All	0.889	**0.050**	0.116	0.310

results. This means that applying only one of the refactoring methods is enough to enhance the CPU usage and improve its quality. Thus, no need to apply all refactoring methods because their impacts will not be significant.

2) RQ2: DOES REFACTORING SPECIFIC CODE SMELLS REDUCE A MOBILE APPLICATION's MEMORY USAGE?

Figure 6 shows the memory usage of the original applications along with the four derived versions of the same application. Table 6 is used to show the average memory usage for each version of the application and the differences with $\mathbf{V}_{orig}$. for comparison purposes.

In Table 6, we present the average and standard deviation of the memory consumption of the five verions of each application, as well as the improvement. Although the improvement in CPU results may vary, it is easier to see that refactoring the code smells being studied almost always results in improved results of average memory usage. For refactoring of HashMap Usage, all eight applications resulted in improved memory usage, with an average improvement of 9.31%. For the refactoring of Member Ignoring Method, five of the six applications tested resulted in improved memory usage, with an average improvement of 5.57%. In refactoring the Slow Loop code smell, five of the seven applications tested resulted in an improvement in memory usage, with an average improvement of 5.54%. Finally, refactoring the three code smells resulted in an improvement in memory usage in all eight applications, with an average improvement of 7.13%.

In Table 7, we apply the same statistical test to the memory usage results to study the impact of code smell refactoring on Android applications. As shown in the results, the refactoring of the three code smells both individually and cumulatively produces significant improvement in average memory usage. From these results, we can conclude that the refactoring of HMU, MIM and SL code smells as well as all three code smells together can result in improved memory usage. Moreover, the data shows that one refactoring method is sufficient to obtain statistically significant results. This means that applying only one of the refactoring methods is enough to enhance the memory usage and improve its quality. Thus, there is no need to apply all refactoring methods because their impacts will not be significant. The data also demonstrates that the HashMap Usage or applying all refactoring methods investigated in this study would have a slightly better improvements in the memory usage when compared to applying only Member Ignoring Method or Slow Loop Method.

C. THREATS TO VALIDITY

In this section, we discuss some of the threats to the validity of this research as outlined in [16].

Construct validity is concerned with the effectiveness of tests made to capture the required results. In this study, we were careful in running the test scenario five times to collect average readings of CPU and memory usage, therefore reducing any error margins caused by collecting a single set of results. However, false positives identified by the tool of choice meant that each code smell recognized must be studied before being considered as a true positive and refactored accordingly. Manual refactoring of code smells has been also considered, whereby we ensured that refactored code segments were related to the observed code smell only. All tests were conducted on an Android emulator to eliminate any side effects such as the effect of other applications on battery usage and background CPU/Memory usage that may arise from testing on a physical device. However, it is worth noting that there may be some performance differences between the

TABLE 6. Percentage differences of average memory usage.

	CycleStreets			*Loop Habit Tracker*			*Travel Mate*			*NSIT-Connect*		
	Average (MB)	**Std. Deviation**	**Improvement**	**Average (MB)**	**Std. Deviation**	**Improvement**	**Average (MB)**	**Std. Deviation**	**Improvement**	**Average (MB)**	**Std. Deviation**	**Improvement**
V_{orig}	124.7	87.1	0%	49.9	4.0	0%	76.8	8.5	0%	78.9	52.5	0%
V_{HMU}	120.7	84.0	-3.2%	46.1	4.0	7.6%	74.3	6.5	-3.3%	72.2	37.4	-8.5%
V_{MIM}	119.4	81.2	-4.3%	48.7	3.6	-2.4%	–	–	–	74.8	49.3	-5.2%
V_{SL}	118.5	80.4	-5.0%	45.4	3.9	-9.0%	73.9	6.4	-3.8%	71.9	43.3	-8.9%
V_{ALL}	118.3	81.2	-5.1%	44.3	4.5	-11.2%	71.7	7.9	-6.6%	70.2	43.7	-11.0%
	GNUCash			*OmniNotes*			*EasyXKCD*			*Memory Game*		
	Average (MB)	**Std. Deviation**	**Improvement**	**Average (MB)**	**Std. Deviation**	**Improvement**	**Average (MB)**	**Std. Deviation**	**Improvement**	**Average (MB)**	**Std. Deviation**	**Improvement**
V_{orig}	77.3	49.7	0	68.0	58.8	0	106.2	35.7	0	77.8	40.7	0
V_{HMU}	67.7	44.7	-12.4%	59.4	44.5	-12.7%	93.1	34.5	-12.3%	66.5	38.8	-14.5%
V_{MIM}	78.1	46.9	+1.0%	59.1	40.2	-13.1%	–	–	–	70.4	37.8	-9.5%
V_{SL}	77.3	46.6	0%	58.6	39.4	-13.8%	108.0	39.8	+1.7%	–	–	–
V_{ALL}	76.3	50.8	-1.3%	58.4	42.1	-14.1%	104.8	32.8	-1.3%	72.9	37.5	-6.3%

TABLE 7. Wilcoxon statistical test results for memory usage.

	Orig.	**HMU**	**MIM**	**SL**
HMU	**0.012**	-	-	-
MIM	**0.046**	0.115	-	-
SL	**0.046**	0.866	**0.043**	-
All	**0.012**	0.674	0.173	**0.018**

emulator and the physical device that may be worth studying in future works.

Internal validity is concerned with the causal relationship between treatment and effect. In this study, we were careful in isolating each derived version of the application to ensure that the outcomes displayed were due to the refactoring of one specific code smell.

External validity is concerned with our ability to generalize the results obtained. Further studies need to be conducted to study the effects of refactoring the aforementioned code smells on a smartphone's resource usage. Therefore, the results obtained cannot be generalized to other applications as of yet. Although the study was conducted on eight mobile applications, we believe this paper can contribute to the ongoing studies investigating the impacts of various code smells on a mobile application's non-functional requirements.

Finally, *Reliability validity* is concerned with the replicability of a study. All technologies used in this paper are publicly and freely available, with the test procedure outlined in Section IV, making the replication of this study possible.

VI. CONCLUSION

The empirical study conducted in this paper attempted to address the effects of three common code smells on the CPU and memory usage of smartphone applications: HashMap Usage, Member Ignoring Method and Slow Loop. Five different versions of the selected mobile applications were tested, with the code smells refactored individually and cumulatively for comparison with the original implementation.

The results obtained statistically demonstrate that refactoring certain code smells, such as Member Ignoring Method, can have a positive impact on a mobile application's CPU usage, while refactoring other code smells such as HashMap Usage can have a positive impact on a mobile application's memory usage. The results obtained suggest that refactoring HashMap Usage and Member Ignoring Methods yielded an average improvement in CPU usage of 12.7% and 13.7% respectively, while the refactoring of all three code smells yielded an average improvement of 7.1% in memory usage. These statistically significant results emphasise the importance of code refactoring with respect to software non-functional requirements such as performance, maintainability, and resource usage.

This research also illustrate that it is not necessary to apply all code refactoring methods used in this study to improve the quality of the CPU usage and memory usage. Sometimes, only one refactoring method might be sufficient to enhance the memory and CPU usage. Therefore, there is no need to spend more effort on applying other refactoring methods. This would definitely save time, effort, and cost. Further, the impacts of some refactoring methods on the CPU usage are different than their impacts on the Memory usage. In other words, the HashMap usage or Member Ignoring Method show significant impacts on improving the CPU usage, while applying all methods (HMU, MIM, SL, ALL) or each single method by itself show significant impacts on improving the Memory usage.

We believe the results of this study can be used as a basis for future implementation guidelines to assist developers in writing code which efficiently utilizes a smartphone's resources. In the future, we plan to extend this study by exploring the effects of other code smells on a smartphone's CPU and memory usage, as well as automating the process of collecting

and testing applications to increase the scale of future studies conducted. Moreover, there may be trade-offs between specific code smells and the non-functional requirements of a mobile application, which can be studied in more detail in a separate study. In addition, more research questions will be formulated in the near future to address the impacts of code refactoring not only on CPU and Memory usage, but also from the management point of view on time, effort, and cost. In addition, more studies will be conducted in the future to study the effect of code smells in applications developed by languages other than Java.

REFERENCES

[1] I. Blair, *Mobile App Download and Usage Statistics, Dosegljivo*. Accessed: Nov. 2019. [Online]. Available: https://buildre.com/app-statistics

[2] D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[3] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 260–269.

[4] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of Android code smells," in *Proc. Int. Conf. Mobile Softw. Eng. Syst.*, May 2016, pp. 59–69.

[5] A. Banerjee and A. Roychoudhury, "Automated re-factoring of Android apps to enhance energy-efficiency," in *Proc. Int. Conf. Mobile Softw. Eng. Syst.*, May 2016, pp. 139–150.

[6] M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in Android," in *Proc. IEEE 17th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2017, pp. 121–130.

[7] 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," in *Proc. 30th Int. Conf. Softw. Eng. Knowl. Eng.*, Jul. 2018, pp. 313–314.

[8] M. Gottschalk, J. Jelschen, and A. Winter, "Saving energy on mobile devices by refactoring," in *Proc. EnviroInfo*, 2014, pp. 437–444.

[9] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of Android smells," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 115–126.

[10] L. Cruz and R. Abreu, "Using automatic refactoring to improve energy efficiency of Android apps," 2018, *arXiv:1803.05889*. [Online]. Available: http://arxiv.org/abs/1803.05889

[11] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in Android applications," in *Proc. Int. Conf. Mobile Softw. Eng. Syst.*, May 2016, pp. 225–236.

[12] M. Fowler, *Refactoring: Improving the Design of Existing Code*. London, U.K.: Pearson, 1999.

[13] S. Habchi, N. Moha, and R. Rouvoy, "Android code smells: From introduction to refactoring," *J. Syst. Softw.*, vol. 177, Jul. 2021, Art. no. 110964.

[14] 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.

[15] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of Android applications along their evolution (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 236–247.

[16] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer-Verlag, 2012.

[17] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Anti-patterns and the energy efficiency of Android applications," 2016, *arXiv:1610.05711*. [Online]. Available: http://arxiv.org/abs/1610.05711

[18] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *Proc. IEEE/ACM 4th Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*, May 2017, pp. 46–57.

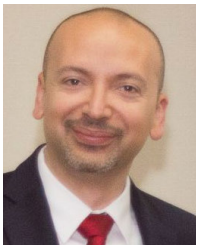
[19] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Inf. Softw. Technol.*, vol. 105, pp. 43–55, Jan. 2019.

[20] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "EARMO: An energy-aware refactoring approach for mobile apps," *IEEE Trans. Softw. Eng.*, vol. 44, no. 12, pp. 1176–1206, Dec. 2018.

[21] N. Pritam, M. Khari, L. H. Son, R. Kumar, S. Jha, I. Priyadarshini, M. Abdel-Basset, and H. V. Long, "Assessment of code smell for predicting class change proneness using machine learning," *IEEE Access*, vol. 7, pp. 37414–37425, 2019.

MOHAMMAD A. ALKANDARI received the Ph.D. degree in computer science from the College of Engineering, Virginia Polytechnic Institute and State University (Virginia Tech). He was the Director of the Office of Engineering Education Technology, College of Engineering and Petroleum, Kuwait University, for three years. He is currently an Assistant Professor of computer engineering with Kuwait University, Kuwait, where he has been on the faculty, since 2012. He is also the Coordinator of the Software and Systems Engineering Research Group, Department of Computer Engineering. He is a Researcher in software engineering, requirements engineering, software project management, software quality assurance, software safety and security, the Internet of Things, and human–computer interaction.

ALI KELKAWI received the B.E. degree in computer engineering from the American University of Kuwait, Kuwait, in 2018. He is currently pursuing the M.Sc. degree in computer engineering with Kuwait University. In 2018, he joined the Department of Computer Science, Gulf University for Science and Technology, Kuwait, where he is currently a Teaching Assistant. His research interests include evolutionary computation methods and artificial intelligence.



MAHMOUD O. ELISH (Senior Member, IEEE) received the Ph.D. degree in computer science from George Mason University, in 2005. He is currently an Associate Professor and the Head of the Department of Computer Science, Gulf University for Science and Technology, Kuwait. He has published more than 45 papers in peer-reviewed journals and international conferences, and his work has received over 1300 citations. His research interests include software metrics, software quality, software maintenance and evolution, software security, empirical software engineering, and computational intelligence in software engineering.