IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 June 2023, accepted 16 July 2023, date of publication 25 July 2023, date of current version 2 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3298678

RESEARCH ARTICLE

A Refactoring Classification Framework for Efficient Software Maintenance

ABDULLAH ALMOGAHED[1], HAIRULNIZAM MAHDIN[1], MAZNI OMAR[2], NUR HARYANI ZAKARIA[2], SALAMA A. MOSTAFA[1], SALMAN A. ALQAHTANI[3], (Member, IEEE), PRANAVKUMAR PATHAK[4], SHAZLYN MILLEANA SHAHARUDIN[5,6], AND RAHMAT HIDAYAT[7], (Member, IEEE)

[1]Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia (UTHM), Johor Bahru 86400, Malaysia
[2]School of Computing, Universiti Utara Malaysia, Sintok 06010, Malaysia
[3]Department of Computer Engineering, College of Computer and Information Sciences, King Saud University, Riyadh 11543, Saudi Arabia
[4]School of Continuing Studies, McGill University, Montreal, QC H3A 0G4, Canada
[5]Department of Mathematics, Faculty of Science and Mathematics, Universiti Pendidikan Sultan Idris, Tanjong Malim 35900, Malaysia
[6]Department of Statistics, Columbia University, New York, NY 10032, USA
[7]Department of Information Technology, Politeknik Negeri Padang, Padang, Sumatera Barat 25164, Indonesia

Corresponding authors: Hairulnizam Mahdin (hairuln@uthm.edu.my) and Abdullah Almogahed (abdullahm@uthm.edu.my)

This Research is funded by Research Supporting Project Number (RSPD2023R585), King Saud University, Riyadh, Saudi Arabia.

ABSTRACT The expenses associated with software maintenance and evolution constitute a significant portion, surpassing more than 80% of the overall costs involved in software development. Refactoring, a widely embraced technique, plays a crucial role in streamlining and minimizing maintenance activities and expenses. However, the effect of refactoring techniques on quality attributes presents inconsistent and conflicting findings, making it challenging for software developers to enhance software quality effectively. Additionally, the absence of a comprehensive framework further complicates the decision-making process for developers when selecting appropriate refactoring techniques aligned with specific design objectives. In light of these considerations, this research aims to introduce a novel framework for classifying refactoring techniques based on their measurable influence on internal quality attributes. Initially, an exploratory study was conducted to identify commonly employed refactoring techniques, followed by an experimental analysis involving five case studies to evaluate the effects of these techniques on internal quality attributes. Subsequently, the framework was constructed based on the outcomes of the exploratory and experimental studies, further reinforced by a multi-case analysis. Comprising three key components, namely the methodology for applying refactoring techniques, the Quality Model for Object-Oriented Design (QMOOD), and the classification scheme for refactoring techniques, this proposed framework serves as a valuable guideline for developers. By comprehending the effect of each refactoring technique on internal quality attributes, developers can make informed decisions and select suitable techniques to enhance specific aspects of their software. Consequently, this framework optimizes developers' time and effort by minimizing the need to weigh the pros and cons of different refactoring techniques, potentially leading to a reduction in maintenance activities and associated costs.

INDEX TERMS Refactoring classification, software metrics, software refactoring, refactoring techniques, software quality, software maintenance, internal quality attributes.

I. INTRODUCTION

Software refactoring refers to a method aimed at enhancing the internal design of a software system without altering its functionality, thereby improving the overall quality of the design [1], [2]. This approach serves to reduce maintenance activities and costs associated with the software [3], [4]. It is recognized as a standard solution that focuses on restructuring the software's design structure while preserving its functionality [5], [6]. Consequently, software refactoring

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

has emerged as a vital aspect of software maintenance and evolution [7], [8], becoming increasingly indispensable in the field of software development due to the evolving IT landscape and user requirements [3]. The significance of refactoring within software development processes has grown significantly, as it can be prompted by various factors such as new requirements, adaptation to diverse contexts, and subpar quality [3], [9]. Refactoring is widely adopted as a prominent technique for enhancing the quality of existing software systems in practical settings [10], [11], [12], establishing an important connection between quality attributes [13].

Within this context, previous empirical investigations have explored the influence of applying refactoring techniques on quality attributes. By examining pertinent literature, these investigations have yielded a variety of outcomes. Certain research studies suggest that the utilization of refactoring techniques has a favorable impact on the quality of software [14], [15], [16], [17], [18]. Conversely, other studies suggest a negative effect of refactoring on quality attributes [19], [20]. Some studies find no significant effect of refactoring techniques on quality attributes [21], [22], [23]. Additionally, there are cases where the effect of refactoring on quality attributes remains ambiguous and lacks clarity [24], [25], [26].

Refactoring is a widely acknowledged practice that has been widely utilized to improve the quality of software systems [27], [28]. However, it has been noted that the impact of software refactoring on different software quality attributes is not consistently positive [13], [29], [30], [31]. An analysis of literature reviews demonstrates varying and conflicting outcomes regarding the influence of refactoring techniques on software quality attributes. This inconsistency can be attributed to the fact that different techniques have distinct effects on different attributes [30], [31]. In essence, there is conflicting evidence concerning the benefits of refactoring. Research findings indicate that the effects of different refactoring techniques on software quality attributes can vary significantly. In some cases, these effects may even be contradictory or opposing to one another [13], [30], [31], [32], [33]. Moreover, the particular sequence in which refactoring techniques are carried out can have different impacts on quality attributes [8], [33]. Consequently, it becomes challenging to differentiate the individual impacts of each refactoring technique or draw definitive conclusions about their impact on quality [13], [34]. A harmful practice is failing to recognize the refactoring techniques used or failing to use each technique separately [30].

Using refactoring techniques to improve software quality presents challenges for developers, given the inconsistent or contradictory results regarding their effects [8], [34]. Chaparro et al. [35] highlight the difficulty developers face in assessing the advantages and disadvantages of refactoring techniques, which becomes even more complex when dealing with conflicting techniques. Moreover, Nyamawe et al. [27] and Almogahed and Omar [36] point out the challenges developers encounter when selecting the most suitable refactoring technique from a range of options to address design flaws. Determining the appropriate type of refactoring to apply is often a daunting task [27]. Consequently, a classification framework that classifies refactoring techniques according to how they influence software quality attributes can support developers in attaining their design goals. By analyzing the specific purpose and impact of every refactoring technique on quality attributes, developers can make informed decisions and choose the most beneficial techniques for their intended purposes [31], [37], [38], [39].

Several studies have attempted to classify refactoring techniques according to their influence on desired quality attributes, although their scope has been limited [31]. These studies [28], [40], [41], [42], [43] have classified a subset of refactoring techniques regarding particular quality attributes. Nevertheless, these classifications have certain limitations, as they do not encompass a comprehensive range of internal quality attributes, for example, composition, abstraction, hierarchies, encapsulation, messaging, and polymorphism. The ongoing debate surrounding the impacts of refactoring techniques on software design quality is characterized by diverse and conflicting opinions [31], [39]. These findings highlight the insufficiency of current research on the subject of refactoring techniques and their impact on software quality [30], [31], [38], [39].

The current lack of frameworks for software developers to identify appropriate refactoring techniques for achieving specific design goals has been noted [8], [39]. Developers require explicit guidance on the optimal path and refactoring choices to effectively achieve desired designs [8], [44]. Researchers are encouraged to explore various aspects in conjunction with refactoring, including the recommendation of specific refactoring techniques [13]. Addressing the challenge of recommending suitable refactoring techniques is highlighted by Abid et al. [3]. To ensure consistent and effective utilization of refactoring techniques, the establishment of a refactoring classification framework is necessary [8]. This framework would serve as a comprehensive set of guidelines to assist software practitioners in selecting appropriate refactoring techniques. However, to date, no study has proposed a refactoring classification framework specifically designed to enhance the internal quality attributes of software systems.

To address these gaps, the primary aim of this study is to develop a comprehensive refactoring classification framework that classifies refactoring techniques based on their impact on internal quality attributes. By utilizing this framework, software practitioners will have the ability to enhance the quality of software systems by choosing and implementing the appropriate refactoring techniques in the appropriate locations. The proposed framework will serve as a valuable guideline, aiding software practitioners in gaining a deeper understanding of the connections between refactoring techniques and object-oriented properties. Additionally, it will enable them to enhance the software system's quality by selecting suitable refactoring techniques aligned with

specific design objectives, ultimately enhancing targeted quality attributes.

This paper is organized as follows: A description of the relevant work done in the field is given in Section II. The methodology used in this study is described in Section III. The results are then presented and thoroughly discussed in Section IV, which follows. Threats to validity are dealt with in Section V. Finally, Section VI brings the paper to a close by outlining our goals for future research.

II. RELATED WORK

Numerous investigations have been undertaken with the objective of categorizing refactoring techniques according to their impact on internal quality metrics. Bois and Mens [45] conducted a categorization of refactoring techniques according to their influence on a few internal quality metrics (LCOM, NOM, CBO, NOC, and RFC), specifically focusing on four techniques: Encapsulate Field, Extract Method, Pull Up Method Subclass, and Pull Up Method Superclass. Their study utilized a simple Java package comprising four classes as a case study. However, there were fewer refactoring techniques and internal quality metrics taken into account in this classification. Furthermore, the experiment employed a small demo program.

In another study, Bois et al. [46] presented a classification that served as a guideline for identifying the conditions under which applying refactoring techniques could enhance code coupling and cohesion. They identified five specific techniques (Replace Data Value with Object, Move Method, Extract Class, Replace Method with Method Object, and Extract Method) that demonstrated improvements in coupling and cohesion. Elish and Alshayeb [24], [40], [41] proposed classifications of several refactoring techniques, including Extract Class, Consolidate Conditional Expression, Extract Method, Hide Method, and Encapsulate Field, based on their measurable effects on internal metrics (LOC, WMC, RFC, NOM, and FOUT). However, the number of refactoring techniques and quality attributes taken into account in these classifications was constrained. It's worth noting that small systems were employed to establish these classifications.

In their study, Malhotra and Chug [42] focused on classifying refactoring techniques according to their impact on quality attributes. They specifically considered five commonly used techniques: Hide Method, Extract Method, Extract Class, Encapsulate Field, and Consolidate Conditional Expression, along with four quality attributes: inheritance, complexity, coupling, and cohesion. The analysis aimed to categorize these refactoring techniques in line with their impacts on quality attributes. To conduct the experiments, five C# software projects from the academic environment were utilized. Similarly, Malhotra and Jain [43] led a study to investigate the individual effects of four refactoring techniques: Encapsulate Field, Wrap Return value, Replace Constructor with Builder, and Replace Constructor with Factory, on four quality attributes: inheritance, cohesion, coupling, and complexity.

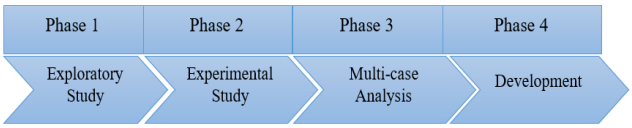


FIGURE 1. Refactoring classification framework development methodology.

Almogahed et al. cite47 investigated the influence of five frequently employed refactoring techniques (Hide Method, Encapsulate Field, Inline Method, Extract Method, and Remove Setting Method) on the security characteristic in the context of information hiding. These techniques were then classified based on security metrics.

However, these existing classifications exhibit several shortcomings that render them insufficiently comprehensive. These shortcomings can be outlined as follows:

- The classifications are limited in scope as they only encompass a specific set of refactoring techniques, failing to cover a wide range of available techniques.
- The selection of refactoring techniques lacks industry evidence and is primarily based on subjective judgments made by researchers or derived from literature review analysis. It is crucial to validate the relevance and usefulness of refactoring techniques by incorporating feedback from industry practitioners.
- The classifications are confined to a narrow range of internal quality properties. Essential internal quality properties such as messaging, hierarchies, abstraction, polymorphism, composition, and encapsulation have not been taken into consideration, limiting the comprehensiveness of the classifications.

Therefore, the purpose of this study is to overcome these limitations by introducing a comprehensive classification framework. This framework aims to classify commonly used refactoring techniques based on a wide range of internal quality attributes. To achieve this, the study will investigate the prevalent refactoring techniques employed by software practitioners in their current practices. Furthermore, it will assess the individual impacts of these refactoring techniques on internal quality properties such as abstraction, hierarchies, messaging, composition, encapsulation, and polymorphism.

III. METHODOLOGY

In this section, we will outline the development phases of the refactoring classification framework. The framework is constructed through four essential process phases: 1) exploratory study, 2) experimental study, 3) multi-case analysis, and 4) development. The methodology for developing the refactoring classification framework is visually depicted in Figure 1.

The exploratory study aimed to identify the most commonly used refactoring techniques and object-oriented properties. Additionally, five case studies of varying sizes were selected for conducting the experiments. In the experimental

study, a total of 39 experiments were conducted to investigate the individual effects of 10 refactoring techniques on internal quality attributes. Throughout the five case studies, the refactoring techniques were performed a total of 889 times. A multi-case analysis was then carried out to classify the refactoring techniques according to their influence on the internal quality attributes. The proposed framework was constructed based on the findings from the exploratory and experimental studies as well as the multi-case analysis. Detailed explanations of these four phases will be provided in the subsequent sections.

A. EXPLORATORY STUDY

The exploratory study serves as the initial phase of this research, involving a critical review and in-depth analysis of relevant literature. Its purpose is to establish the scope of the proposed classification framework. Within this phase, three primary activities were conducted. Firstly, the identification of the commonly utilized refactoring techniques in current software refactoring practices was carried out. Secondly, internal quality attributes and metrics were identified for inclusion in the study. Lastly, case studies from both real-world and academic environments were carefully selected for experimental purposes. Detailed information regarding these three activities will be presented in the subsequent subsections.

1) IDENTIFYING THE REFACTORING TECHNIQUES

In the domain of object-oriented paradigms, Fowler et al. [1] introduced a comprehensive refactoring catalog encompassing 68 original techniques, categorized into six distinct groups. To determine the most commonly employed refactoring techniques, an extensive review and analysis of existing literature were conducted in this study. Notably, five systematic literature reviews [13], [30], [48], [49], [50] and one systematic mapping study [31] identified frequently utilized refactoring techniques within academic research. In addition, Kim et al. [15], [51] identified prevalent refactoring techniques employed in the software engineering practices at Microsoft. Gatrell and Counsell [52] solicited input from ex-industry developers to select 13 refactoring techniques that are deemed most likely to be applied and represent a broad range of refactoring practices. Furthermore, Ouni et al. [53] reported widely adopted refactoring techniques in practical settings. A recent survey by Almogahed and Omar [36] identified the most frequently employed refactoring techniques among practitioners in current industry practices.

Based on the analysis conducted in this study, the following 10 refactoring techniques emerged as the most frequently employed: 1) Add Parameter (AP), 2) Extract Class (EC), 3) Encapsulate Field (EF), 4) Hide Method (HM), 5) Inline Class (IC), 6) Pull Up Field (PUF), 7) Pull Up Method (PUM), 8) Push Down Field (PDF), 9) Push Down Method (PDM), and 10) Remove Parameter (RP).

2) IDENTIFYING OBJECT ORIENTED PROPERTIES AND METRICS

When selecting appropriate metrics, it is essential to consider those that have been empirically validated through previous studies. Various metric suites, such as Metrics for Object-Oriented Designs (MOOD) [54], Lorenz and Kidd (L&K) [55], Quality Model for Object-Oriented Design (QMOOD) [56], and Chidamber and Kemerer (C&K) [57], have undergone empirical validation and are extensively utilized in object-oriented environments [31], [39], [47], [58], [59]. To accomplish the goals of this research, it is essential to employ a comprehensive quality model capable of assessing the quality of software systems and evaluating the influence of refactoring techniques on their quality. This quality model should include the ability to measure and analyze the internal quality attributes of the software.

Therefore, for this study, the QMOOD model proves to be more suitable due to its comprehensive nature in evaluating software design quality [60]. With its 11 internal quality attributes, the QMOOD model provides a broader perspective on software quality compared to other metrics specific to object-oriented design [61]. These properties effectively capture the essential characteristics of object-oriented systems [60] and offer a comprehensive understanding of software design quality [61]. Furthermore, the widespread use of QMOOD metrics allows for assessment at both the system and class levels, making them valuable in evaluating software designs [59].

Consequently, for this study, a comprehensive set of object-oriented properties and corresponding metrics were selected from the QMOOD model. These properties encompass complexity, hierarchies, polymorphism, inheritance, abstraction, messaging, cohesion, coupling, composition, encapsulation, and design size. The specific metrics chosen to capture these properties include number of complexity (NOM), number of hierarchies (NOH), number of polymorphic methods (NOP), measure of functional abstraction (MFA), average number of ancestors (ANA), class interface size (CIS), cohesion among methods in a class (CAM), direct class coupling (DCC), measure of aggregation (MOA), data access metric (DAM), and design size in classes (DSC). Each metric serves to measure a distinct aspect: DSC quantifies design size, NOH captures hierarchies, ANA evaluates abstraction, DAM assesses encapsulation, DCC measures coupling, CAM gauges cohesion, MOA quantifies composition, MFA reflects inheritance, NOP evaluates polymorphism, CIS quantifies messaging, and NOM measures complexity.

3) SELECTING CASE STUDIES

To ensure a comprehensive and generalized investigation, case studies were collected from two distinct environments: the real-world and academic settings. The rationale behind this selection was to incorporate projects developed by programmers with varying abilities and expertise, ranging from

students and beginners to professionals. By considering these diverse development skills and environments, the study's investigations become more comprehensive and allow for the generalization of results. The proposed refactoring classification framework in this study was developed using five well-known case studies [8], [30], [31]: the bank management system (BMS) [62], library management system (LMS) [63], payroll management system (PMS) [64], jHotDraw [65], and jEdit [66]. Including these varied case studies adds further depth and applicability to the research.

B. EXPERIMENTAL STUDY

An experimental study aims to uncover, describe, validate, and gain a comprehensive understanding of the processes, activities, and characteristics of a current phenomenon. Its purpose is to extend or develop new theories or methods that can enhance current practice [67]. Within the scope of this study, a total of 39 experiments were conducted across five case studies. The main aim of these experiments was to examine the influence of each refactoring technique individually on object-oriented properties. The next sections will provide a detailed outline of the experimental plan, including the procedures and materials used during the experiments.

1) EXPERIMENTAL PROCEDURES

The total count of experiments conducted in a case study aligns with the number of identified chances to apply refactoring techniques within that particular case study. Hence, for each experiment, there exists a unique version of the case study where a specific refactoring technique is applied. Essentially, every case study's original version underwent multiple independent runs, each run representing the application of a detected refactoring opportunity. After each run, a new version of the case study was generated, reflecting the changes resulting from the applied refactoring technique. Figure 2 illustrates the experimental steps undertaken to analyze the influence of each refactoring technique on internal quality attributes across different case studies.

Each experiment was carried out through several steps, as described in the following:

Step One: Identifying opportunities to perform a refactoring technique

In this step, the classes of a software project were analyzed to find out where they may need refactoring to improve the quality of the design. Fowler, in his book, provided examples to explain how to use the refactoring techniques [1], [2]. By the end of this step, the possible classes that require refactoring had been identified, and the related refactoring technique had been selected.

Step Two: Collecting object-oriented metrics before performing a refactoring technique

The object-oriented metrics (CIS, MFA, NOH, DCC, NOP, CAM, DSC, DAM, ANA, NOM, and MOA) were collected to measure their relevant internal quality attributes (inheritance, coupling, cohesion, design size, composition, encapsulation, polymorphism, abstraction, messaging, hierarchies, and complexity). The object-oriented metrics were automatically collected by Eclipse Metrics plugin 1.3.8.

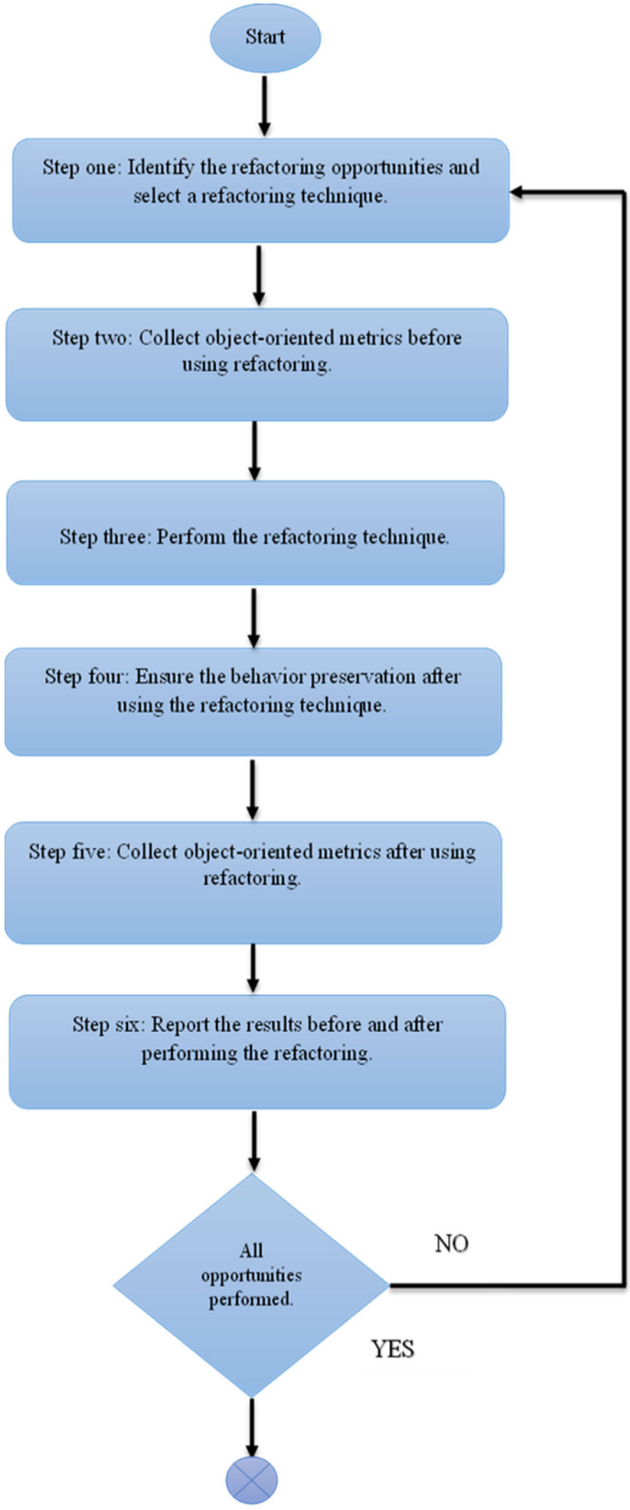


FIGURE 2. Experiment steps.

Step Three: Performing the refactoring technique selected individually

To assess the influence of each selected refactoring technique on internal quality attributes, they were individually applied. The procedures for using each refactoring technique were outlined by Fowler in his works [1], [2]. Refactoring techniques are capable of being carried out manually or with the assistance of software tools, although only a few techniques have dedicated tools. For this study, the Eclipse Refactor plug-in was utilized for three refactoring techniques (Add Parameter, Encapsulate Field, and Remove Parameter). However, even when using the tool, manual verification was conducted to ensure adherence to Fowler's recommended mechanics. This manual check was necessary due to the potential for errors in existing refactoring tools, which could lead to inaccurately refactored code segments [13], [30]. For the remaining refactoring techniques, they were manually performed according to Fowler's prescribed mechanics, as there were no available tools specifically designed for those techniques.

Step Four: Ensuring behavior preservation after using refactoring techniques

Behavior preservation of a system means the outputs of the software system after implementing the refactoring must be maintained as its outputs before applying the refactoring [31]. The preservation of the behavior of the software systems was achieved by regression testing. In addition, preserving the behavior of the software systems was achieved by compiling and executing the source codes after using the refactoring and comparing their outputs after the refactoring with their outputs before the refactoring.

Step Five: Collecting object-oriented metrics after performing a refactoring technique

After performing the selected refactoring technique, the object-oriented metrics have been collected to assess the relevant internal quality attributes.

Step Six: Reporting the measured metrics of the object-oriented properties before and after performing the refactoring technique, as well as saving the source code after performing the refactoring technique.

Step Seven: After completing steps one to six for a particular refactoring technique, the process returns to the original software version. This cycle continues for all the selected refactoring techniques, repeating steps one to six each time. This iterative process ensures that every opportunity to apply the chosen refactoring techniques has been thoroughly explored, leaving no chance for any of them to be overlooked.

2) EXPERIMENTAL MATERIALS

This section presents a description of the tools employed in the experimental process. The two tools utilized for conducting the experiments include the Eclipse refactoring tool and the Eclipse Metrics 1.3.8 tool. Detailed descriptions of these tools are presented in the subsequent subsections.

a: ECLIPSE REFACTORING TOOL

Eclipse, an open-source Integrated Development Environment (IDE) developed by IBM, offers a comprehensive tool platform for software development. It provides a universal development environment for various programming languages and supports a wide range of functionalities. The Eclipse IDE can be downloaded from the official website at [68]. Eclipse is one of the popular IDEs that supports automated refactoring and is a widely used refactoring tool [69]. Eclipse supports a number of refactoring techniques listed in Fowler's catalog [10]. Eclipse Refactoring has gained significant recognition for its ongoing enhancements and continuous efforts to optimize the utilization of refactoring techniques [13]. One of the benefits of utilizing this tool is that it guarantees the effective implementation of refactoring operations [13]. However, it is up to the developers to find and know the refactoring to apply [13].

b: ECLIPSE METRICS TOOL

The Eclipse Metrics 1.3.8 tool, accessible at [70], is an open-source Eclipse plug-in designed for calculating metrics and analyzing dependencies [71], [72]. It offers a comprehensive range of metrics, including those proposed in the QMOOD. For this study, the Eclipse Metrics 1.3.8 tool was utilized to gather 11 QMOOD metrics for evaluating the internal quality attributes. The selection of this tool was motivated by its widespread usage in numerous research applications focusing on Java. Moreover, it is compatible with major operating systems such as Windows, Mac, and Linux [73].

C. MULTI-CASE ANALYSIS

The application of a multi-case analysis proves valuable in unraveling the intricate mechanisms of complex phenomena or systems [74], [75]. This approach facilitates researchers development of a more profound comprehension of theoretical constructs associated with novel phenomena or systems under investigation. Within the scope of this study, the primary objective of the multi-case analysis is to classify refactoring techniques according to their impact on internal quality attributes. The multi-case analysis was performed on the many experiments (39) that were conducted on five case studies. All measurement information relevant to the internal quality attributes before and after performing the refactoring techniques across all experiments was put in one pool. In this study, the term pool refers to combining all experiments' data before and after performing the refactoring in one place and dealing with them as one unit to conduct the multi-case analysis. The data collected from the pool underwent comprehensive analysis and cross-case comparison, enabling the attainment of general insights [76]. Through the categorization of cases, similarities and differences were identified [77], offering a deeper understanding of the patterns and variations within the dataset.

1) CLASSIFICATION OF REFACTORING TECHNIQUES

To classify each refactoring technique individually based on its impact on the internal quality attributes, a common practice design approach was employed. This approach involved

identifying, comparing, and analyzing the impacts of every refactoring technique on the internal quality attributes across the five case studies in the pool. The analysis took into account the frequency of occurrence of each effect in the experiments. The impact with the highest occurrence was subsequently selected and classified for inclusion in the refactoring classification framework.

To analyze the influence of each refactoring technique on the internal quality attributes in the dataset, a comparison was made between the calculated values of object-oriented metrics before and after applying the technique across the five case studies. The QMOOD was utilized to interpret the differences observed in the computed values. The assessment of each refactoring technique involved subtracting the calculated value of the object-oriented metrics before refactoring from the calculated value after refactoring. A positive difference indicated that the refactoring technique increased the corresponding quality attribute, while a negative difference indicated an adverse effect on the quality attribute. A difference of zero signified that the refactoring technique had no impact on the related quality attribute.

After this analysis, the number of times for each effect (improve, impair, or no effect) on the quality attribute in the pool was counted for each refactoring technique. The percentage for each effect was then determined. The effect with the highest percentage of effect was classified and chosen for the refactoring classification framework. This process was replicated for all refactoring techniques across the five case studies in the pool.

D. CLASSIFICATION FRAMEWORK DEVELOPMENT

In this phase, three main activities were performed: 1) developing a glossary and common terminology; 2) defining components of the framework and their name conventions; and 3) a graphical representation of the framework. Three terminologies were identified and used in the proposed framework. There are ineffective refactoring techniques, unsafe refactoring techniques, and safe refactoring techniques. In this study, the proposed framework was composed of three components, and each component contained items. The three components were identified based on the literature review, the findings of the exploratory study, the findings of the experimental study, and the findings of a multi-case analysis. The three components of the proposed framework are: 1) the methodology of applying refactoring techniques; 2) the QMOOD model; and 3) the classification of refactoring techniques. The proposed framework was designed based on the goals, results, glossary, modeling, and conventions obtained from the previous phases. The conceptual approach was used to design the proposed framework. The framework presents the three components, their items, and the interactions among them in a detailed way.

IV. RESULTS AND DISCUSSION

A total of 39 experiments were conducted as part of this study, distributed across different case studies: 5 in BMS, 4 in LMS, 10 in PMS, 10 in jHotDraw, and 10 in jEdit. Throughout these case studies, a total of 889 refactoring techniques were performed. The frequency with which each refactoring technique was employed across all five case studies is displayed in Table 1.

TABLE 1. Statistical information regarding the utilization of individual refactoring techniques in the five case studies.

Refactoring Technique	Case studies					Total
	LMS	BMS	P MS	jHot Draw	jEdit	
Extract Class	3	7	2	13	77	102
Inline Class	1	0	2	9	40	52
Encapsulate Field	2	0	20	39	104	165
Add Parameter	0	1	1	16	34	52
Remove Parameter	0	0	2	10	14	26
Hide Method	2	32	23	164	230	451
Pull Up Field	0	1	1	3	12	17
Pull Up Method	0	0	1	2	4	7
Push Down Method	0	0	2	4	4	10
Push Down Field	0	3	2	1	1	7
Total	8	44	56	261	520	889

The 10 refactoring techniques have been classified based on internal quality attributes. The refactoring techniques have been classified into three classifications, described as follows:

- Safe Refactoring Techniques (SRT):

These are the designated refactoring techniques that were classified according to their beneficial effect on internal quality attributes.

- Unsafe Refactoring Techniques (URT):

These are the specific refactoring techniques that were classified according to their adverse effect on internal quality attributes.

- Ineffective Refactoring Techniques (IRT):

These are the specific refactoring techniques that were classified according to their null effect on internal quality attributes.

The impact of every refactoring technique (RT) on the internal quality attributes is determined by computing the difference between the metric value before applying RT and the metric value after applying RT. The calculation was performed using the following formula:

$$\begin{aligned}\Delta\text{Value of internal quality attribute} &= \text{its metric value after using RT} \\ &\quad - \text{its metric value before using RT}\end{aligned} \tag{1}$$

If the Δ value of the internal quality attribute is positive, the refactoring technique is considered a safe refactoring technique (SRT) as it improves the quality attribute. Conversely, if the Δ value of the internal quality attribute is negative, the refactoring technique is classified as an unsafe refactoring technique (URT) as it negatively impacts the internal quality attribute. If the Δ value of the internal quality attribute is zero, the refactoring technique is classified as an ineffective refactoring technique (IRT) for that specific internal quality attribute, indicating that it does not alter the

TABLE 2. Summary of the results of the multi-case analysis: the impact of every refactoring technique on internal quality attributes.

Refactoring Techniques	Abstraction	Cohesion	Complexity	Composition	Coupling	Design Size	Encapsulation	Hierarchies	Inheritance	Messaging	Polymorphism
Extract Class	↓	↑	↑	↑	↑	↑	↑	–	–	↑	–
Inline Class	↑	↓	–	–	↓	↓	↓	–	–	–	–
Encapsul ate Field	–	↓	↑	–	–	–	↑	–	–	↑	–
Add Parameter	–	–	–	–	–	–	–	–	–	–	–
Remove Parameter	–	–	–	–	–	–	–	–	–	–	–
Hide Method	–	–	–	–	–	–	–	–	–	↓	–
Pull Up Field	–	↓	↑	–	–	–	–	–	–	↑	–
Pull Up Method	–	↓	↓	–	–	–	–	–	–	↓	–
Push Down Method	–	↑	↓	–	–	–	–	–	–	↓	↓
Push Down Field	–	–	–	–	–	–	–	–	–	–	–

attribute. The refactoring techniques are classified according to their highest common impact observed across the five case studies for each internal quality attribute. In other words, the classification considers the refactoring technique's highest rate of effect and the highest average rate of equality in relation to its impact on each object-oriented property. Table 2 summarizes the multi-case analysis results of the impact of each refactoring technique on the internal quality attributes, where the symbol (↑) refers to the improvement of the internal quality attribute (except for complexity and coupling), the symbol (↓) refers to the impairment of the internal quality attribute (except for complexity and coupling), and the symbol (−) refers to the no change quality attribute.

Extract Class (EC) increased cohesion, composition, design size, encapsulation, and messaging; consequently, it was classified as SRT for these attributes. EC decreased abstraction and increased complexity and coupling; accordingly, it was classified as URT for these attributes. EC did not change hierarchies, inheritance, or polymorphism; consequently, it was classified as IRT for these attributes.

Inline Class (IC) increased abstraction and decreased coupling; consequently, it was classified as SRT for these attributes. IC decreased cohesion, design size, and encapsulation; accordingly, it was classified as URT for these attributes. IC did not change composition, messaging, complexity, hierarchies, inheritance, or polymorphism; consequently, it was classified as IRT for these attributes.

Encapsulate Field (EF) increased encapsulation and messaging; consequently, it was classified as SRT for these attributes. EF decreased cohesion and increased complexity; accordingly, it was classified as URT for these attributes. EF did not change abstraction, coupling, design size, composition, hierarchies, inheritance, or polymorphism; consequently, it was classified as IRT for these attributes.

Hide Method (HM) decreased messaging and was classified as URT for this attribute. HM did not change design size, inheritance, abstraction, polymorphism, complexity, encapsulation, composition, hierarchies, coupling, or cohesion; consequently, it was classified as IRT for these attributes.

Pull Up Field (PUF) increased messaging and was classified as SRT for this attribute. PUF decreased cohesion and increased complexity; accordingly, it was classified as URT for these attributes. PUF did not change abstraction, polymorphism, composition, inheritance, encapsulation, hierarchies, design size, or coupling; consequently, it was classified as IRT for these attributes.

Pull Up Method (PUM) decreased complexity and was classified as SRT for this attribute. PUM decreased cohesion and messaging; accordingly, it was categorized as URT for these attributes. PUM did not change abstraction, polymorphism, encapsulation, inheritance, coupling, hierarchies, design size, or composition; consequently, it was classified as IRT for these attributes.

Push Dawn Method (PDM) increased cohesion and decreased complexity; accordingly, it was classified as SRT for these attributes. PDM decreased messaging and polymorphism; accordingly, it was classified as URT for these attributes. PDM did not change abstraction, composition, coupling, design size, encapsulation, hierarchies, or inheritance; consequently, it was classified as IRT for these attributes.

Push Down Field (PDF), Remove Parameter (RP), and Add Parameter (AP) did not change any internal quality attribute and were therefore categorized as IRT.

1) THE PROPOSED CLASSIFICATION FRAMEWORK FOR REFACTORING TECHNIQUES

In this section, the framework was constructed based on its scope, purpose, and goals, as well as on the results of the exploratory study, the experimental study, and the multi-case analysis obtained from the preceding phases. The conceptual modeling approach was used to represent the framework. The design shows the components and their items. Figure 3 shows the refactoring classification framework.

The proposed framework was constructed based on three main components. These components were derived from the findings of the exploratory and experimental studies as well as the multi-case analysis. These components are described in the following subsections:

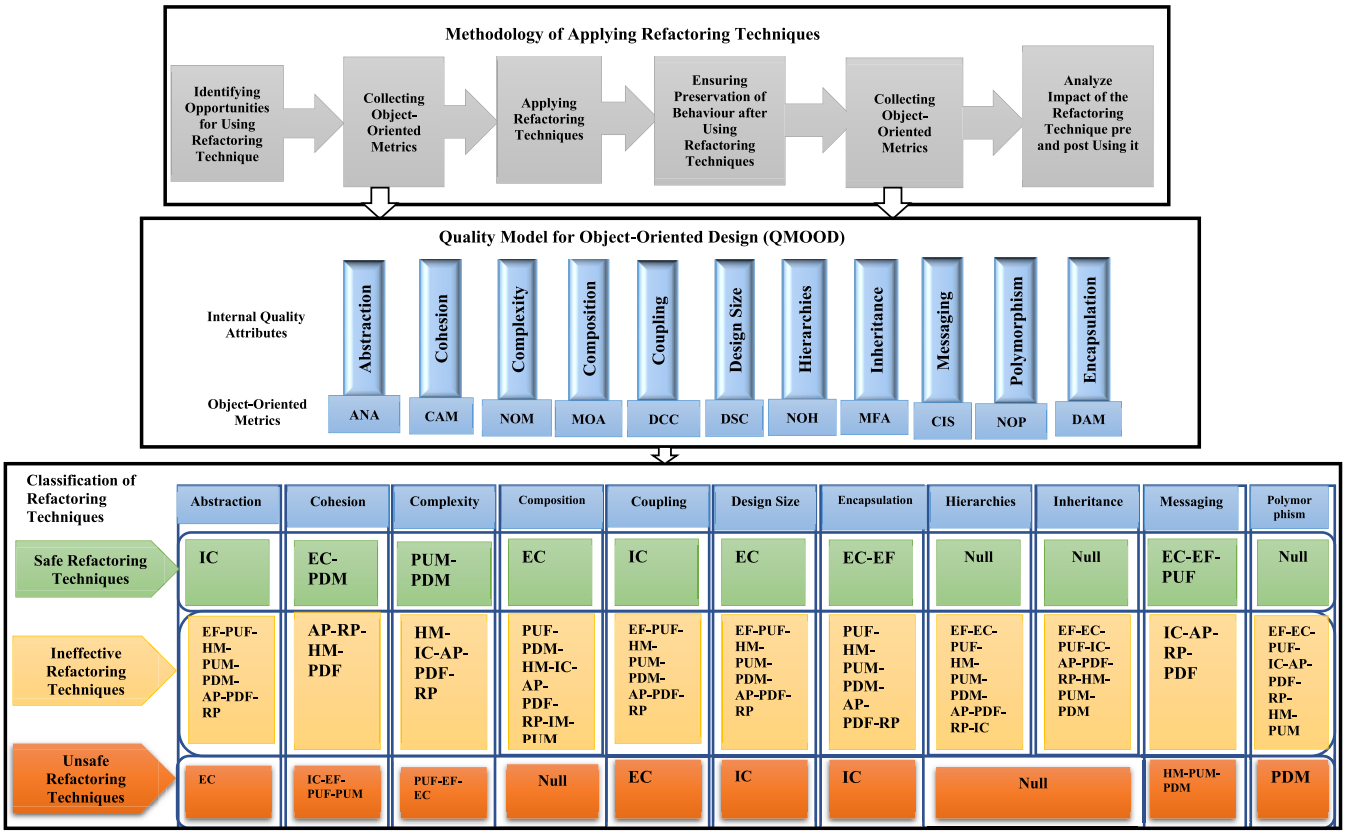


FIGURE 3. The proposed refactoring classification framework.

a: METHODOLOGY OF APPLYING REFACTORING TECHNIQUES

This component is derived from the literature review and applied to the experimental study of this research. The process employed to examine and analyze the effects of each refactoring technique on the internal quality attributes comprises the following stages:

1. Identifying opportunities for the application of refactoring techniques.
2. Gathering object-oriented metrics before applying a refactoring technique.
3. Applying each refactoring technique individually.
4. Ensuring the preservation of behavior after applying the refactoring techniques.
5. Collecting object-oriented metrics after applying the refactoring technique.
6. Analyzing the effects of the refactoring technique by comparing the pre- and post-application values on the internal quality attributes.

b: QUALITY MODEL FOR OBJECT-ORIENTED DESIGN (QMOOD)

This component is derived from the literature review and utilized for measurements in this research. The QMOOD framework was employed to measure the object-oriented properties both before and after applying each refactoring technique. The internal quality attributes considered in the measurements encompass the following elements: 1) complexity, 2) encapsulation, 3) abstraction, 4) design size, 5) polymorphism, 6) coupling, 7) composition, 8) messaging, 9) cohesion, 10) hierarchies, and 11) inheritance. The object-oriented metrics employed for the measurements are as follows: 1) DCC, 2) CAM, 3) CIS, 4) DSC, 5) DAM, 6) MOA, 7) NOP, 8) ANA, 9) MFA, 10) NOH, and 11) NOM.

c: CLASSIFICATION OF REFACTORING TECHNIQUES

The 10 refactoring techniques were derived from the findings of the exploratory study. The classification of the 10 refactoring techniques was derived from multi-case analysis. The 10 refactoring techniques were classified into three categories. The three categories are: 1) safe refactoring techniques; 2) unsafe refactoring techniques; and 3) ineffective refactoring techniques.

The methodology for applying refactoring techniques is outlined in the framework, which details the step-by-step procedures for investigating and analyzing the impact of each technique on internal quality attributes using the QMOOD. The process involves identifying opportunities for applying each technique and subsequently assessing its effect on object-oriented properties based on the QMOOD. The framework also elucidates the relationship between internal

quality attributes and object-oriented metrics within the QMOOD.

The classification of the 10 refactoring techniques into three categories (safe, unsafe, and ineffective) is an integral part of the framework. Stakeholders can refer to the framework's design to replicate the investigation procedures, utilizing the QMOOD or other relevant quality models in the field of refactoring. Software developers can select appropriate refactoring techniques from the proposed classification to enhance the quality of software systems, aligning with their specific design goals and objectives.

The proposed framework for classifying refactoring techniques brings significant value to software developers in the industry. By offering a well-organized and extensive classification of refactoring techniques based on their influence on internal quality attributes, the framework provides valuable guidance and assistance during the decision-making process of selecting and applying suitable refactoring techniques.

The framework equips software developers with the means to improve the quality of their software systems by effectively identifying and implementing the most appropriate refactoring techniques. Through a deep understanding of the correlation between refactoring techniques and internal quality attributes, developers are empowered to make well-informed choices that align with their design objectives, ultimately leading to overall enhancements in software quality.

The framework minimizes the time and effort expended by developers when evaluating the advantages and disadvantages of various refactoring techniques. By presenting empirical evidence and a classification system, the framework mitigates the risks of maintenance costs and effort by providing insights into the anticipated effects of each technique on software quality attributes. This enables developers to make confident decisions and streamline the refactoring process efficiently.

Moreover, the framework fosters consistency and standardization in refactoring practices. By establishing a shared language and comprehension of the impact of refactoring techniques on software quality attributes, the framework facilitates effective communication and collaboration among developers. Additionally, it serves as a valuable point of reference for future research and development endeavors within the software engineering domain.

In conclusion, the proposed refactoring classification framework empowers software developers to make informed decisions, elevate software quality, and streamline the refactoring process. Its contribution lies in presenting a structured approach for selecting and applying refactoring techniques, ultimately benefiting the entire software development community.

V. THREATS TO VALIDITY

In this section, we address the potential validity threats and the measures taken to mitigate them, ensuring increased confidence in the study's results. To systematically address the validity threats, we follow the classification proposed by Wohlin et al. [78] and Cook et al. [79]. These classifications encompass the four most prevalent validity threats: 1) construct validity, 2) conclusion validity, 3) internal validity, and 4) external validity. Subsequent sections elaborate on each potential threat to the study's validity and outline the strategies employed to address them effectively.

A. CONSTRUCT VALIDITY

Threats to construct validity pertain to the alignment between theory and observation, which in this study primarily stems from the measurement procedures employed. To mitigate this threat, a meticulous approach was adopted. Firstly, the selection of the 10 most widely employed refactoring techniques in both literature and practice was based on comprehensive literature reviews and an exploratory study involving software practitioners in the field. This approach aimed to avoid any potential bias or subjectivity in the technique selection process. Furthermore, adherence to Fowler's guidelines ensured the precise identification and execution of the selected refactoring techniques. Additionally, the study utilized a reliable quality model, namely the QMOOD, to accurately assess the impact of the refactoring techniques on the internal quality attributes.

B. CONCLUSION VALIDITY

Threats to the validity of the conclusion revolve around the connection between the treatment and the outcome. Conclusion validity pertains to the degree to which other researchers can reproduce the study's findings and obtain comparable results when employing the same procedures as the original researcher. To address this concern, 39 distinct experiments were conducted across five case studies (PMS, LMS, BMS, JHotDraw, and jEdit) during the development of the framework. This comprehensive approach provides ample evidence to draw valid conclusions. Moreover, to ensure replicability, a rigorous methodology was followed in performing the experiments, thereby enabling other researchers to replicate the study with precision.

C. INTERNAL VALIDITY

Internal validity refers to the extent to which a study establishes a reliable and causal relationship between a treatment or intervention and its observed outcomes. The case studies selected for analysis and evaluation in this study are predominantly focused on software refactoring, making them highly representative of the overall population. These case studies were subjected solely to the treatment of refactoring, enabling a focused observation of the impacts of refactoring techniques across different scenarios. Uniform treatment conditions were maintained across all case studies, ensuring consistency in the experimental setup. Careful preservation of the case study states after refactoring facilitated the computation of metrics. The experiments were conducted in a sequential manner, commencing with smaller case studies

and progressively advancing to larger ones. This progressive approach allowed the researcher to accumulate valuable experience throughout the course of the experiment.

D. EXTERNAL VALIDITY

External validity concerns the extent to which the results of a study can be generalized. To enhance the external validity of this research, experiments have been conducted on diverse case studies encompassing both open-source and academic Java software systems. The case studies selected for this research encompassed diverse application domains and varied in terms of their sizes. The primary focus of the study was on analyzing Java projects, as the refactoring techniques under investigation are primarily targeted at Java systems. It is important to note that Java is widely used in both industry and academia. However, it is crucial to acknowledge that the generalization of results to other programming languages may be limited due to potential variations in refactoring techniques and tool support. Researchers are encouraged to extend this work to other languages such as Python and JavaScript to broaden the scope of investigation and increase external validity.

VI. CONCLUSION AND FUTURE WORK

The software industry places great importance on software quality, as low-quality software can introduce complexity and become a significant maintenance concern. Therefore, software refactoring is recognized as a crucial practice in software maintenance and evolution. Nevertheless, when applying refactoring techniques to enhance software quality, developers encounter the challenge of balancing the potential improvement in some quality attributes with the risk of deterioration in others.

This study introduces a refactoring classification framework aimed at enhancing the internal quality attributes of software systems. The development of the framework consisted of four main phases: an exploratory study, an experimental study, a multi-case analysis, and the actual framework development. The exploratory study identified the most commonly used refactoring techniques and internal quality attributes. Additionally, five case studies of varying sizes were selected for the experiments. A total of 39 experiments were conducted in the experimental study to individually investigate the impact of 10 refactoring techniques on internal quality attributes. Across the five case studies, the refactoring techniques were applied 889 times. The multi-case analysis was performed to classify the refactoring techniques according to their effect on the 11 internal quality attributes. These phases were undertaken to gather initial insights, conduct experiments, analyze multiple case studies, and finally develop the framework based on the findings and observations from the previous stages.

The proposed framework consists of three components: a methodology for applying refactoring techniques, the QMOOD quality model, and the classification of refactoring techniques. The 10 commonly used refactoring techniques were classified into three categories (safe, unsafe, and ineffective) based on their impact on internal quality attributes. The framework aims to serve as a guideline for software developers, providing assistance in choosing suitable refactoring techniques to improve the internal quality attributes of software systems. By leveraging empirical evidence, the framework reduces the time and effort required for developers to evaluate conflicts and trade-offs among refactoring techniques, thereby mitigating the risks associated with software maintenance costs and effort.

In the future, the proposed framework will be expanded to include other commonly used refactoring techniques. Additionally, expert reviews in academic and industrial settings will be conducted to evaluate the framework's effectiveness. Furthermore, the framework will be empirically evaluated through various case studies conducted by domain experts.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley Professional, 2002.

[2] M. Fowler and K. Beck, *Refactoring Improving the Design of Existing Code Refactoring*, 2nd ed. Reading, MA, USA: Addison-Wesley Professional, 2019.

[3] C. Abid, V. Alizadeh, M. Kessentini, T. D. N. Ferreira, and D. Dig, "30 years of software refactoring research: A systematic literature review," 2020, *arXiv:2007.02194*.

[4] M. Kaya, S. Conley, Z. S. Othman, and A. Varol, "Effective software refactoring process," in *Proc. 6th Int. Symp. Digit. Forensic Secur. (ISDFS)*, Antalya, Turkey, Mar. 2018, pp. 1–6, doi: 10.1109/ISDFS.2018.8355350.

[5] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "Interactive and dynamic multi-objective software refactoring recommendations," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw.*, Sep. 2019, pp. 1–30. [Online]. Available: https://deepblue.lib.umich.edu/bitstream/handle/2027.42/147343/papertse.pdf?sequence=1

[6] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, "RefBot: Intelligent software refactoring bot," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 823–834. [Online]. Available: https://ieeexplore.ieee.org/document/8952287

[7] M. Alotaibi, "Advances and challenges in software refactoring: A tertiary systematic literature review," M.S. thesis, Dept. Softw. Eng., B. Thomas Golisano College Comput. Inf. Sci., Rochester Inst. Technol., Rochester, NY, USA, 2018.

[8] A. Almogahed, M. Omar, N. H. Zakaria, G. Muhammad, and S. A. AlQahtani, "Revisiting scenarios of using refactoring techniques to improve software systems quality," *IEEE Access*, vol. 11, pp. 28800–28819, 2023, doi: 10.1109/ACCESS.2022.3218007.

[9] D. Arcelli, V. Cortellessa, and D. D. Pompeo, "Performance-driven software model refactoring," *Inf. Softw. Technol.*, vol. 95, pp. 366–397, Mar. 2018, doi: 10.1016/j.infsof.2017.09.006.

[10] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, pp. 1–53, Aug. 2016, doi: 10.1145/2932631.

[11] L. Sousa, D. Cedrim, A. Garcia, W. Oizumi, A. C. Bibiano, D. Oliveira, M. Kim, and A. Oliveira, "Characterizing and identifying composite refactorings: Concepts, heuristics and patterns," in *Proc. IEEE/ACM 17th Int. Conf. Mining Softw. Repositories (MSR)*, Seoul, South Korea, May 2020, pp. 186–197, doi: 10.1145/3379597.3387477.

[12] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, and W. Oizumi, "Refactoring effect on internal quality attributes: What haven't they told you yet?" *Inf. Softw. Technol.*, vol. 126, Oct. 2020, Art. no. 106347, doi: 10.1016/j.infsof.2020.106347.

[13] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *J. Syst. Softw.*, vol. 167, Sep. 2020, Art. no. 110610, doi: 10.1016/j.jss.2020.110610.

[14] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Inf. Softw. Technol.*, vol. 51, no. 9, pp. 1319–1326, Sep. 2009, doi: 10.1016/j.infsof.2009.04.002.

[15] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, pp. 357–366, doi: 10.1109/ICSM.2012.6405293.

[16] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proc. Conf. Center Adv. Stud. Collaborative Res.*, 2013, pp. 132–146. [Online]. Available: https://dl.acm.org/doi/10.5555/2555523.2555539

[17] N. Naiya, S. Counsell, and T. Hall, "The relationship between depth of inheritance and refactoring: An empirical study of eclipse releases," in *Proc. 41st Euromicro Conf. Softw. Eng. Adv. Appl.*, Aug. 2015, pp. 88–91. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7302436

[18] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, May 2017, pp. 176–185. [Online]. Available: https://ieeexplore.ieee.org/document/7961515

[19] S. H. Kannangara and W. M. J. I. Wijayanayake, "An empirical exploration of refactoring effect on software quality using external quality factors," *Int. J. Adv. ICT Emerg. Regions*, vol. 7, no. 2, p. 36, May 2014, doi: 10.4038/icter.v7i2.7176.

[20] K. Stroggylos and D. Spinellis, "Refactoring-does it improve software quality?" in *Proc. 5th Int. Workshop Softw. Quality (WoSQ: ICSE Workshops)*, Minneapolis, MN, USA, May 2007, p. 10, doi: 10.1109/WOSQ.2007.11.

[21] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, pp. 1–14, Sep. 2015, doi: 10.1016/j.jss.2015.05.024.

[22] Q. D. Soetens and S. Demeyer, "Studying the effect of refactorings: A complexity metrics perspective," in *Proc. 7th Int. Conf. Quality Inf. Commun. Technol.*, Porto, Portugal, Sep. 2010, pp. 313–318, doi: 10.1109/QUATIC.2010.58.

[23] D. Wilking, U. F. Khan, and S. Kowalewski, "An empirical evaluation of refactoring," *E-Inform. Softw. Eng. J.*, vol. 1, no. 1, pp. 1–16, 2007. [Online]. Available: https://www.e-informatyka.pl/attach/e-Informatica_-_Volume_1/Vol1Iss1Art2eInformatica.pdf

[24] K. O. Elish and M. Alshayeb, "Investigating the effect of refactoring on software testing effort," in *Proc. 16th Asia–Pacific Softw. Eng. Conf.*, Dec. 2009, pp. 29–34, doi: 10.1109/APSEC.2009.14.

[25] M. Alshayeb, "Refactoring effect on cohesion metrics," in *Proc. Int. Conf. Comput., Eng. Inf.*, Fullerton, CA, USA, Apr. 2009, pp. 3–7, doi: 10.1109/ICC.2009.12.

[26] A. Halim and P. Mursanto, "Refactoring rules effect of class cohesion on high-level design," in *Proc. Int. Conf. Inf. Technol. Electr. Eng. (ICITEE)*, Yogyakarta, Indonesia, Oct. 2013, pp. 197–202, doi: 10.1109/ICITEED.2013.6676238.

[27] A. S. Nyamawe, H. Liu, Z. Niu, W. Wang, and N. Niu, "Recommending refactoring solutions based on traceability and code metrics," *IEEE Access*, vol. 6, pp. 49460–49475, 2018, doi: 10.1109/ACCESS.2018.2868990.

[28] A. Almogahed, M. Omar, and N. H. Zakaria, "Refactoring codes to improve software security requirements," *Procedia Comput. Sci.*, vol. 204, pp. 108–115, 2022, doi: 10.1016/j.procs.2022.08.013.

[29] A. Almogahed, M. Omar, and N. H. Zakaria, "Categorization refactoring techniques based on their effect on software quality attributes," *Int. J. Innov. Technol. Exploring Eng.*, vol. 8, no. 8S, pp. 439–445, 2019. [Online]. Available: https://www.ijitee.org/wp-content/uploads/papers/v8i8s/H10760688S19.pdf

[30] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 44–69, Jan. 2018, doi: 10.1109/TSE.2017.2658573.

[31] S. Kaur and P. Singh, "How does object-oriented code refactoring influence software quality? Research landscape and challenges," *J. Syst. Softw.*, vol. 157, Nov. 2019, Art. no. 110394, doi: 10.1016/j.jss.2019.110394.

[32] A. S. Nyamawe, "Mining commit messages to enhance software refactorings recommendation: A machine learning approach," *Mach. Learn. Appl.*, vol. 9, Sep. 2022, Art. no. 100316, doi: 10.1016/j.mlwa.2022.100316.

[33] I. Alazzam, B. Abuata, and G. Mhediat, "Impact of refactoring on OO metrics: A study on the extract class, extract superclass, encapsulate field and pull up method," *Int. J. Mach. Learn. Comput.*, vol. 10, no. 1, pp. 158–163, Jan. 2020, doi: 10.18178/ijmlc.2020.10.1.913.

[34] A. Almogahed, M. Omar, and N. H. Zakaria, "Recent studies on the effects of refactoring in software quality: Challenges and open issues," in *Proc. 2nd Int. Conf. Emerg. Smart Technol. Appl. (eSmarTA)*, Ibb, Yemen, Oct. 2022, pp. 1–7, doi: 10.1109/eSmarTA56775.2022.9935361.

[35] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, "On the impact of refactoring operations on code quality metrics," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Victoria, BC, Canada, Sep. 2014, pp. 456–460, doi: 10.1109/ICSME.2014.73.

[36] A. Almogahed and M. Omar, "Refactoring techniques for improving software quality: Practitioners' perspectives," *J. Inf. Commun. Technol.*, vol. 20, pp. 511–539, 2021, doi: 10.32890/jict2021.20.4.3.

[37] M. Alshayeb, H. Al. Jamimi, and K. O. Elish, "Empirical taxonomy of refactoring methods for aspect-oriented programming," *J. Softw., Evol. Process*, vol. 25, no. 1, pp. 1–25, 2013, doi: 10.1002/smr.544.

[38] M. Abebe and C. Yoo, "Trends, opportunities and challenges of software refactoring: A systematic literature review," *Int. J. Softw. Eng. Appl.*, vol. 8, no. 6, pp. 299–318, 2014. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=84ae6851e5b130906091f604a32d6f940cd8b2ec

[39] R. S. Bashir, S. P. Lee, C. C. Yung, K. A. Alam, and R. W. Ahmad, "A methodology for impact evaluation of refactoring on external quality attributes of a software design," in *Proc. Int. Conf. Frontiers Inf. Technol. (FIT)*, Islamabad, Pakistan, Dec. 2017, pp. 183–188, doi: 10.1109/FIT.2017.00040.

[40] K. O. Elish and M. Alshayeb, "A classification of refactoring methods based on software quality attributes," *Arabian J. Sci. Eng.*, vol. 36, no. 7, pp. 1253–1267, Nov. 2011, doi: 10.1007/s13369-011-0117-x.

[41] K. O. Elish and M. Alshayeb, "Using software quality attributes to classify refactoring to patterns," *J. Softw.*, vol. 7, no. 2, pp. 408–419, Feb. 2012, doi: 10.4304/jsw.7.2.408-419.

[42] R. Malhotra and A. Chug, "An empirical study to assess the effects of refactoring on software maintainability," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, Sep. 2016, pp. 110–117, doi: 10.1109/ICACCI.2016.7732033.

[43] R. Malhotra and J. Jain, "Analysis of refactoring effect on software quality of object-oriented," in *Proc. Int. Conf. Innov. Comput. Commun.* Singapore: Springer, 2019, pp. 197–212, doi: 10.1007/978-981-13-2354-6.

[44] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 535–546, doi: 10.1145/2950290.2950317.

[45] B. Du. Bois, and T. Mens, "Describing the impact of refactoring on internal program quality," in *Proc. Int. Workshop Evol. Large-Scale Ind. Softw. Appl.*, 2003, pp. 1–9. [Online]. Available: http://plg2.math.uwaterloo.ca/~migod/papers/2003/ELISAproceedings.pdf#page=39

[46] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring–improving coupling and cohesion of existing code," in *Proc. 11th Work. Conf. Reverse Eng.*, 2004, pp. 144–151, doi: 10.1109/WCRE.2004.33.

[47] A. Almogahed, M. Omar, N. H. Zakaria, and A. Alawadhi, "Software security measurements: A survey," in *Proc. Int. Conf. Intell. Technol., Syst. Service Internet Everything (ITSS-IoE)*, Hadhramaut, Yemen, Dec. 2022, pp. 1–6, doi: 10.1109/ITSS-IoE56359.2022.9990968.

[48] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Inf. Softw. Technol.*, vol. 58, pp. 231–249, Feb. 2015, doi: 10.1016/j.infsof.2014.08.002.

[49] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Inf. Softw. Technol.*, vol. 83, pp. 1–21, Mar. 2016, doi: 10.1016/j.infsof.2016.11.009.

[50] A. Almogahed, M. Omar, and N. H. Zakaria, "Empirical studies on software refactoring techniques in the industrial setting," *Turkish J. Comput. Math. Educ.*, vol. 12, no. 3, pp. 1705–1716, 2021, doi: 10.17762/turcomat.v12i3.995.

[51] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, Jul. 2014, doi: 10.1109/TSE.2014.2318734.

[52] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial C# software," *Sci. Comput. Program.*, vol. 102, pp. 44–56, May 2015, doi: 10.1016/j.scico.2014.12.002.

[53] A. Ouni, M. Kessentini, H. Sahraoui, M. Ó. Cinnéide, K. Deb, and K. Inoue, "A multi-objective refactoring approach to introduce design patterns and fix anti-patterns," in *Proc. 1st North Amer. Search Based Softw. Eng. Symp., NASBASE*, 2015, pp. 1–15. [Online]. Available: https://kir.ics.es.osaka-u.ac.jp/lab-db/betuzuri/archive/990/990.pdf

[54] F. B. Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *Proc. 4th Int. Conf. Softw. Quality*, Oct. 1994, pp. 3–5.

[55] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*. Upper Saddle River, NJ, USA: Prentice-Hall, 1994.

[56] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002, doi: 10.1109/32.979986.

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

[58] R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review," *Empirical Softw. Eng.*, vol. 20, no. 3, pp. 640–693, Jun. 2015, doi: 10.1007/s10664-013-9291-7.

[59] V. Pham, C. Lokan, and K. Kasmarik, "A better set of object-oriented design metrics for within-project defect prediction," in *Proc. Eval. Assessment Softw. Eng.*, Apr. 2020, pp. 230–239, doi: 10.1145/3383219.3383243.

[60] P. K. Goyal and G. Joshi, "QMOOD metric sets to assess quality of Java program," in *Proc. Int. Conf. Issues Challenges Intell. Comput. Techn. (ICICT)*, Ghaziabad, India, Feb. 2014, pp. 520–533, doi: 10.1109/ICICICT.2014.6781337.

[61] C. M. S. Couto, H. Rocha, and R. Terra, "A quality-oriented approach to recommend move method refactorings," in *Proc. XVII Brazilian Symp. Softw. Quality*, Oct. 2018, pp. 1–10, doi: 10.1145/3275245.3275247.

[62] *Banking System Management*. Accessed: Aug. 25, 2020. [Online]. Available: https://github.com/derickfelix/BankApplication

[63] *Source Code & Projects*. Accessed: Aug. 18, 2019. [Online]. Available: https://code-projects.org/library-management-system-in-java-with-source-code/

[64] *Payroll Management System*. Accessed: Feb. 11, 2021. [Online]. Available: https://cutt.ly/Xn36RLP

[65] *JHotDraw Files*. Accessed: Jul. 25, 2019. [Online]. Available: https://sourceforge.net/projects/jhotdraw/files/JHotDraw/5.2/

[66] *Jedit Files*. Accessed: Nov. 25, 2019. [Online]. Available: https://sourceforge.net/projects/jedit/files/jedit/5.5.0/

[67] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: A roadmap," in *Proc. Conf. Future Softw. Eng.*, May 2000, pp. 345–355. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/336512.336586

[68] *Eclipse Foundation*. Accessed: Jul. 25, 2019. [Online]. Available: https://www.eclipse.org/downloads/

[69] E. L. G. Alves, M. Song, T. Massoni, P. D. L. Machado, and M. Kim, "Refactoring inspection support for manual refactoring edits," *IEEE Trans. Softw. Eng.*, vol. 44, no. 4, pp. 365–383, Apr. 2018, doi: 10.1109/TSE.2017.2679742.

[70] *Metrics 3—Eclipse Metrics Plugin Continued 'Again'*. Accessed: Aug. 5, 2019. [Online]. Available: https://github.com/qxo/eclipse-metrics-plugin

[71] A. Kaur and M. Kaur, "Analysis of code refactoring impact on software quality," in *Proc. MATEC Web Conf.*, 2016, pp. 1–15, doi: 10.1051/matecconf/20165702012.

[72] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2008, pp. 131–142, doi: 10.1145/1390630.1390648.

[73] N. Alsolami, Q. Obeidat, and M. Alenezi, "Empirical analysis of object-oriented software test suite evolution," *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 11, pp. 89–98, 2019, doi: 10.14569/IJACSA.2019.0101113.

[74] K. M. Eisenhardt, "Building theories from case study research," *Acad. Manage. Rev.*, vol. 14, no. 4, pp. 532–550, Oct. 1989. [Online]. Available: https://journals.aom.org/doi/abs/10.5465/AMR.1989.4308385

[75] P. P. Maglio and C. Lim, "Innovation and big data in smart service systems," *J. Innov. Manage.*, vol. 1, pp. 1–11, May 2016, doi: 10.24840/2183-0606_004.001_0003.

[76] M. Ketokivi and T. Choi, "Renaissance of case research as a scientific method," *J. Operations Manage.*, vol. 32, no. 5, pp. 232–240, Jul. 2014, doi: 10.1016/j.jom.2014.03.004.

[77] C. Lim, K. Kim, and P. P. Maglio, "Smart cities with big data: Reference models, challenges, and considerations," *Cities*, vol. 82, pp. 86–99, Dec. 2018, doi: 10.1016/j.cities.2018.04.011.

[78] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Cham, Switzerland: Springer, 2012.

[79] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Boston, MA, USA: Houghton Mifflin, 1979.

ABDULLAH ALMOGAHED received the B.S. degree in engineering and information technology from Taiz University, Yemen, in 2009, and the M.S. degree in information technology and the Ph.D. degree in computer science with a major in software engineering from Universiti Utara Malaysia (UUM), Malaysia, in 2017 and 2021, respectively. He is currently a Postdoctoral Fellow of the Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia (UTHM), Johor Bahru, Malaysia. His research interests include software refactoring, empirical software engineering, software quality, software maintenance, security, applied machine learning, and wireless networks.

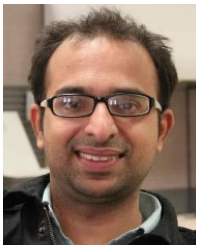
HAIRULNIZAM MAHDIN is currently an Associate Professor with the Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia. He has published more than 100 journal and conference papers indexed by various indexes, including WOS, Scopus, and Google Scholar. His research interests include the Internet of Things (IoT), data management, and artificial intelligence (AI). He has an extensive background in computer science and has been actively involved in many conferences internationally, serving in various capacities including chairman, program committee, general co-chair, and vice-chair.

MAZNI OMAR received the Ph.D. degree from Universiti Teknologi MARA, Malaysia. Her Ph.D. thesis was on the empirical studies of agile methodology in humanistic aspects. She is currently an Associate Professor with the School of Computing (SOC), College of Arts and Sciences, Universiti Utara Malaysia (UUM), where she is a Research Fellow of the Institute for Advanced and Smart Digital Opportunities (IASDO). She has published several articles in Scopus and indexed journals, conference papers, and other publications, such as the book of chapters and technical reports. She managed to secure several research grants from the university, national, international, and industry grants. Her research interests include software engineering, knowledge management, and data mining.

NUR HARYANI ZAKARIA received the Ph.D. degree in computing science from Newcastle University, U.K. She is currently an Associate Professor with the School of Computing (SoC), College of Arts and Sciences, Universiti Utara Malaysia (UUM). She has published several articles in Scopus and indexed journals, conference papers, and other publications, such as the book of chapters and technical reports. Besides that, she involves in several research and consultation activities from the university, national, international, and industry grants. Her research interests include usable security, information security, cybersecurity, and computer forensics. She is an Editorial Board Member of the *Journal of Information and Communication Technology* (JICT).

SALAMA A. MOSTAFA received the B.Sc. degree in computer science from the University of Mosul, Iraq, in 2003, and the M.Sc. and Ph.D. degrees in information and communication technology from Universiti Tenaga Nasional (UNITEN), Malaysia, in 2011 and 2016, respectively. He is currently a Senior Lecturer with the Department of Software Engineering, Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia (UTHM). His research interests include soft computing, data mining, software agents, and intelligent autonomous systems.

SALMAN A. ALQAHTANI (Member, IEEE) is currently a Professor with the Department of Computer Engineering, King Saud University, Riyadh. His research interests include 5G networks, broadband wireless communications, radio resource management for 4G and beyond networks (call admission control, packet scheduling, and radio resource sharing techniques), cognitive and cooperative wireless networking, small cell and heterogeneous networks, self-organizing networks, SDN/NFV, 5G network slicing, smart grids, the intelligent IoT solutions for smart cities, dynamic spectrum access, co-existence issues on heterogeneous networks in 5G, industry 4.0 issues, the Internet of Everything, mobile edge and fog computing, cyber sovereignty, performance evaluation and analysis of high-speed packet switched networks, system model and simulations, and integration of heterogeneous wireless networks. Mainly his focus is on the design and optimization of 5G MAC layers, closed-form mathematical performance analysis, energy-efficiency, and resource allocation and sharing strategies. He has been authored two scientific books and authored/coauthored around 76 journals and conference papers in the topic of his research interests, since 2004. He serves as a reviewer for several national and international journals.



PRANAVKUMAR PATHAK received the Ph.D. degree in AI. He is currently a Renowned Researcher and an Academician from Montreal, Canada. With extensive experience in teaching, research, and academia, he has showcased his expertise across diverse Computer Science courses. He remains dedicated to professional development, staying updated with the latest advancements in the field. He holds certifications in AI, IT, and networking, including Cisco and cloud computing. His research interests include artificial intelligence, big data, and networking.

SHAZLYN MILLEANA SHAHARUDIN received the bachelor's degree in industrial mathematics and the Ph.D. degree in statistics from the University of Technology Malaysia (UTM), Malaysia, in 2010 and 2017, respectively. She is currently a Senior Lecturer with the Department of Mathematics, Faculty of Science and Mathematics, Universiti Pendidikan Sultan Idris (UPSI), Malaysia. Her research interests include dimensions reduction methods applied to climate informatics, which analyzes huge climate-related datasets using data mining techniques. Additionally, she is focusing her current research on the development of a prediction model for hydrological, environmental, microbiological, and educational data. The majority of the research is using historical data and deals with high dimensional data. Her approach to solving the problems in the study involves the use of multivariate analysis approaches combined with time series methods and machine learning techniques. She have a wide range of experience in multivariate analysis, such as Principal Component Analysis (PCA) and Factor Analysis (FA), for time series analysis she uses Singular Spectrum Analysis (SSA), and for machine learning techniques she specialize in Support Vector Machine (SVM), Relevant Vector Machine (RVM), and Random Forest. As a Statistician with a view toward data science, she believes that the skills she have acquired throughout her research experience would be a valuable asset to the research committee.

RAHMAT HIDAYAT (Member, IEEE) received the M.Sc.IT. degree in information technology (system science and management) from the National University of Malaysia, Bangi, Malaysia, in 2013. He is currently pursuing the Ph.D. degree in information technology from Universiti Tun Hussein Onn, Malaysia. He joined the Department of Information Technology, Politeknik Negeri Padang, Indonesia, as a Lecturer, in 2015. His research interests include data classification, bioinformatics, machine learning, and deep learning.