JOURNAL OF INFORMATION AND COMMUNICATION TECHNOLOGY
http://e-journal.uum.edu.my/index.php/jict

How to cite this article:
Almogahed, A., & Omar, M. (2021). Refactoring techniques for improving software quality: A practitioners' perspectives. *Journal of Information and Communication Technology, 20*(4), 511-539. https://doi.org/10.32890/jict2021.20.4.3

Refactoring Techniques for Improving Software Quality: Practitioners' Perspectives

[1]Abdullah Almogahed & [2]Mazni Omar
[1]Department of Software Engineering, Taiz University, Yemen
[1&2]School of Computing, Universiti Utara Malaysia, Malaysia

abdullah.almogahed@outlook.com
mazni@uum.edu.my

Received: 30/8/2020 Revised: 12/1/2021 Accepted: 10/3/2021 Published: 27/9/2021

ABSTRACT

Refactoring is a critical task in software maintenance and is commonly applied to improve system design or to cope with design defects. There are 68 different types of refactoring techniques and each technique has a particular purpose and effect. However, most prior studies have selected refactoring techniques based on their common use in academic research without obtaining evidence from the software industry. This is a shortcoming that points to the existence of a clear gap between academic research and the corresponding industry practices. Therefore, to bridge this gap, this study identified the most frequently used refactoring techniques, the commonly used programming language, and methods of applying refactoring techniques in the current practices of software refactoring among software practitioners in the industry, by using an online survey.

The findings from the survey revealed the most used refactoring techniques, programming language, and the methods of applying the refactoring techniques. This study contributes toward the improvement of software development practices by adding empirical evidence on software refactoring used by software developers. The findings would be beneficial for researchers to develop reference models and software tools to guide the practitioners in using these refactoring techniques based on their effect on software quality attributes to improve the quality of the software systems as a whole.

Keywords: Exploratory study, software refactoring, survey, refactoring techniques.

INTRODUCTION

The codes and associated documentation of software systems always undergo modifications because of a problem or the necessity for improvement (L'Erario & Thomazinho, 2020; Rajlich, 2014). Therefore, software maintenance has become an integral component of software development and management (Rehman et al., 2018; Sun et al., 2015). The maintenance process includes the essential tasks to preserve the integrity of the existing software system (Ghannem et al., 2017). These modifications are incremental and aim to either update some functionalities, correct some design flaws, or fix some bugs (Ghannem et al., 2017; L'Erario & Thomazinho, 2020). These software maintenance activities become more complex when the size of the system and the number of requirements increase at any one time (Ghannem et al., 2017). Another term usually associated with software maintenance is software evolution (Godfrey & German, 2008; Rajlich, 2014). The term 'evolution' is defined as the "capability of software products to be evolved to continue to serve their customers in a cost-effective manner" (Ciraci & van den Broek, 2006; Cook et al., 2000). Therefore, software evolution is a subset of software maintenance activities. Software maintenance and evolution activities are inevitable due to requests generated for improvements and change (L'Erario & Thomazinho, 2020; Rajlich, 2014). Many studies have reported that software maintenance and evolution activities represent more than 80 percent of the total software development costs (Alizadeh et al., 2019; Fernández-Sáez et al., 2018; L'Erario & Thomazinho, 2020; Ouni et al., 2016). It has also been shown that software developers

typically spend around 60 percent of their time to understand the codes they are maintaining (Abid et al., 2020; Alizadeh et al., 2019). With the evolution of the software industry resulting in the growth and complexity of software day-by-day, software practitioners are acknowledging the significance of good quality software (Malhotra & Jain, 2019). Clearly, software developers need better ways to manage and reduce the growing complexity of software systems and improve their productivity.

Fortunately, the cost of software maintenance and evolution activities can be significantly reduced by the software refactoring process (Besker et al., 2018; Kaur & Singh, 2019; Mkaouer et al., 2014; Ouni et al., 2016). Refactoring is considered as a standard solution, which involves improving the design structure of the software while preserving its functionality (Alizadeh et al., 2019). For this purpose, 68 basic types of refactoring techniques have been proposed and categorized into six categories based on their similarity in purpose (Fowler et al., 2002; Fowler & Beck, 2019). Each refactoring technique comes with the motivation to use it and the explanation on how to perform each technique (Elish & Alshayeb, 2011; Fowler et al., 2002; Fowler & Beck, 2019; Rochimah et al., 2015).

Many studies have addressed the impact of different refactoring techniques on software quality attributes. The findings reveal that the refactoring techniques do not always improve all aspects of software quality (Al-Dallal & Abdin, 2018; Almogahed et al., 2018; Almogahed et al., 2019; Kaur & Singh, 2019). Different refactoring techniques have a diverse (and sometimes opposite) impact on software quality (Al-Dallal & Abdin, 2018; Almogahed et al., 2018; Almogahed et al., 2019; Kaur & Singh, 2019). Therefore, there is no consensus among researchers regarding the impact of the refactoring techniques on software quality. The inconsistent or contradictory results concerning the impact of refactoring techniques on software quality have become challenges for developers when they use the refactoring techniques to improve software quality. Chaparro et al. (2014) posited that assessing the pros and cons of a refactoring technique is very challenging for software developers and this becomes even more challenging when one refactoring technique conflicts with another. Additionally, Nyamawe et al. (2018) indicated that selecting the best refactoring technique from some potential techniques to remove a design flaw

is challenging for software developers. According to Nyamawe et al. (2019), it is often challenging to determine which kind of refactoring technique should be applied. In other words, it is very challenging for software practitioners to select appropriate refactoring techniques to improve software quality (Al-Dallal & Abdin, 2018; Almogahed et al., 2018; Almogahed et al., 2019; Kaur & Singh, 2019). Evaluating the pros and cons of each refactoring technique involves a great deal of effort and time, and this in turn, leads to increased maintenance costs. However, according to Kaur and Singh (2019), most of the previous studies have not provided any valid justification when choosing refactoring techniques. The issues of refactoring techniques that software practitioners most or least frequently perform as part of their daily maintenance tasks have been selected by very few studies (Kaur & Singh, 2019). This observation indicates the gap between refactoring techniques examined by academic researchers and refactoring techniques actually applied by industry practitioners (Al-Dallal & Abdin, 2018; Kaur & Singh, 2019). Therefore, it is suggested for researchers to involve industry professionals when conducting a survey to select the most frequently used refactoring activities (Al-Dallal & Abdin, 2018; Kaur & Singh, 2019).

This study identifies the most frequent refactoring techniques applied by software practitioners at present. In addition, it distinguishes the commonly used programming language and methods of applying refactoring techniques. For these purposes, an exploratory study was conducted using the quantitative approach, i.e., an online survey, to obtain insights from the practitioners on their current software refactoring practices. The identification of the most frequently used refactoring techniques by software practitioners would enable researchers to focus on the techniques in their studies. In this way, solutions to mitigate the challenges faced by software practitioners in selecting appropriate refactoring techniques that can improve the quality of software systems and eliminate design flaws can be proposed. As a result, the effort and time spent by software practitioners to assess the pros and cons of each refactoring technique can be saved, which in turn, will reduce maintenance costs. In addition, the common programming languages used for refactoring by software practitioners can be identified. Researchers will therefore be able to develop techniques and tools to apply automatically in refactoring techniques based on the common programming languages.

The remainder of this paper is structured as follows. The Related Works Section describes the literature review, the Methodology Section explains the methodology used for the research, the Results and Findings Section reports the results and findings, the Discussion Section deliberates the findings, and lastly, the Conclusion Section concludes and recommends future research.

RELATED WORKS

The term 'refactoring' was first coined by Opdyke in his PhD thesis in the context of object-oriented programming (Opdyke, 1992). Refactoring has been defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code and yet, improves its internal structure" (Fowler et al., 2002; Fowler & Beck, 2019). In other words, refactoring is a process that makes a change in the internal structure of the software in order to make it simpler to understand and cheaper to modify without altering the software's behavior (Al-Dallal, 2015; Choi et al., 2018; Kaur & Singh, 2017). This means the internal structure of a software can be improved by refactoring without creating any new functionality (Alves et al., 2016). This strategy can be achieved by restructuring classes, methods, and variables, mainly to assist in modifications and extensions in the future (Elish & Alshayeb, 2011; Fowler et al., 2002; Fowler & Beck, 2019). This restructuring is utilized to enhance several software qualities attributes, including extensibility, maintainability, reusability, and understandability (Fowler et al., 2002; Fowler & Beck, 2019; Wang et al., 2015). Fowler et al. (2002) proposed 68 refactoring techniques in their refactoring catalog grouped into six categories as shown in Table 1.

Table 1

Categories of Refactoring with Their Relevant Techniques

Category	Refactoring Techniques
Composing Methods	1) Extract Method 2) Inline Method 3) Replace Temp with Query 4) Replace Method with Method Object 5) Substitute Algorithm 6) Extract Variable 7) Inline Temp 8) Split Temporary Variable 8) Remove Assignments to Parameters 9) Introduce Explaining Variable.
Simplifying Conditional Expressions	1) Decompose Conditional 2) Replace Conditional with Polymorphism 3) Introduce Null Object 4) Introduce Assertion 5) Consolidate Conditional Expression 6) Consolidate Duplicate Conditional Fragments 7) Remove Control Flag 8) Replace Nested Conditional with Guard Clauses.
Moving Features between Objects	1) Move Method 2) Move Field 3) Extract Class 4) Inline Class 5) Hide Delegate 6) Remove Middleman 7) Introduce Foreign Method 8) Introduce Local Extension.
Organizing Data	1) Replace Data Value with Object 2) Replace Array with Object 3) Duplicate Observed Data 4) Change Bidirectional Association to Unidirectional 5) Encapsulate Field 6) Encapsulate Collection 7) Replace Type Code with Class 8) Replace Type Code with Subclasses 9) Replace Type Code with State/Strategy 10) Self Encapsulate Field 11) Change Value to Reference 12) Change Reference to Value 13) Change Unidirectional Association to Bidirectional 14) Replace Magic Number with Symbolic Constant 15) Replace Subclass with Fields 16) Replace Record with Data Class.

(continued)

Category	Refactoring Techniques
Dealing with Generalization	1) Pull Up Method 2) Push Down Method 3) Push Down Field 4) Extract Subclass 5) Extract Superclass 6) Extract Interface 7) Collapse Hierarchy 8) Form Template Method 9) Replace Inheritance with Delegation 10) Replace Delegation with Inheritance 11) Pull Up Field 12) Pull Up Constructor Body
Simplifying Method Calls	1) Rename Method 2) Remove Parameter 3) Replace Parameter with Explicit Methods 4) Preserve Whole Object 5) Replace Parameter with Method 6) Introduce Parameter Object 7) Remove Setting Method 8) Add Parameter 9) Separate Query from Modifier 10) Parameterize Method 11) Hide Method 12) Replace Constructor with Factory Method 13) Replace Error Code with Exception 14) Replace Exception with Test 15) Encapsulate Downcast.

The refactoring techniques in the 'Composing Methods' category are used to package codes effectively. Typically, large methods cause most of the problems as they often include numerous information that makes them complex and hard to understand. The refactoring techniques in the 'Composing Methods' category streamline methods, eliminate code duplication, and facilitate future improvements. The refactoring techniques in the 'Simplifying Conditional Expressions' category are used to simplify complicated conditional statements. The refactoring techniques in the 'Moving Features between Objects' category are used to distribute functionalities in a perfect way among different classes in case these functionalities are not distributed in an appropriate way. These refactoring techniques demonstrate ways to move functionalities in a safe way between classes and generate new classes. The refactoring techniques in the 'Organizing Data' category help to deal with data in an easy way. In other words, they help to handle data and hide information from public access. The refactoring techniques in the 'Dealing with Generalization' category deal with moving methods around the inheritance hierarchy. They are mainly attached to moving functionalities around a hierarchy of the class inheritance, producing new classes, and replacing inheritance among classes with a delegation and vice versa. The refactoring techniques

in the 'Simplifying Method Calls' category help to make the calling of the methods simpler to understand. In turn, this leads to simplifying the interaction among classes.

Regarding the most commonly used refactoring techniques, a few prior studies have discussed the selection of the refactoring techniques. Kim et al. (2014) studied the benefits and challenges of refactoring at the Microsoft company by using three complemental methods (interview, survey, and quantitative analysis) for the version history of Windows 7. The overall results showed that the benefits of refactoring included an improvement of the quality, while its challenges were costs and risks. Additionally, Kim et al. (2014) identified seven refactoring techniques as shown in Table 2 that are commonly used by the software practitioners at the Microsoft company. Ouni et al. (2015) claimed that there are 11 commonly used refactoring techniques in the practices as shown in Table 2. Al-Dallal (2015) conducted a systematic literature review (SLR) and identified eight refactoring techniques commonly used by the reviewed studies. Mariani and Vergilio (2017) performed an SLR and detected 14 refactoring techniques commonly used. Al-Dallal and Abdin (2018) carried out an SLR and determined the ten most used refactoring techniques in the studies they reviewed. Kaur and Singh (2019) conducted a systematic mapping study (SMS) of previous works and reported the ten most used refactoring techniques. Lacerda et al. (2020) carried out an SLR and reported the top commonly used refactoring techniques in the previous studies reviewed. Table 2 presents the most frequently used refactoring techniques based on the previous studies (Al-Dallal, 2015; Al-Dallal & Abdin, 2018; Kaur & Singh, 2019; Kim et al., 2014; Lacerda et al., 2020; Mariani & Vergilio, 2017; Ouni et al., 2015) and the refactoring coverage in percentage based on these studies. The coverage is calculated base on Equation 1 as follows:

$$Coverage = (No.of\ studies\ reporting\ the\ technique) / (the\ total\ number\ of\ studies) * 100 \quad (1)$$

Table 2

The Most Frequently Used Refactoring Techniques Based on Previous Studies

No.	Common refactoring techniques	Coverage	SLR by Lacerda et al. (2020)	SMS by Kaur and Singh (2019)	SLR by Al-Dallal and Abdin (2018)	SLR by Mariani and Vergilio (2017)	Ouni et al. (2015)	SLR by Al-Dallal (2015)	Kim et al. (2014)
1	Add Parameter	14.3%	x	x	x	✓	x	x	x
2	Collapse Hierarchy	14.3%	x	x	x	✓	x	x	x
3	Encapsulate Field	42.9%	x	x	✓	✓	x	x	✓
4	Extract Class	71.4%	x	✓	✓	✓	✓	✓	x
5	Extract Subclass	28.6%	x	x	✓	x	✓	x	x
6	Extract Superclass	57%	x	✓	x	✓	✓	✓	x
7	Extract Method	100%	✓	✓	✓	✓	✓	✓	✓
8	Inline Class	42.9%	x	x	x	✓	✓	✓	x
9	Inline Method	28.6%	✓	x	x	x	x	x	✓
10	Introduce Null Object	14.3%	x	x	✓	x	x	x	x
11	Move Field	85.7%	✓	✓	✓	✓	✓	✓	x
12	Move Method	85.7%	✓	✓	✓	✓	✓	✓	x

(continued)

No.	Common refactoring techniques	Coverage	SLR by Lacerda et al. (2020)	SMS by Kaur and Singh (2019)	SLR by Al-Dallal and Abdin (2018)	SLR by Mariani and Vergilio (2017)	Ouni et al. (2015)	SLR by Al-Dallal (2015)	Kim et al. (2014)
13	Pull Up Field	71.4%	✓	✓	x	✓	✓	x	✓
14	Push Down Field	71.4%	✓	✓	x	✓	✓	x	✓
15	Pull Up Method	71.4%	x	✓	✓	✓	✓	✓	x
16	Push Down Method	42.8%	x	✓	x	✓	✓	x	x
17	Rename Method	57%	✓	✓	x	✓	x	x	✓
18	Remove Parameter	42.9%	✓	x	x	x	x	✓	✓
19	Replace Method with Method Object	14.3%	x	x	✓	x	x	x	x
20	Replace Conditional with Polymorphism	28.6%	✓	x	✓	x	x	x	x

Note. ✓*: A refactoring technique was reported by a study.*

X: A refactoring technique was not reported by a study.

It is noted that only the Extract Method had the highest consensus between academics (Al Dallal, 2015; Al-Dallal & Abdin, 2018; Kaur & Singh, 2019; Lacerda et al., 2020; Mariani & Vergilio, 2017; Ouni et al., 2015) industry research (Kim et al., 2014) with 100 percent coverage; while full consensus was missing on the other 19 refactoring techniques as shown in Table 2. This observation showed the gap between academics and industrial research on the refactoring techniques studied by academic researchers and those currently being applied by industry practitioners.

Since there are many different refactoring techniques, each of which has a specific purpose and effect (positive, negative, ineffective) on software quality, it is very challenging for software practitioners to evaluate the pros and cons of each refactoring technique and choose suitable refactoring techniques to improve software quality or remove design flaws (Chaparro et al., 2014; Nyamawe et al., 2019). The selection of improper refactoring techniques can lead to deterioration in the quality of software systems, which in turn, will increase maintenance costs. Empirical investigations are required on the relationship between each refactoring technique individually and the software quality attributes (Al-Dallal & Abdin, 2018; Almogahed et al., 2018; 2019; Kaur & Singh, 2019). Therefore, the refactoring techniques that should be investigated for their effects on software quality are those that are the most commonly used by software practitioners. Consequently, researchers can propose solutions, such as a reference model based on the results of these investigations. Such solutions can serve as a guideline to enable the software practitioners to have a better understanding of the impact of each refactoring technique on the software quality attributes and enable them to select a suitable refactoring technique to make improvements (Almogahed et al., 2018; 2019). Ultimately, efforts taken and time spent by software practitioners in assessing and choosing the right refactoring techniques can be saved, which in turn, will reduce maintenance costs.

METHODOLOGY

This section describes the exploratory study (using survey) conducted among software practitioners. The discussion in this section starts with the questionnaire design, and continues with the sampling, questionnaire testing, data collection, and response rate.

Questionnaire Design

The questionnaire was designed based on the guideline proposed by Gay et al. (2012). According to Gay et al. (2012), it is important that the questionnaire is attractive and brief, contains only items that relate to the study's objectives, collects demographic information as necessary, focuses on items based on single topics or ideas, defines and explains ambiguous terms, words the questions clearly, avoids leading questions, organizes items from general to specific, and keeps items and response options together. Moreover, careful attention must be given to the length of the questionnaire, as well as the length, content, order, and type of individual questions. The questionnaire comprised two sections: (i) demographic data; and (ii) the current refactoring techniques practiced. The questionnaire was designed by using Google Form (http://www.googledocs.com), consisting of 18 questions organized into two main sections. The following subsections describe the two main sections of the questionnaire.

Demographic Information

It is quite common to begin the questionnaire by gathering information related to the demographic data to identify and understand the respondents' profile. This demographic section included respondents' details, such as their job function in the organization, their years of experience, and their familiarity with refactoring techniques. The questions in this section were of two types: i) in the form of a checkbox, whereby the respondents could choose one or multiple answers; and ii) in the form of a direct question for which the respondents could type their answer.

The Current Practices of Refactoring Techniques

This section aims to identify the current practices related to the most frequently used refactoring techniques among software practitioners. Precisely, there were seven questions relating to the application of refactoring techniques. These questions were in the form of check-box questions as well as open-ended questions for which the responses must be typed. Table 3 presents the questions and types of response.

Table 3

The Questions and Types of Response to Current Practices for Refactoring

No.	Questions	Types of Response
1	The methods used to apply the refactoring techniques.	Checkbox
2	Asking to mention other methods if they have not been included in the methods presented.	Open-ended
3	Used programming languages.	Checkbox
4	Asking to report other programming languages if they have not been included in programming languages presented.	Open-ended
5	Respondents' opinions as to whether or not they agree on the 20 most commonly used refactoring techniques shown in Table 2.	Yes/No
6	Identifying other than the 20 refactoring techniques used in practice and not mentioned in the questionnaire.	Open-ended
7	Requiring respondents to indicate whether they have any difficulties with the use of refactoring techniques and, if any, to specify those difficulties.	Open-ended

Sampling

The target population for this study was the software practitioners who apply the software refactoring techniques. The main constraint in selecting these software practitioners was that they had a tight work schedule and could not be reached easily. Due to this limitation and the

fact that not all software practitioners were using software refactoring techniques, this study used non-probability sampling, i.e., purposive (judgmental) sampling, which is considered appropriate when only a limited number or category of people can be approached (Sekaran & Bougie, 2016). It involves the selection of a unique sample with specific features important for the study (Nardi, 2003). The sample comprised software practitioners in Malaysia, Saudi Arabia, and Yemen. Furthermore, the sample could definitely meet the objectives of the study since they are chosen based on specific characteristics (Zikmund et al., 2010).

The sample software practitioners were obtained by contacting software practitioners working in private and public companies. The target respondents were the persons responsible for system maintenance, system renovation, system re-development (enhancement), or system development (new development). The sample encompassed 103 software practitioners, which were considered sufficient for this study. This corresponded to Roscoe's (1975) rule of thumb, whereby a sufficient sample size is between 30 and 500. The minimum sample size of 30 is acceptable for statistical analysis (Fisher, 2007; Sekaran, 2003).

Questionnaire Testing

The purpose of this questionnaire testing is for face and content validity of the questions. The pre-testing of content validity was to ensure that the correct quality data were provided in the questionnaire, while face validity was used to test whether or not the instrument measures what it was designed to measure. The pre-test identified whether the survey had any problems, whether it was too difficult to understand, whether the wording of the questions was ambiguous, or whether it could lead to biased responses.

The questionnaire was subjected to two rounds of analysis and revision, to ensure not only that the contents were detailed and relevant, but also that the design was user-friendly, the instructions were straightforward, and the language was comprehensible. These components were tested before the questionnaire was distributed to the selected sample. Ikart (2019) reported that the minimum number of experts to review a questionnaire is two or three. In this study, the pre-

test was administered to six respondents drawn from the population of interest. After the review of the questionnaire, slight changes were made to some of the questions to enhance understandability and readability. Table 4 summarizes the questionnaire pre-testing carried out in two rounds by the six respondents, three in each round.

Table 4

Questionnaire Pre-Testing

Round	ID	Field of Specialization	Results/Outcomes
First Round	A1	MSc in Information Technology	The study overview was modified to include the background and accurately explain the purpose of the questionnaire.
	B1	MSc in Information Technology	Some questions have been altered and updated in the demographics section
	C1	BSc Software Engineering	Adding some items to the programming language used and methods of performing the refactoring techniques.
Second Round	A2	BSc Software Engineering	It was all clear, easy, and understandable.
	B2	BSc Software Engineering	It was all clear, easy, and understandable.
	C2	BSc in Information Technology	It was all clear, easy, and understandable.

Data Collection and Response Rate

The main purpose of data collection is to gather data from the representative sample. The online survey created by Google Form was used for the data collection. The questionnaire link was sent online to target respondents through emails and social media networks. The respondents were given eight weeks to fill in the online questionnaire. Table 5 indicates the total number of questionnaires that were

distributed to the respondents and their response rate. The unreturned questionnaires were labeled as lost, while the incomplete questionnaires were considered as rejected due to incomplete answers and excluded from data analysis. The completed questionnaires were labeled as usable and included in data analysis. Based on Table 5, the response rate for this study was 31.07 percent. This denoted that the completed questionnaires could be analyzed since Saunders et al. (2015) recommended that the reasonable average response rate is between 30.0 percent – 40.0 percent.

Table 5

Response Rate of the Questionnaire

Description	Software Practitioners	Rate (%)
Sent	103	100.0%
Lost	61	59.22%
Received	42	40.78%
Usable/ Complete online survey	32	31.07%
Rejected/ Incomplete online survey	4	3.88%
Not familiar with refactoring	6	5.8%

Software practitioners are very busy individuals and cannot be easily accessed. Indeed, not all software practitioners use software refactoring techniques because the refactoring process is not a simple task and involves high costs and huge risks. Due to these limitations, the number of software practitioners who are familiar with refactoring techniques was low (32 out of 103). However, the average response rate (31.07 %) was acceptable for statistical analysis (Fisher, 2007; Saunders et al., 2015; Sekaran, 2003).

RESULTS AND FINDINGS

This section discusses the findings obtained from the exploratory study. The first section presents the demographic and background information on the participating software practitioners. The second section describes the current practices of refactoring techniques.

Demographic Data

The demographic data were presented in terms of the respondents' core team role, main team assignment, and years of experience. Table 6 depicts that a majority of the respondents were software testers (37.5%), followed by software developers (34.4%), project managers (12.5%), requirement analysts (6.3%), software architects (3.1%), and networking (3.1%).

Table 6

Core Team Role in Organization

Core Team Role	Frequency	Percentage
Software Tester	12	37.5%
Software Developer	11	34.4%
Project Manager	4	12.5%
Requirement Analyst	2	6.3%
Software Architect	1	3.1%
Networking	1	3.1%
All of the above	1	3.1%
Total	32	100.0%

Regarding their main team assignment as shown in Table 7, most of the respondents were assigned to system maintenance (37.5%), followed by system - new development (34.4%), system renovation (12.5%), system re-development (9.4%), and network engineering (3.1%).

Table 7

Main Team Assignment in Organization

Main Team Assignment	Frequency	Percentage
System maintenance	12	37.5%
System - new development	11	34.4%
System renovation (enhancement)	4	12.5%
System re-development	3	9.4%
Network engineering	1	3.1%
All of the above	1	3.1%
Total	32	100.0%

Table 8 portrays information relating to the respondents' work experience. Most of the respondents (46.9%) had a range of 5 to 10 years of work experience, while 37.5 percent had work experience between 11 and 15 years, followed by 12.5 percent having work experience between three and five years. Only a few (3.1%) had a very long work experience, at more than 15 years.

Table 8

Work Experience

Experience	Frequency	Percentage
5–10 years	15	46.9%
11–15 years	12	37.5%
3–5 years	4	12.5%
More than 15 years	1	3.1%
Total	32	100.0%

This indicated that most of the software practitioners were responsible for software refactoring as they had work experience with systems maintenance, new system development, renovation, and re-development.

Current Practices of Refactoring Techniques

This section describes the survey findings related to the methods of applying refactoring techniques, the programming languages used, and the most frequently used refactoring techniques as of current.

Regarding the methods of applying the refactoring techniques, Table 9 reveals that most of the software practitioners (63.3%) used Eclipse Integrated Development Environment (IDE) to automatically apply the refactoring techniques, whilst 20 percent utilized other automatic tools, such as IntelliJ IDEA and RefactorIt tools. On the other hand, 10 percent of the software practitioners applied the refactoring techniques manually and 6.7 percent used semi-automatic methods.

Table 9

Methods of Applying the Refactoring Techniques

Method	Frequency	Percentage
Automatic using Eclipse IDE	19	63.3%
Automatic using other tools	6	20%
Manual	3	10%
Semi-automatic	2	6.7%
Total	30	100.0%

Regarding the programming languages used for refactoring, some of the respondents used more than one programming language (Table 10). The majority of the software practitioners (96.7%) applied Java. This was followed by C# (13.3%), C++ (10.0%), Python (10.0%), JavaScript (3.3%), HTML (3.3%), VB.Net (3.3%), and ASP.Net (3.3%).

Table 10

Programming Languages Used for Refactoring

Programming Language	Number	Percentage of Cases
Java	29	96.7%
C++	3	10.0%
C#	4	13.3%
Python	3	10.0%
JavaScript	1	3.3%
HTML	1	3.3%
VB.Net	1	3.3%
ASP.Net	1	3.3%
Total	42	139.9%

Note. **percentage of cases* is used to describe the data because it shows the percentage of the number of respondents who chose each item (it is appropriate for multi-responses question)

These findings indicated that Java was the most frequently used programming language for the refactoring techniques.

Regarding the most frequently used refactoring techniques currently used among the software practitioners, a list of 68 refactoring techniques categorized into six categories as shown in Table 1 was presented to the respondents. Additionally, a list of the 20 most used refactoring techniques in research and practice based on comprehensive literature reviews as portrayed in Table 2 was showed to the respondents.

Then, the respondents were asked whether they agreed or disagreed, based on their current use of the refactoring techniques, on the refactoring techniques included in the list of the 20 commonly used refactoring techniques. A 'Yes' answer referred to the agreement that the refactoring technique was commonly used in their current practice; while a 'No' answer meant the refactoring technique was not commonly used in their current practices. Table 11 shows the results obtained from the respondents.

Table 11

Voting of the Respondents on the Most Frequently Used Refactoring Techniques in their Current Practices

No.	Refactoring Technique	Yes	No	Total	Yes/ Agreement Percentage	No/ Disagreement Percentage
1	Add Parameter	16	16	32	50.0%	50.0%
2	Collapse Hierarchy	7	25	32	21.9%	78.1%
3	Encapsulate Field	30	2	32	93.75%	6.25%
4	Extract Class	29	3	32	90.6%	9.4%
5	Extract Subclass	29	3	32	90.6%	9.4%
6	Extract Superclass	20	12	32	62.5%	37.5%
7	Extract Method	28	4	32	87.5%	12.5%
8	Inline Class	17	14	31	54.8%	45.2%
9	Inline Method	18	13	31	58.1%	41.9%
10	Introduce Null Object	4	26	30	13.3%	86.7%
11	Move Field	26	4	30	86.7%	13.3%
12	Move Method	27	4	31	87.1%	12.9%
13	Pull Up Field	29	1	30	96.7%	3.3%
14	Push Down Field	28	2	30	93.3%	6.7%
15	Pull Up Method	29	1	30	96.7%	3.3%
16	Push Down Method	29	1	30	96.7%	3.3%
17	Rename Method	30	2	32	93.75%	6.25%
18	Remove Parameter	13	18	31	41.9%	58.1%
19	Replace Method with Method Object	3	28	31	9.68%	90.32%

(continued)

No.	Refactoring Technique	Yes	No	Total	Yes/ Agreement Percentage	No/ Disagreement Percentage
20	Replace Conditional with Polymorphism	4	27	31	12.9%	87.1%

Note. Highlighted rows refer to refactoring techniques that have received a low rate of agreement (below 50%).

In addition, the respondents were asked to specify other refactoring techniques they were using in their current practices that were not included in the list of the 20 frequently used refactoring techniques. The findings revealed that only one respondent (3.1%) specified two refactoring techniques (Transform Parameters and Convert Abstract Class to Interface) that were not in Fowler's (2002) catalog.

These findings clearly indicated the most frequently used refactoring techniques in current practices. 15 refactoring techniques obtained a high agreement rate (greater than or equal to 50%). However, five refactoring techniques obtained a low agreement rate (below 50%), as highlighted in Table 11, due to the lack of opportunities to apply them: 1) Remove Parameter (41.9%); 2) Collapse Hierarchy (21.9%); 3) Introduce Null Object (13.3%); 4) Replace Method with Method Object (9.68%); and 5) Replace Conditional with Polymorphism (12.9%). Overall, it is recommended that researchers focus on these 15 most used refactoring techniques to study and produce more effective solutions for the industry practitioners.

DISCUSSION

Refactoring is one of the fastest-growing, if not the fastest-growing area in software engineering research, due to its strategic importance in the process of improving and evolving software quality, which in turn, can reduce maintenance costs. However, most academic research have not explored the challenges that software practitioners face in refactoring software. Researchers have selected refactoring techniques and methods applicable to them subjectively, or on the basis of literature review, without obtaining evidence from the software industry. As a result, the existing gap between academic research and the software

industry prevents the software industry practitioners from exploiting the benefits of well-researched refactoring techniques. Therefore, this study bridges this gap by exploring the most applied refactoring techniques, the programming languages used, and the methods used to apply refactoring techniques in current practices among software practitioners.

The findings revealed that 15 refactoring techniques as shown in Table 11 are commonly applied among the software practitioners. The logic behind applying the Extract Method is to easily understand the method, as overly long methods make it extremely difficult to understand – and even more difficult to change. The Inline Method is utilized when the method body is more obvious than the method itself. The Extract Method and Inline Method are commonly employed as they simplify methods, remove duplication of codes, and create opportunities for new improvements. The Extract Class is used when a class has many responsibilities that make it hard to understand. It is commonly applied as it helps to maintain adherence to the principle of single responsibility, therefore making the class more obvious and understandable. Inline Class is normally used when a class does almost nothing and is not responsible for anything, and no additional responsibilities are planned. The Move Method or Move Field is utilized when a method or field is used more in another class than in its own class. The Move Method and Move Field are commonly used to reduce dependence between classes by moving the method or field from the source class to the target class. The reason for applying Encapsulate Field is that it hides the data and restricts the accessibility of the data, therefore improving the security of software. The Add Parameter is employed if the method has insufficient data to execute certain actions. The Rename Method is utilized when the method name does not illustrate what the method is doing. The Add Parameter and Rename Method are normally used to make method calls simpler and easier to understand. This, in turn, streamlines the interaction interfaces between classes. The Extract Subclass, Extract Superclass, Pull Up Method, Push Down Method, and Push Down Field are refactoring techniques that deal with generalization and move methods/fields around the inheritance hierarchy. They are mainly attached to moving functionalities around a hierarchy of the class inheritance, producing new classes, and replacing inheritance among classes with a delegation and vice versa. As a result, they make the inheritance hierarchy more organized and understandable.

On the other hand, different programming languages are used in the refactoring process as presented in Table 10. However, the Java programming language is the most used in the refactoring process in current practices. This because Java is the most commercially important and a recent object-oriented language, and the most widely used platform by industry and academia. In addition, different methods have been used to perform the refactoring techniques as shown in Table 9. Automatically using the Eclipse IDE method is the most common method. Eclipse IDE is one of the most popular IDEs that supports automated refactoring and is a widely used refactoring tool. It is, however, up to the software developers to find and know the refactoring technique to apply. In other words, Eclipse IDE does not provide full automation to identify opportunities for the use of refactoring techniques or full automation to perform common refactoring techniques.

The stockholders can benefit from the findings obtained from this study in the improvement of the software refactoring process that takes into consideration the most commonly used refactoring techniques in current practices. This will contribute to overcoming challenges faced by software practitioners.

CONCLUSION

This study addressed the issue of industry practitioners not being involved or considered in previous studies on refactoring techniques. This exploratory study using the online survey method was conducted among software practitioners to identify the most frequently used refactoring techniques, the programming language commonly used, and methods of applying the refactoring techniques in the current practices of software refactoring. The findings revealed the 15 most commonly used refactoring techniques in current practices. Additionally, Java was the most commonly used programming language (96.7%) in software development and refactoring. Automatically using Eclipse IDE was the most common method (63.3%) to apply the refactoring techniques.

This study provides evidence from the industry related to the current practices of refactoring and will be beneficial for academicians and the industry. Researchers can investigate the effect of the identified refactoring techniques individually, or in combination, on the software quality attributes to produce more productive results for the software industry. Additionally, it is recommended that reference models using the identified refactoring techniques be developed to guide the software developers to select suitable refactoring techniques based on their effect on software quality attributes. Moreover, via these empirical results, researchers can develop techniques and software tools that can automatically identify opportunities of these common refactoring techniques, thus applying them more efficiently.

Future research can conduct empirical investigations on the impact of the 15 commonly used refactoring techniques on internal quality attributes, such as abstraction, cohesion, coupling, complexity, encapsulation, and messaging, and the estimated external quality attributes, for instance effectiveness, reusability, understandability, and security, to identify the pros and cons of each refactoring technique.

ACKNOWLEDGMENT

The authors would like to gratefully thank all the participants of this study for their help and cooperation.

REFERENCES

Abid, C., Alizadeh, V., Kessentini, M., Ferreira, N., & Dig, D. (2020). 30 years of software refactoring research: A systematic literature review. IEEE Transactions on Software Engineering, 1(1), 1–24.

Al-Dallal, J. (2015). Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology*, *58*, 231–249. https://doi.org/10.1016/j.infsof.2014.08.002

Al-Dallal, J., & Abdin, A. (2018). Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, *44*(1), 44–69. https://doi.org/10.1109/TSE.2017.2658573

Alizadeh, V., Kessentini, M., Mkaouer, W., Ocinneide, M., Ouni, A., & Cai, Y. (2019). Interactive and dynamic multi-objective software refactoring recommendations. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software.* 1–30. IEEE Computer Society.

Almogahed, A., Omar, M., & Zakaria, N. H. (2018, July). Impact of software refactoring on software quality in the industrial environment: A review of empirical studies. In *Proceedings of Knowledge Management International Conference (KMICe)*, *25 –27 July 2018*. Miri Sarawak, Malaysia.

Almogahed, A., Omar, M., & Zakaria, N. H. (2019). Categorization refactoring techniques based on their effect on software quality attributes. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, *8*(8S), 439–445.

Alves, E. L. G., Massoni, T., Duarte, P., & Machado, D. L. (2016). Test coverage of impacted code elements for detecting refactoring faults: An exploratory study. *Journal of Systems and Software*, *0*, 1–16. https://doi.org/10.1016/j.jss.2016.02.001

Besker, T., Martini, A., & Bosch, J. (2018). Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software*, *135*, 1–16.

Chaparro, O., Bavota, G., Marcus, A., & Penta, M. Di. (2014, September). On the impact of refactoring operations on code quality metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution* (pp. 456–460). https://doi.org/10.1109/ICSME.2014.73

Choi, E., Fujiwara, K., Yoshida, N., & Hayashi, S. (2018). A survey of refactoring detection techniques based on change history analysis. arXiv preprint arXiv:1808.02320.

Ciraci, S., & van den Broek, P. (2006, January). Evolvability as a quality attribute of software. In *The International ERCIM Workshop on Software Evolution* (pp. 29–31).

Cook, S., Ji, H., & Harrison, R. (2000). *Software evolution and software evolvability*. University of Reading, UK. 1–12.

Elish, K. O., & Alshayeb, M. (2011). A classification of refactoring methods based on software quality attributes. *Arabian Journal*

for Science and Engineering, *36*(7), 1253–1267. https://doi.org/10.1007/s13369-011-0117-x

Fernández-Sáez, A. M., Chaudron, M. R., & Genero, M. (2018). An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles. *Empirical Software Engineering*, *23*(6), 1–65.

Fisher, C. M. (2007). *Researching and writing a dissertation: A guidebook for business students.* England: Prentice Hall.

Fowler, M., Beck, K. (2019*). Refactoring: Improving the design of existing code* (2nd ed.). Addison-Wesley Professional. https://doi.org/10.1007/3-540-45672-4_31

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2002). *Refactoring: Improving the design of existing code* (1st ed.). Addison-Wesley Professional.

Gay, L. R., Mills, G. E., & Airasian, P. (2012). *Educational research: Competencies for analysis and application* (10th ed.). Pearson.

Ghannem, A., Kessentini, M., Hamdi, M. S. S., & Boussaidi, G. El. (2017). Model refactoring by example: A multi-objective search-based software engineering approach. *Journal of Software: Evolution and Process*, *30*(4), e1916. https://doi.org/10.1002/smr.1916

Godfrey, M. W., & German, D. M. (2008, September). The past, present, and future of software evolution. In *2008 Frontiers of Software Maintenance* (pp. 129–138).

Ikart, E. M. (2019). Survey questionnaire survey pretesting method: An evaluation of survey questionnaire via expert reviews technique. *Asian Journal of Social Science Studies*, *4*(2), 1–17. https://doi.org/10.20849/ajsss.v4i2.565

Kaur, G., & Singh, B. (2017, June). Improving the quality of software by refactoring. In *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)* (pp. 185–191). IEEE.

Kaur, S., & Singh, P. (2019). How does object-oriented code refactoring influence software quality? Research landscape and challenges. *Journal of Systems and Software*, *157*, 110394. https://doi.org/10.1016/j.jss.2019.110394

Kim, M., Zimmermann, T., & Nagappan, N. (2014). An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, *40*(7), 633–649. https://doi.org/10.1109/TSE.2014.2318734

L'Erario, A., & Thomazinho, H. C. S. (2020). An approach to software maintenance: A case study. *International Journal of Software Engineering and Knowledge Engineering*, *30*(5), 603–630. https://doi.org/10.1142/S0218194020500217

Lacerda, G., Petrillo, F., Pimenta, M., & Gaël, Y. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, *167*, 110610. https://doi.org/10.1016/j.jss.2020.110610

Malhotra, R., & Jain, J. (2019, March). Analysis of refactoring effect on software quality of object-oriented. In *International Conference on Innovative Computing and Communications* (pp. 197–212). Springer Singapore. https://doi.org/10.1007/978-981-13-2354-6

Mariani, T., & Vergilio, S. R. (2017). A systematic review on search-based refactoring. *Information and Software Technology*, *83*, 14–34. https://doi.org/10.1016/j.infsof.2016.11.009

Mkaouer, W., Kessentini, M., Bechikh, S., Deb, K., & Cinnéide, M. Ó. (2014, September). Recommendation system for software refactoring using innovation and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (pp. 331–336).

Nardi, P. M. (2003). *Doing survey research: A guide to quantitative methods*. Boston: Pearson Education.

Nyamawe, A. S., Liu, H., Niu, Z., Wang, W., & Niu, N. (2018). Recommending refactoring solutions based on traceability and code metrics. *IEEE Access*, *4*(c), 49460–49475. https://doi.org/10.1109/ACCESS.2018.2868990.

Nyamawe, A. S., Liu, H., Niu, N., Umer, Q., & Niu, Z. (2019, September). Automated recommendation of software refactorings based on feature requests. In *2019 IEEE 27th International Requirements Engineering Conference (RE)* (pp. 187–198). IEEE. https://doi.org/10.1109/RE.2019.00029

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks.* (Doctoral dissertation, University of Illinois).

Ouni, A., Kessentini, M., Sahraoui, H., Cinnéide, M. Ó., Deb, K., & Inoue, K. (2015, February). A multi-objective refactoring approach to introduce design patterns and fix anti-patterns. In *First North American Search Based Software Engineering Symposium (NASBASE)* (pp. 1–16).

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., & Deb, K. (2016). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *25*(3), 1–53.

Rajlich, V. (2014, May). Software evolution and maintenance. In *Proceedings of the on Future of Software Engineering (FOSE)* (pp. 133–144). ACM.

Rehman, F., Maqbool, B., Riaz, M. Q., Qamar, U., & Abbas, M. (2018, April). Scrum software maintenance model: Efficient software maintenance in agile methodology. In *2018 21st Saudi Computer Society National Computer Conference (NCC)* (pp. 1–5). IEEE.

Rochimah, S., Arifiani, S., & Insanittaqwa, V. F. (2015). Non-source code refactoring: A systematic literature review. *International Journal of Software Engineering and Its Applications*, *9*(6), 197–214. https://doi.org/10.14257/ijseia.2015.9.6.19

Roscoe, J. T. (1975). *Fundamental research statistics for the behavioral sciences*. Holt.

Saunders, M., Lewis, P., & Thornhill, A. (2015). *Research methods for business students.* Pearson Education Limited.

Sekaran, U. (2003). *Research methods for business* (4th ed.). John Wiley

Sekaran, U., & Bougie, R. (2016). *Research methods for business: A skill building approach* (7th ed.). John Wiley & Sons Ltd.

Sun, X., Li, B., Leung, H., Li, B., & Li, Y. (2015). MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks. *Information and Software Technology*, *66*, 1–12. https://doi.org/10.1016/j.infsof.2015.05.003

Wang, H., Kessentini, M., Grosky, W., & Meddeb, H. (2015, October). On the use of time series and search based software engineering for refactoring recommendation. In *Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems* (pp. 35–42). ACM.

Zikmund, W. G., Babin, B. J., Carr, J. C., & Griffin, M. (2010). *Business research methods*. (8th ed.). South-Western: Cengage Learning.