A literature review for measuring maintainability of code clone

Shahbaa I. Khaleel, Ghassan Khaleel Al-Khatouni
Department of Software, College of Computer Science and Mathematics, The University of Mosul, Mosul, Iraq

Article Info

Article history:

Received Nov 26, 2022
Revised Mar 23, 2023
Accepted Apr 2, 2023

Keywords:

Code clone
Cyclomatic complexity
Maintainability
Maintainability index
Refactoring

ABSTRACT

Software organizations face constant pressure due to stakeholder requirements and the increasing complexity of software systems. This complexity, combined with defects in code quality and failures, can pose risks to software systems. To ensure code is understood before maintenance, developers must spend over 60% of their time modifying and improving code quality, which is costly. This study examines the impact of code refactoring activities on software maintainability and quality by reviewing relevant research and explaining key terms. The research finds that refactoring activities can enhance specific quality characteristics, including maintainability, understandability, and testability. The study also identifies important factors that should be considered when developing refactoring tools. Refactoring enables code improvement without altering program behavior and can be applied multiple times to source code.

Corresponding Author:

Shahbaa I. Khaleel
Department of Software, College of Computer Science and Mathematics, The University of Mosul
Mosul, Iraq
Email: shahbaaibrkh@uomosul.edu.iq

1. INTRODUCTION

The success of software is closely tied to its quality, which is influenced by two categories of traits [1]: internal and external. Internal characteristics, such as cohesion class, are typically measured from within the software, while external attributes, such as maintenance, are indirectly measured. A major external quality trait and a source of concern in software engineering is software maintainability, which refers to the ease with which a software component can be modified to correct errors, improve performance, adapt to a changing environment, or make other adjustments. The maintainability of software is determined by various factors related to software modification, including correction, improvement, adaptation, and prevention [2].

The issue of code cloning, which involves copying and pasting code segments, can have a negative impact on software quality, particularly with regards to maintainability. As software expands, code cloning can occur unintentionally, resulting in repeated code throughout the source code [3]. This repetition is also known as software code or code clones, which can lead to challenges in software maintenance, including increased costs and difficulty working with the software [4]. To address these issues, it is important to identify and address code cloning early on in software development. This can be done by using tools to remove code clones, followed by refactoring and comparing the software before and after these changes are made [5].

2. MAINTAINABILITY

Maintainability refers to how easy or difficult it is to modify a software system and is a key aspect of software quality. It encompasses any changes made to the software after installation or release, such as fixing errors or adding new features [3]. The quality of the source code is a major factor in determining maintainability, and the term "maintainability" is commonly used to describe the quality of software [6]. However, measuring maintainability can be challenging and is often tied to the cost of maintenance [7].

Software systems with poor maintainability may require more costly maintenance work. Overall, maintainability is best understood as the ease with which software can be modified, corrected, and updated, and it is important from the moment the software is delivered to customers [8]. Software maintenance is a critical component of the software development lifecycle and can account for up to 80% of the total costs associated with software development. Therefore, ensuring that software is maintainable is crucial. Code cloning can result in larger code sizes, which can hinder software maintenance. Additionally, code cloning can lead to the creation of unnecessary or dead code, which can further complicate software maintenance [9], [10].

3. MEASURING MAINTAINABILITY

Measuring maintainability involves identifying metrics that capture the key characteristics of maintenance programs and then quantifying them. In traditional software development practices, developers use code standards to identify and address challenging areas of the system. This often involves refactoring code to improve maintainability. To better assess maintainability and measure it in the early stages of development, it would be useful to establish a set of metrics that can evaluate maintainability effectively [11].

During the software development process, maintainability is a crucial quality that is frequently evaluated. One common way to assess maintainability is through the use of the maintainability index, a statistical measure that produces a numerical score representing the overall maintainability of the system. The maintainability index relies on three metrics: cyclomatic complexity, lines of code, and halstead metrics. These measures are used to determine how maintainable the system is and to identify areas that may need improvement [12], [13]. Knowing the maintenance of the program helps in [14]:

- Determine whether the current software and any parts of them will reused.
- Decide on maintenance or redevelopment of the components of programs.
- Estimating or guessing the amount of effort made in the maintenance of the components of programs.
- Determine the costs and criteria for admission for software.
- Providing evaluation data for reuse warehouses.
- Providing or preserving a large part of the total cost of software ownership.

4. CODE CLONE

"Code clones" refer to two or more identical or similar parts of source code. Copying or reproducing code is a common practice in the software industry, as it allows developers to reuse existing code and resources. However, the proliferation of code clones can create maintenance problems that are time-consuming and costly to address [15]. Code clones can be classified into two basic categories: syntactic clones, which are based on the structure of the code (first, second, and third types), and semantic clones, which based on the functionality of the code (fourth type) [16].

- The first type: It is parts of identical software code except for white distances and comments.
- The second type: It is the clips of the code similar to each other but they have slight changes in the name, such as renaming identifiers or variables, which are also referred to as the renamed clones.
- The third type clones or near-miss clones: It is parts of the similar code modified in addition or deletions to the code.
- The fourth type: It is one of the semantic clones, and it is these cloned code that perform a similar function, but it may have a different syntactic structure.

The process of cloning software involves writing code once and then reusing it in different parts of the program with some modifications to fit the needs of each version. This practice can save time and resources. However, inconsistencies in the cloned code can lead to the emergence of bugs or weaknesses in the system, which can further contribute to the deterioration of the software design [17].

5. CODE CLONE DETECTION TECHNIQUES

In software development and maintenance, detecting cloned codes is essential. To minimize software repetition, reduce maintenance costs, and enhance program productivity, various techniques are used, each with its unique characteristics and benefits. These techniques can be categorized into six methods [18]–[20]: text-based, token-based, tree-based, graph-based, metrics-based, and hybrid techniques. The text-based technique involves treating the original source code as a text composed of lines or sequential words and comparing two texts or sections to identify similarities in lines. Hash comparison algorithms are used to calculate hashes for each line, and cloned codes are detected by comparing the hashes. In the token-based technique, the source code is converted into sequential segments, and lexical rules specific to the programming language are used to represent each line of code [21]. The tree-based technique involves representing the source code as an abstract syntax tree and comparing parts of the tree using algorithms to detect code clones [22]. The graph-based technique creates a program dependency graph (PDG) that contains information about the flow of data and control in the software and is considered the most efficient in detecting code clones [23], [24]. Metrics-based techniques involve combining several code metrics to detect clones, while hybrid techniques combine more than one of the aforementioned techniques to detect code clones. Finally, the hybrid technique combines two or more of the aforementioned techniques to detect clones and find cloned code. Developers can choose the most suitable technique based on the type of code and the level of similarity detection required [25]-[28]. All these techniques are explained in Table 1 that shows features, weaknes and all attributes that related to them [29]. As for the sixth type, which is the last, it depends on the techniques used.

Table 1. Explains the comparison between the techniques of detecting cloned code

No	Technology name	Features	Weaknesses	Execution Time	The type of clones discovered	Accuracy
1	Text-based	Easily implementation	The fourth type cannot be discovered	short	1,2,3	High
2	Token-based	expands easier	The fourth type cannot be discovered	short	1,2,3	High
3	AST-based	Contains more information about software	Difficulty in composition (AST)	short	1,2,3,4	High
4	PDG-based	Discover all types of clones	Difficulty in establishing (PDG)	long	1,2,3,4	High
5	Metric-based	Discover all types of clones	Difficulty implementing measure	long	1,2,3,4	High

6. REFACTORING

The main goal of identifying duplicated codes is to eliminate them from the system via refactoring, which enhances the quality of the system [27]. Refactoring is a crucial criterion for evaluating code quality, and it involves adjusting the internal structure of a software component to make it more comprehensible and modifiable without altering the overall functionality of the software. This method is utilized to enhance code quality by directly measuring software metrics, indirectly measuring software quality attributes, and improving software performance [30]. Refactoring is a technique that developers typically employ throughout the software's lifecycle and evolution, with the goal of improving software quality attributes [31]. The aim of refactoring is to modify the internal structure of software components to make them more straightforward to comprehend and modify without affecting their external behavior [32]. It entails reducing the cost of code maintenance by employing a variety of strategies, such as extracting, moving methods, fields, or classes [33]. The refactoring process includes many of the following steps, which are used to enhance and measure the code's quality [34], as depicted in Figure 1.

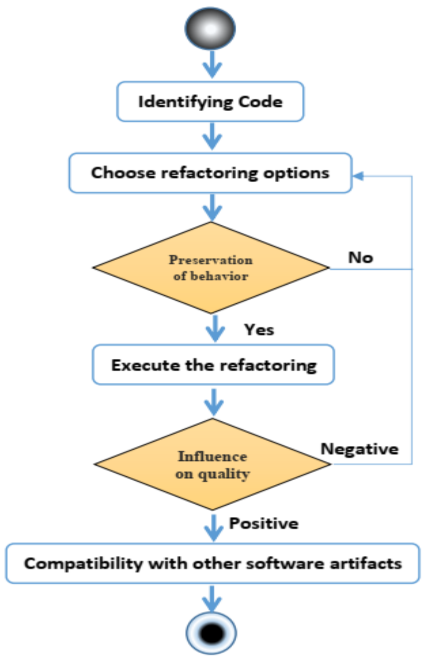

Figure 1. Explains the steps refactoring process in terms of code quality

First: It defines the code that needs to be refactoring. Second: Choosing one of the refactoring methods or techniques to be used. Third: Ensure that the refactoring methods that have been used maintain the behavior of the programs after refactoring. Fourth: the application of specific refactoring techniques. Fifth: Assessing the impact of refactoring techniques on software quality. Sixth: Maintaining congruence between the refactoring software and the design of other software.

7. REFACTORING TECHNIQUES

The main objective of refactoring is to modify the internal structure of the components of the software in this a manner easily understandability and adjustable without any change in external behavior, and this is an effective way to make a change in an old system without compromising the functionality of the system from the external side. Refactoring can significantly reduce the number of errors. This also contributes to improved internal structure, readability, maintainability, and reduced complexity [32]. Software systems naturally need to evolve. Studies have shown that over a period of time, maintenance is very costly, sometimes consuming up to 90% of total software spending [35]. Using code refactoring techniques is an effective way to maintain software quality without increasing maintenance costs. There are some refactoring concepts like (bad smells) where "Bad Smells" refers to design flaws in the code. These are considered problems in the code and can be removed and resolved using refactoring [36]. Refactoring techniques are used to remove "Bad Smells" and clean up the code with refactoring. The research offers dozens of techniques for refactoring the code. Fowler identified 72 techniques for refactoring [32], of which 10 are mentioned in Table 2, namely:

– Duplicate observed data.
– Replace type code with subclass.
– Replace conditional with polymorphism.
– Extract subclass.
– Extract interface.
– Form template method.
– Introduce null objects.
– Push down method.
– Introduce local extension.
– Replace type code with state/strategy.

Table 2. Refactoring techniques overview

Author	Study of a case	Intern al metrics	Extern al metrics	Technique
Kataoka *et al.* [37]	C + + program	Cohesion	Maintain ability	Extraction method, class extraction
Stroulia and Kapoor [38]	Academy/theory	Size, coupling	De sign expansion	Super class extraction, Ab stract class extraction
Leitch and Stroulia [39]	Academy/ Commercial	size of the code and number of actions	Maintenance efforts and costs	Extraction method and transportation method
Tahvildari and Kontogiannis [40]	Four open-source applications	Cohesion, coupling, Inheritance, and Complexity	maintainability	Code transformations
Bois and Mens [41]	Free and open-source software	Cohesion, coupling	-------	Method extraction, method transfer, method replacement, method object, data value ob ject replacement, class extraction
Moser *et al.* [42]	Industrial proj ect	CK, L OC, FOUT, effect Metrics	Outcomes (LOC)	-------
Alshayeb [43]	3 small open - sourced projects	CK, LO C, FO UT Metrics	Ability to adapt, maintenance, understanding, re - usability, and testability	Class extraction, field encapsulation, subclass extraction, class move, extraction method, temporary replace with query
Sahraoui *et al.* [44]	C + + pro gram	Inheritance, coupling	Exposure to errors	Super class extraction, subclass extraction, and full class extraction
Tahvildari *et al.* [45]	Both the industrial project and the open library are in C.	Halstead effort, LOC, number of comment lines per unit	Maintain ability and performance	Pattern design
Kumari and Saha [46]	The code is open source.	Cohesion, coupling and Inheritance	Ability to adapt, maintenance, understanding, re - usability, and testability, stability and completeness	Hide return value, safe delete and overwrite constructor Hide the return value, delete safely, and overwrite the constructor

By observing the above table, it is clear that the process of refactoring software directly affects the internal and external measures of software quality. The quality of the technique used in conducting the refactoring process has a major and direct role on the type of measures affected by it. Therefore, when choosing a refactoring technique, care must be taken so that it is taken advantage of and does not negatively affect the quality of the software.

8. CHARACTERISTICS OF SOFTWARE QUALITY

The quality model in ISO 9126 is its division of internal and external software product quality concepts into 6 main attributes which are further subdivided into quality sub-attributes. This division is shown in Figure 2 [47]. The focus is on maintainability, which is divided into:

– Analyzability: How simple or hard it is to detect system flaws or define components that need modification.
– Changeability: How simple or complicated it is to make changes to the system.
– Stability: How simple or hard it is to maintain the system's consistency throughout adjustment.
– Testability: How simple or hard it is to test a system after it has been modified.

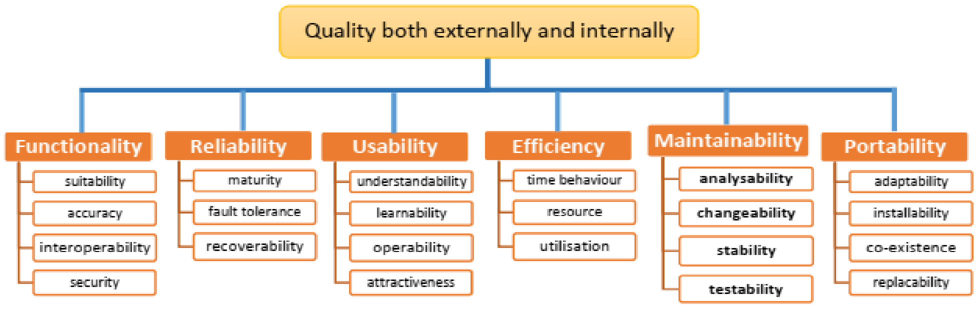

Figure 2. Explains the concepts the quality of internal, external software

9. RELATED RESEARCH

This section provides an in-depth analysis of research conducted on software maintainability and how code refactoring can enhance code quality. Additionally, it investigates the impact of software refactoring on both the internal and external aspects of software. The results of these studies highlight the significance of code refactoring as a beneficial technique in software development.

9.1. Related research to maintainability

Najm [48] introduced a practical implementation of the maintainability index (MI) using a new, simple and efficient method compared to the traditional method. To find MI in relation to LOC only. To validate the method was developed the maintainability index (MMIS) software, which finds first MI in relation to (LOC, CC, HV) and secondly finds MI in relation to LOC only. The study showed that using the MI calculation method that only considers LOC tended to produce more maintainable software, whereas the method that takes into account (LOC, HV, CC) was more challenging to maintain. However, it is important to note that this research only evaluated the maintainability of the software tool used for calculating MI, not the software code itself.

Bal and Sood [7] presented a study on cloning management (code cloning) where various aspects of cloning management were discussed in the form of approaches, metrics, tools and techniques. The relationship between code cloning, software reliability and maintainability of the software was explained, leading to the following results: Modules involving code cloning are 1.7 times more reliable than non-cloned modules. Units or modules with larger code clones are less maintainability than units with smaller code clones.

Ashtaputre *et al.* [49] presented an approach in 2016 that takes source code as input to identify code clones and determine which cloned codes can be refactored without compromising software behavior. This method has the advantage of addressing system defects and has a lower or nearly negligible computational cost. wang and colleagues [50] developed a machine learning approach called CREC that automatically identifies refactoring and non-refactoring code sets in software repositories and trains models to suggest cloning for refactoring. Interestingly, they found that features derived from previous versions of the software were more successful in this process compared to those from the current version, implying that developers tend to consider past development more than the present when refactoring software. This finding emphasizes the significance of taking into account historical versions of software in making refactoring decisions, according to the authors.

Biase *et al.* [51] introduced the delta maintainability model (DMM), a novel method for assessing code maintainability by adapting and extending the SIG maintainability model. The DMM focuses on measuring precise code modifications and categorizes modified lines of code into low- and high-risk categories. The primary objectives of the DMM are twofold: to generate high-quality and actionable grades for code maintainability, and to rank and compare maintainability based on exact changes while also quantifying the percentage of low-risk code lines.

Devi *et al.* [3] developed a regression-based model to evaluate the maintainability of cloned and refactored code, exploring the correlation between the two factors. The researchers utilized CCFinder to identify code clones and categorize them into Type I and II, as well as measured clone classes based on file sizes, copy group dimensions, and line-based metrics. Various metrics were applied to each code clone, yielding different results. Ultimately, the authors concluded that while some code clones could be easily maintained through refactoring, others showed little to no improvement even after extensive reworking.

Sood and Bal [17] evaluated the maintenance index and compliance metrics for a software maintenance case study. They used two different tools, namely Analyze and CppDepend, to calculate the maintenance index and compliance values, respectively. The researchers conducted a case study of the C++ program "DECEIVER" to explore the relationship between maintenance index and compliance calculations in reproducing the software code. The findings indicated that imbalanced or inconsistent compliance values could negatively impact the software design or architecture.

Draz [30] conducted a study to evaluate the impact of code quality refactoring by measuring several internal and external quality characteristics of the code. Their findings showed that after refactoring, the internal quality characteristics such as inheritance, complexity, cohesion, and coupling had improved positively, except for the size metric. Additionally, the external quality characteristics, including reusability, maintainability, understanding, and efficiency, showed improvement in code quality, except for performance, which had a negative effect.

9.2. Related research to the quality of code

Shahjahan *et al.* [52] conducted a study to explore how graph theory techniques could be utilized to improve code attributes. Refactoring was identified as a method to improve code quality without modifying its internal or external behavior, while also improving response time, analytical capabilities, and the time required to make changes. Bavota *et al.* [53] conducted a study where they used the RefFinder2 tool to detect refactoring and measure development time for 3 open-source projects. The study aimed to investigate the correlation between refactoring and program quality. The results showed that 42% of the refactoring process was influenced by code smell, and only 7% of the refactored elements were successful in removing code smell.

Kadar *et al.* [54] put forward the idea to examine code refactoring as a means of creating an open dataset that would be necessary for utilizing software code metrics and refactoring techniques across various versions of seven systems. Through the use of refactoring methods, the authors aimed to analyze the quality features of duplicate source code classes, as well as the effectiveness of updating source code metrics. The study also investigated how to achieve the refactoring effect on source code metrics and the correlation between maintenance and refactoring techniques. To achieve this, the authors proposed a dataset that encompasses information on refactoring, as well as over 50 software code metrics, for 37 versions of seven open-source programs at both the class and action levels.

Ouni *et al.* [55] suggested a multistandard refactoring approach wherein refactoring is performed simultaneously using various automated methods. The study revealed that the results of the refactoring had a beneficial effect on cyclomatic complexity. Cedrim *et al.* [56] conducted a study that involved analyzing 25 projects in order to assess improvements in software quality. Their analysis focused on the connection between code smells and code refactoring, specifically identifying instances where bad structures were added or removed from the code. The results of their study showed that only a small percentage of refactoring involved the removal of software smell (2.24%), while an even smaller percentage involved the addition of smell (2.66%).

Kaur and Singh [57] conducted a study to analyze software and maintain different versions of the codebase for a project. They utilized a tool called "ref-finder" to identify refactoring opportunities and assess the impact of previous versions on the current one. The code maintainability index was used to measure the code metrics for each update, and new functionality was introduced along with improved refactoring methods to better measure code metrics. Finally, the most recent version was compared to the previous revision to evaluate the outcomes.

Chavez *et al.* [58] conducted a study to assess the impact of refactoring on the internal quality of software, using 25 quality metrics to evaluate 5 specific features: cohesion, complexity, inheritance, coupling, and size. The findings of the study suggest that refactoring techniques can improve internal quality characteristics without introducing any negative effects. Although 55% of refactoring processes resulted in improvements, there was only a 10% decrease in quality. This indicates that refactoring is generally effective in enhancing software quality.

Vashisht *et al.* [59] conducted an empirical study to examine the impact of copy refactoring on software products, using four different open-source tools. Their findings revealed that the refactoring process led to a decrease in the software's complexity, while enhancing its functionality, reusability, and certain quality attributes. However, other quality characteristics such as effectiveness, scalability, understandability, and flexibility were diminished. The study suggests that refactoring techniques can have either positive or negative effects on software quality characteristics.

Pantiuchina *et al.* [60] conducted an empirical study to explore the relationship between seven software code metrics and quality improvement. Their research showed that not all metrics related to quality had a significant impact on the efforts of developers to improve quality. Additionally, they found that refactoring had minimal impact on software quality.

Alawairdhi [32] investigated the impact of code refactoring on quality attributes(external, internal) like performance, maintainability, efficiency, and code lines next refactoring, 6 internal and 4 external quality metrics were quantified for each component. The findings indicate that refactoring had a powerful effect on parts of the internal quality. Yet, the impact of refactoring code on external quality was little.

Alomar *et al.* [61] developed a research design study based on 8 features for internal quality, also 27 metrics to assess the relationship between software metrics and quality improvement The findings indicate enhancement and deterioration of quality. Many of the metrics used to enhance quality attributes (coupling, complexity, and cohesion) provide ideas for improving quality for developers. Kiyak [62] utilized data mining and refactoring even though refactoring automated not provide the required performance; the manual refactoring process takes a long time through algorithms for unsupervised learning that decrease the numeral of refactoring choices and had a positive impact on quality. Increases code readability and source code maintainability; reduces the complexity of the software system.

Fernandes *et al.* [63] conducted a study analyzing 23 software projects that underwent 29,000 refactoring measures, nearly half of which involved re-refactoring. The study aimed to understand the impact of refactoring and re-refactoring on five factors: cohesion, complexity, coupling, inheritance, and size, using both illustrative analysis and statistical studies. The findings indicate that 90% of refactoring and 100% of re-refactoring were performed on code components with only one required feature. Table 3 shows a summary of previous studies related to software refactoring and its impact on code quality, and the researchers' findings.

Table 3. Shows a summary of previous studies with code quality and software quality

Author	year	Methodology	result
Shahjahan *et al.* [52]	2015	Using graphic theory techniques. refactoring code is a method of beneficent quality with out affecting its external and internal behavior.	The response time has been improved, to analyzability, changeability, and improve the quality of the code.
Bavota *et al.* [53]	2015	used the RefFinder2 tool and this not only to detect re factoring but also to record development time for three open-source projects.	The study showed that 42% of the Code Smell was affected. there 7% of the components, the refactoring was able to delet code smell.
Ouni *et al.* [55]	2016	Multi-standard refactoring as structurally was automatically repeated using different techniques.	refactoring had a positive impact on software cyclomatic complexity, according to the findings.
Cedrim *et al.* [56]	2016	Presented an analysis included 25 project to verify quality improvement. They investigate the relationship between code smell and code refactoring. of code to identify refactoring based on adding or removing bad structures to the code.	According to the findings, only 2.24% of refactoring was removed from software smell, while 2.66% was added.
Chavez *et al.* [58]	2017	Presented a validation of using 25 quality metrics, we looked into the effect of refactoring on five aspects of quality (cohesion, coupling, complexity, inheritance, and size).	The study shows that refactoring processes Internal quality attributes should be enhanced, not aggravated.
Vashisht *et al.* [59]	2018	Through an empirical analysis, they investigated the impact of copy refactoring on software quality using four different open-source software.	It turns out that refactoring methods have a positive or negative impact on the characteristics of quality.
Pantiuchina *et al.* [60]	2018	Applying an experimental study to verify the relationship between seven measures of software codes and quality improvement.	They discovered that refactoring changes had little impact on software quality.
Alawairdh [32]	2019	Investigate the impact of re factoring the code on internal, external quality properties such as maintain ability, execution, efficiency and code lines.	Re factoring had a major impact the internal quality of the application. Like complexity.
Alomar *et al.* [61]	2019	Developed an exploratory research study to evaluate the relationship between quality improvement and software metrics based on (8) internal quality characteristics and (27) metrics.	The findings that there are numerous measures available to improve and degrade program quality.
Kiyak [62]	2020	Utilized data mining and refactoring even though refactoring automated not provide the required performance; the manual refactoring process takes a long time through algorithms for unsupervised learning that decrease the number of refactoring choices.	The software quality improved because of the restructuring. Reduces the complexity of the software system by improving readability and maintainability.
Fernandes *et al.* [63]	2020	Used 23 software systems for more than 29,000 refactoring actions, about half of them a refactored in order to understand the effect of refactoring and re-refactoring.	According to this study, 90% of refactoring and 100% of re-refactoring are performed on code segments that have one required characteristic.

The Table 3 briefly shows all relevant conclusions about whether code refactoring has a positive effect on program quality. Most research has focused on internal or external quality traits. Nor has anyone provided coverage of both quality attributes and the effect of large-scale refactoring on all quality attributes. Most research focuses on measuring the effect of software refactoring on quality characteristics. On the other hand, previous research has shown that re-engineering improves software quality. Increases code readability and maintainability while reducing system complexity. It was concluded from previous works that in order to obtain high quality, all quality characteristics must be taken into account and their importance in improving the quality of the program when restructuring it.

10. CONCLUSIONS

Refactoring is activities that increase the quality of software and allow software engineers to repair software code that is difficult to maintain. There are many methods and tools for the application of restructuring activities. The method or tool chosen depends on the nature of the program. In this study, some terms were explained review studies for previous and relevant research that identifies code refactoring activities and their impact on software maintainability. It also examines their impact on software quality characteristics and outlines the important factors that need to be filled when creating refactoring tool. The research concludes: First, the application of refactoring activities will increase the values of certain quality characteristics such as understandable, maintainability and testability. Second, there are many factors that affect refactoring activities. Third, refactoring helps improve code without changing program behavior. Finally, refactoring can be applied to the source code several times.

ACKNOWLEDGEMENTS

Authors would like to thank the University of Mosul in Iraq for providing Moral support.

REFERENCES

[1] S. Khaleel, “Designing a tool to estimate software projects based on the swarm intelligence,” *International Journal of Intelligent Systems*, vol. 14, no. 4, pp. 524–538, 2021, doi: 10.22266/ijies2021.0831.46.

[2] H. Alsolai, S. Arabia, U. Kingdom, M. Roper, and U. Kingdom, “A systematic literature review of machine learning techniques for software maintainability prediction,” *Information and Software Technology*, vol. 119, no. 0950–5849, pp. 1–52, 2020, doi: 10.1016/J.INFSOF.2019.106214.

[3] U. Devi, N. Kesswani, and A. Sharma, “An efficient model for measuring maintainability from code cloning and refactoring using regression,” *International Journal of Scientific and Technology Research*, vol. 8, no. 12, pp. 4019–4023, 2019.

[4] D. Hou, F. Jacob, and P. Jablonski, “Exploring the design space of proactive tool support for copy-and-paste programming,” *In Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, 2009, pp. 188–202, doi: 10.1145/1723028.1723051.

[5] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “Method and implementation for investigating code clones in a software system,” *Information and Software Technology*, vol. 49, no. 9–10, pp. 985–998, 2007, doi: 10.1016/j.infsof.2006.10.005.

[6] A. Calleja, J. Tapiador, and J. Caballero, “The malsource dataset: Quantifying complexity and code reuse in malware development,” *IEEE IEEE Transactions on Information Forensics and Security*, vol. 14, no. 12, pp. 3175–3190, 2019, doi: 10.1109/TIFS.2018.2885512.

[7] S. Bal and S. Sood, “Software clone management-an insight,” *Journal Computer Technology*, vol. 5, no. 2, pp. 43–51, 2015.

[8] M. Gradišnik, S. Karakatič, T. Beranič, M. Heričko, G. Mauša, and T. G. Grbac, “The impact of refactoring on maintability of Java code: A preliminary review,” *CEUR Workshop Proceddings*, vol. 2508, no. September, pp. 22–25, 2019.

[9] S. Khaleel and R. Khaled, “Selection and prioritization of test cases by using bees colony,” *AL-Rafidain Journal of Computer Sciences and Mathematics*, vol. 11, no. 1, pp. 179–201, Jul. 2014, doi: 10.33899/csmj.2014.163746.

[10] S. Schulze, “Analysis and removal of code clones in software product lines,” PhD diss., Magdeburg, Universität, Diss, 2013.

[11] A. Awad and S. Saleh, “Identifying metrics for measuring maintainability of models defined in systemweaver,” 2020.

[12] S. Khaleel and A. Al Thanoon, “Design a tool for generating test cases using swarm intelligence,” *AL-Rafidain Journal of Computer Sciences and Mathematics*, vol. 10, no. 1, pp. 421–444, Mar. 2013, doi: 10.33899/csmj.2013.163468.

[13] S. Kadi, “Measuring Maintainability and latency of Node. js frameworks,” 2021.

[14] C. Chen, R. Alfayez, K. Srisopha, B. Boehm, and L. Shi, “Why is it important to measure maintainability and what are the best ways to do it?,” in *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, 2017, pp. 377–378, doi: 10.1109/ICSE-C.2017.75.

[15] X. Zhang, T. Wang, Y. Yu, Y. Zhang, Y. Zhong, and H. Wang, “The development and prospect of code clone,” *CHI ’22 ACM CHI Conference on Human Factors in Computing Systems April 30-May 6, 2022, New Orleans, LA*, vol. 1, no. 1, 2022, [Online]. Available: http://arxiv.org/abs/2202.08497.

[16] A. Kumar, R. Yadav, and K. Kumar, “A systematic review of semantic clone detection techniques in software systems,” *IOP Conference Series: Materials Science and Engineering*, vol. 1022, no. 1, 2021, doi: 10.1088/1757-899X/1022/1/012074.

[17] S. Sood and S. Bal, “Analyzing maintainability and compliance of different version of software in code cloning,” *Think India Journal*, no. 14, pp. 14222–14228, 2019.

[18] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, “Functional code clone detection with syntax and semantics fusion learning,” in *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2020, vol. 20, pp. 516–527, doi: 10.1145/3395363.3397362.

[19] S. Karthik, “Detection of metrics based code cloning using optimised SVM algorithm,” no. 6641. EasyChair, 2021.

[20] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” Citeseer, 2007. Accessed: Sep. 28, 2022. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.9931&rep=rep1&type=pdf

[21] W. Toomey, “Ctcompare: Code clone detection using hashed token sequences,” in *2012 6th International Workshop on Software Clones, IWSC 2012 - Proceedings*, 2012, pp. 92–93, doi: 10.1109/IWSC.2012.6227881.

[22] L. Jiang, Z. Su, and E. Chiu, “Context-based detection of clone-related bugs,” in *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2007*, 2007, pp. 55–64, doi: 10.1145/1287624.1287634.

[23] J. Li and M. D. Ernst, “CBCD: Cloned buggy code detector,” in *Proceedings - International Conference on Software Engineering*, 2012, pp. 310–320, doi: 10.1109/ICSE.2012.6227183.

[24] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, “SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code,” in *Proceedings of the Digital Forensic Research Conference, DFRWS 2015 EU*, Mar. 2015, vol. 12, pp. S61–S71, doi: 10.1016/j.diin.2015.01.011.

[25] P. Batta and M. Himanshi, “Hybrid technique for software code clone detection,” *International Journal of Computer Technology*, vol. 2, no. 2, pp. 98–102, 2012, doi: 10.24297/ijct.v2i2b.2639.

[26] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, May 2009, doi: 10.1016/j.scico.2009.02.007.

[27] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” vol. 541, p. 115, 2007, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.62.7869%5Cnhttp://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.9931&rep=rep1&type=pdf

[28] H. Nasirloo and F. Azimzadeh, “Semantic code clone detection using abstract memory states and program dependency graphs,” in *2018 4th International Conference on Web Research, ICWR 2018*, Jun. 2018, pp. 19–27, doi: 10.1109/ICWR.2018.8387232.

[29] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, “A systematic review on code clone detection,” *IEEE Access*, vol. 7, pp. 86121–86144, 2019, doi: 10.1109/ACCESS.2019.2918202.

[30] A. M. Draz, “A survey of refactoring impact on code quality,” *FCI-H Informatics Bulletin*, vol. 3, no. 1, pp. 16–22, 2021.

[31] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, “How does refactoring affect internal quality attributes?: A multi-project study,” in *ACM International Conference Proceeding Series*, 2017, pp. 74–83, doi: 10.1145/3131151.3131171.

[32] M. Alawairdhi, “Code refactoring and its impact on internal and external software quality : an experimental study,” *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 19, no. 6, pp. 12–17, 2019.

[33] B. S. G. Kaur, “Improving the quality of software by refactoring,” *International Conference on Intelligent Computing and System*, pp. 185–191, 2017, doi: 10.1109/ICCONS.2017.8250707.

[34] A. M. E. S. M. Draz, M. S. Farhan, and M. M. Eldefrawi, “A survey of refactoring impact on code quality,” *FCI-H Informatics Bulletin*, vol. 3, no. 1, pp. 16–22, Feb. 2021, doi: 10.21608/FCIHIB.2021.54539.1007.

[35] J. Koskinen, “Software maintenance costs l,” *University of Jyksky*, 2010.

[36] D. B. Roberts and R. Johnson, “Practical analysis for refactoring,” *Univ. Illinois Urbana-Champaign, Champaign*, 1999.

[37] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, “A quantitative evaluation of maintainability enhancement by refactoring,” in *Conference on Software Maintenance*, 2002, pp. 576–585, doi: 10.1109/icsm.2002.1167822.

[38] E. Stroulia and R. Kapoor, “Metrics of refactoring-based development: an experience report,” in *OOIS 2001*, Springer London, 2001, pp. 113–122, doi: 10.1007/978-1-4471-0719-4_13.

[39] R. Leitch and E. Stroulia, “Assessing the maintainability benefits of design restructuring using dependency analysis,” in *Proceedings - International Software Metrics Symposium*, 2003, vol. 2003-Janua, pp. 309–322, doi: 10.1109/METRIC.2003.1232477.

[40] L. Tahvildari and K. Kontogiannis, “Improving design quality using meta-pattern transformations: A metric-based approach,” in *Journal of Software Maintenance and Evolution*, 2004, vol. 16, no. 4–5, pp. 331–361, doi: 10.1002/smr.299.

[41] B. D. Bois and T. Mens, “Describing the impact of refactoring on internal program quality,” *In International Workshop on Evolution of Large-scale Industrial Software Applications*, pp. 37-48, 2022, [Online]. Available: http://soft.vub.ac.be/FFSE/Workshops/ELISA-submissions/07-DuBoisMens-full.pdf

[42] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, “A case study on the impact of refactoring on quality and productivity in an agile team,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5082 LNCS, pp. 252–266, 2008, doi: 10.1007/978-3-540-85279-7_20.

[43] M. Alshayeb, “Empirical investigation of refactoring effect on software quality,” *Information and Software Technology*, vol. 51, no. 9, pp. 1319–1326, 2009, doi: 10.1016/j.infsof.2009.04.002.

[44] H. A. Sahraoui, R. Godin, and T. Miceli, “Can metrics help to bridge the gap between the improvement of OO design quality and its automation?,” in *Conference on Software Maintenance*, 2000, pp. 154–162, doi: 10.1109/icsm.2000.883034.

[45] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos, “Quality-driven software re-engineering,” *Journal of Systems and Software*, vol. 66, no. 3, pp. 225–239, 2003, doi: 10.1016/S0164-1212(02)00082-1.

[46] N. Kumari and A. Saha, “Effect of refactoring on software quality,” in *airccj.org*2014 , pp. 37–46, doi: 10.5121/csit.2014.4505.

[47] I. Heitlager, T. Kuipers, and J. Visser, “A practical model for measuring maintainability - A preliminary report,” *In 6th international Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pp. 30–39, 2007, doi: 10.1109/QUATIC.2007.7.

[48] N. M. A. M. Najm, “Measuring maintainability index of a software depending on line of code only,” *IOSR Journal of Computer Engineering*, vol. 16, no. 2, pp. 64–69, 2014, doi: 10.9790/0661-16276469.

[49] P. Ashtaputre, C. Kulkarni, and Y. Lonkar, “An effective approach to find refactoring opportunities for detected code clones,” *International Journal of Innovative Research in Science, Engineering and Technology*, pp. 10042–10046, 2016, doi: 10.15680/IJIRSET.2015.0506071.

[50] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, “Automatic clone recommendation for refactoring based on the present and the past,” *In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, no. 1, 2018, pp. 115–126, doi: 10.1109/ICSME.2018.00021.

[51] M. D. Biase, A. Rastogi, M. Bruntink, and A. V. Deursen, “The delta maintainability model: Measuring maintainability of fine-grained code changes,” *Proc. - 2019 IEEE/ACM Int. Conf. Tech. Debt, TechDebt 2019*, pp. 113–122, 2019, doi: 10.1109/TechDebt.2019.00030.

[52] A. Shahjahan, W. H. Butt, and A. Z. Ahmad, “Impact of refactoring on code quality by using graph theory: An empirical evaluation,” in *IntelliSys 2015 - Proceedings of 2015 SAI Intelligent Systems Conference*, Dec. 2015, pp. 595–600, doi: 10.1109/IntelliSys.2015.7361201.

[53] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, Sep. 2015, doi: 10.1016/j.jss.2015.05.024.
[54] I. Kádár, P. Hegedüs, R. Ferenc, and T. Gyimóthy, "A code refactoring dataset and its assessment regarding software maintainability," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016*, May 2016, pp. 599–603, doi: 10.1109/SANER.2016.42.
[55] 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 Transactions on Software Engineering and Methodology*, vol. 25, no. 3, Jun. 2016, doi: 10.1145/2932631.
[56] D. Cedrim, L. Sousa, R. Gheyi, and A. Garcia, "Does refactoring improve software structural quality? A longitudinal study of 25 projects," in *ACM International Conference Proceeding Series*, Sep. 2016, pp. 73–82, doi: 10.1145/2973839.2973848.
[57] G. Kaur and B. Singh, "Improving the quality of software by refactoring," in *Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems, ICICCS 2017*, Jul. 2017, pp. 185–191, doi: 10.1109/ICCONS.2017.8250707.
[58] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes?: A multi-project study," in *ACM International Conference Proceeding Series*, Sep. 2017, pp. 74–83, doi: 10.1145/3131151.3131171.
[59] H. Vashisht, S. Bharadwaj, and S. Sharma, "Impact of clone refactoring on external quality attributes of open source softwares," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 8, no. 5, pp. 86–94, 2018, doi: 10.32628/cseit183833.
[60] J. Pantiuchina, M. Lanza, and G. Bavota, "Improving code: The (mis) perception of quality metrics," in *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, Nov. 2018, pp. 80–91, doi: 10.1109/ICSME.2018.00017.
[61] E. A. Alomar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the impact of refactoring on the relationship between quality attributes and design metrics," in *International Symposium on Empirical Software Engineering and Measurement*, Sep. 2019, doi: 10.1109/ESEM.2019.8870177.
[62] E. O. Kiyak, "Data mining and machine learning for software engineering," in *Data Mining - Methods, Applications and Systems*, IntechOpen, 2021, doi: 10.5772/intechopen.91448.
[63] E. Fernandes *et al.*, "Refactoring effect on internal quality attributes: What haven't they told you yet?," *Information and Software Technology*, vol. 126, p. 106347, Oct. 2020, doi: 10.1016/j.infsof.2020.106347.

BIOGRAPHIES OF AUTHORS

Shahbaa I. Khaleel was born in Mosul, Nineveh, Iraq. She received the B.S., M.Sc. and Ph.D. degrees in computer science from Mosul University, in 1994, 2000 and 2006, respectively, assistant prof. since 2011, and finaly, prof. degree on 2021. From 2000 to 2020, she was taught computer science, software engineering and techniques in the College of Computer Sciences and Mathematics, University of Mosul. She has research in a field computer science, software engineering, and intelligent technologies. She can be contacted at email: shahbaaibrkh@uomosul.edu.iq.

Ghassan Khaleel Al-Khatouni received his Bachelor's degree in Software Engineering from the University of Mosul, in Iraq 2007. And he now a student who is a master's in the research stage, he taught computer subject in secondary schools for the years 2010 to 2020. He can be contacted at email: chassan.kalel.1984@gmail.com.