
Journal of Informatics and Web Engineering

Vol. 4 No. 3 (October 2025)

eISSN: 2821-370X

Improving Code Effectiveness Through Refactoring: A Case Study

**Abdullah Almogahed^{1*}, Manal Othman², Mazni Omar³, Samera Obaid Barraood⁴,
Abdul Rehman Gilal⁵**

¹Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia, Jalan Delta 1/6, 86400 Parit Raja, Johor, Malaysia.

^{2,3}School of Computing, Universiti Utara Malaysia, Sintok, 06010 Bukit Kayu Hitam, Kedah, Malaysia.

⁴Faculty of Computers and Information Technology, Hadhramout University, Mukalla, Hadhramout, Yemen.

⁵Knight Foundation School of Computing and Information Sciences, Florida International University, 11200 SW 8th St CASE 352, Miami, Florida 33199, United States.

**corresponding author: (abdullahm@uthm.edu.my; ORCID: 0000-0001-5408-1529)*

Abstract - Software refactoring is a crucial practice in modern software development methodologies, such as Agile and DevOps, as it enables teams to iteratively improve and evolve their codebases while minimizing the risk of introducing bugs or regressions. It fosters a culture of continuous improvement and code hygiene, ultimately leading to more robust, maintainable, and scalable software systems. However, research examining the impact of refactoring on code effectiveness is scarce. This study, therefore, seeks to investigate the impact of refactoring methods on the code's effectiveness. The study was carried out in four distinct phases: refactoring methods selection, case study selection, software metric selection for evaluating the effectiveness of the code, and refactoring methods implementation. The five most prevalent refactoring methods (Extract Subclass, Extract Class, Introduce Parameter Object, Extract Method, and Move Method) were chosen and implemented in the jHotDraw case study. The refactoring methods were implemented 86 times across five experiments in the jHotDraw case study. The results indicate that Extract Subclass, Extract Class, and Introduce Parameter Object have a significant positive impact on code effectiveness, while Extract Method and Move Method do not affect code effectiveness. Practitioners and software designers can utilize this knowledge to make informed assessments regarding refactoring methods and produce software systems that are more reliable and effective.

Keywords— *Refactoring, Refactoring Methods, Code Effectiveness, Software Quality, Software Metrics, Software Maintenance.*

Received: 14 May 2025; Accepted: 20 August 2025; Published: 16 October 2025

This is an open access article under the [CC BY-NC-ND 4.0](#) license.



1. INTRODUCTION

Software refactoring refers to the process of restructuring existing code without changing its external behaviour [1]. It involves making modifications to the internal structure of software systems to improve their design, readability, maintainability, and overall quality without altering the functionality perceived by end-users [2]. The primary

objective of refactoring is to enhance the codebase's clarity, simplicity, and extensibility, thereby facilitating easier maintenance, evolution, and adaptation to changing requirements over time [3]. Refactoring eliminates duplicated code segments and addresses technical debt—accumulated inefficiencies and suboptimal design decisions [4]. By removing duplicate code and refactoring to cleaner, more maintainable structures, refactoring reduces the risk of errors, inconsistencies, and future maintenance challenges [5]. This reduction in technical debt improves code effectiveness by ensuring that the codebase remains adaptable, scalable, and easy to maintain over time [6].

In the refactoring process, developers usually make many small, incremental changes, which are guided by established patterns, principles, and good practices [7]. Some examples of typical refactoring techniques are renaming the variables, splitting a method or class into several methods or classes, dealing with duplicated code blocks, and rearranging the code segments to improve the readability or modularity [8]. Such particular methods as code refactoring, object-oriented design, and maximizing cohesion/minimizing coupling can be used by developers to improve the code structure and organization, as well as to get rid of the code smells and decrease the technical debt [9]. However, refactoring has a crucial role in the enhancement of the quality of the software code by improving code readability, modularity, flexibility, and maintainability [10]. Through the improvement of code quality and the adaptation of the software in small and incremental ways, refactoring helps the creation of a culture centred on continuous improvement and engineering-skilled software development. Nevertheless, the improvement that is widely recognized as necessary for code quality refactoring is still a topic of research [11, [12].

Code effectiveness refers to the ability of software code to achieve its intended objectives efficiently and reliably [13], [14]. It encompasses various aspects of code quality, performance, and functionality, with the goal of delivering value to end-users and stakeholders [15], [16]. Effective code meets the functional requirements specified by stakeholders, providing the expected features, capabilities, and behaviours. It also effectively executes the requested functionality without bugs or divergence in intended behaviour [17], [18].

A good code is efficient and runs at optimum performance (fast, responsive, and resource-friendly), which directly contributes to improved maintainability, scalability, and user satisfaction, and reduces long-term development costs [19], [20], [21]. Additionally, good code has a small computational overhead, latency, and memory usage to guarantee fluid and snappy performance [22], [23], [24]. A properly written code will have some security mechanisms that help it combat vulnerabilities, threats, and attacks [25], [26], [27]. It adheres to safe programming principles, uses encryption, authentication, and authorization protocols, and blocks typical security threats [28], [29].

Past research has already discovered valuable information concerning the effect of refactoring on different software quality features, and how it stands as a significant activity conducted in the field of software engineering to enhance maintainability, reliability, performance, and scale of software systems [30], [31], [32]. These results are essential in stressing the need to integrate refactoring practices in software development processes to make software sustainable and viable in the long run. The impacts of refactoring on different quality attributes of software systems, specifically, many researchers have studied maintainability, and they have revealed more about the advantages, challenges, and best practices for refactoring [33-36].

However, despite these insights, the direct effect of refactoring on code effectiveness remains underexplored. Recent studies [33-38] highlight that while modern refactoring tools and techniques are increasingly integrated into development workflows, there is still a lack of quantitative, empirical evidence linking specific refactoring methods to measurable improvements in code effectiveness. Addressing this gap is critical for both researchers and practitioners. For academia, it provides a deeper understanding of how refactoring influences fundamental internal design properties. For the industry, it offers evidence-based guidance on selecting and applying refactoring methods to achieve tangible quality gains. Therefore, this study investigates the role of specific refactoring methods in improving code effectiveness, using a structured experimental approach and well-established quality metrics.

The following is the outline for the remainder of the paper. In Section 2, the relevant work is detailed. The research methodology is described in more detail in Section 3. In Section 4, the findings are discussed, and in Section 5, the conclusion is drawn.

2. LITERATURE REVIEW

Several studies have been conducted to investigate the impact that various refactoring methods have on the quality of software. [32] investigated the relationship between security and refactoring in real-world scenarios by analysing

refactoring activities that were performed at the same time as vulnerability remedies. The data indicates that programmers involve refactoring efforts in their fixes, with 31.9% of the risks being linked to refactoring processes. [39] analysed how automated refactoring affects the ability to be maintained (the maintainability) across five projects in the industry. Except for one project, the results revealed that refactoring for maintainability enhanced it. [40] proposed a refactoring navigation technique that uses the given implementation as a starting point and the intended design as a goal. Subsequently, a sequence of repetitive refactoring procedures was proposed to get the intended design. The design quality was assessed utilizing the coupling between objects (CBO) and the lack of cohesion in methods (LOCM) metrics. A case study was undertaken in the industry to assess the efficacy of the technique, and the findings indicated that the approach may be beneficial for practical refactoring.

[41] performed an investigation in industrial environments to evaluate whether the process of clean code refactoring contributed to enhancing developers' productivity in terms of comprehensibility. The researchers noted that the comprehensibility did not consistently increase due to the developers' varying coding approaches. [42] performed a study to investigate whether the practice of refactoring assists in reducing maintenance efforts and, thus, decreases both defects and change vulnerability. The code that caused the defect and the modification for the same set of refactored components were studied to see how refactoring decreased the occurrence of faults or changes. Furthermore, the examined components were contrasted with the components in the project that were not refactored simultaneously. The findings demonstrated a considerable decrease in both fault-proneness and change-proneness inside the refactored components.

[43] introduced a multifaceted search-based approach to automate 11 refactoring procedures. The objective was to identify a suitable series of refactoring strategies that might enhance the quality by minimizing design flaws. An experiment was conducted to evaluate this method using open-source projects and 11 refactoring techniques. The quality model for object-oriented design (QMOOD) was used to estimate the effect of refactoring on four external quality attributes (understandability, reusability, effectiveness, and flexibility). In addition, validation was conducted using the industrial system that their industrial partner offered. The results showed that the quality attributes of the software were improved by this method.

[44] performed a case study where they implemented refactoring on an application in order to enhance its maintenance capacity. There were several deficiencies in the initial version of this application with respect to maintainability. The refactoring efforts were conducted by combining the findings of the computerized code analysis with the suggestions provided by the developers. These evaluations were carried out to evaluate the efficacy of refactoring strategies. The findings demonstrated that the maintenance process was enhanced to minimize the occurrence of duplicate codes.

[45] examined the impact of refactoring on some external quality attributes: understandability, maintainability, modifiability, and analysability. The research was carried out at the architectural level, specifically focusing on the Unified Modelling Language (UML) class diagram. The refactoring strategies of Extract Method, Move Method, and Extract Class were applied to a total of nine minor projects. The findings indicated that applying three refactoring strategies across nine small projects resulted in improvements to four external quality characteristics (understandability, maintainability, modifiability, and analysability). Nevertheless, the analysis is restricted to just three refactoring strategies.

[46] examined the effects of 11 refactoring strategies on some well-known quality attributes: coupling, complexity, cohesion, inheritance, and size. The version histories of 23 applications were examined. The Miner tool was utilized to find refactoring strategies, while the Understanding tool was utilized to determine the pre- and post-refactoring values of each measure. Twenty students manually validated the Miner tool's findings using random samples. The data revealed that 65% of quality indicators were improved, while 35% remained unchanged.

[47] investigated empirically the effect of clone refactoring on complexity, coupling, size, and test code size. An open-source project called Ant was used to perform clone refactoring, and software metrics CBO, Weighted methods for Class (WMC), lines of code (LOC) were used to assess the improvement. Results of refactored classes indicated that the complexity, coupling, and size attributes were improved, and test code size was reduced. However, the use of just one system is inadequate for extrapolating the acquired findings. Furthermore, just a few metrics were analysed.

[48] conducted a study on the impact of five often-used refactoring strategies (Encapsulate Field, Hide Method, Inline Method, Remove Setting Method, and Extract Method) on the security attribute, specifically in relation to

information concealing. The study included five projects. Subsequently, a comprehensive study was performed on the five projects to classify the refactoring strategies according to security metrics such as classified class data accessibility (CCDA), classified operation accessibility (COA), and classified instance data accessibility (CIDA). The suggested classification sought to assist software engineers in choosing suitable refactoring strategies to enhance software security. [49] examined the impact of refactoring on security measures, including information flow and access control. They utilized the security measures suggested by [50]. Table 1 summarizes the relevant studies discussed above.

Table 1. Summary of Relevant Studies on the Impact of Refactoring on Software Quality

Year	Refactoring Methods Investigated	Quality Attributes Measured	System Type	Main Findings	Limitations / Research Gap
[32]	Refactoring is linked to vulnerability fixes	Security	Real-world projects	31.9% of vulnerabilities are linked to refactoring	No measurement of maintainability or effectiveness
[39]	Automated refactoring (various)	Maintainability	5 industrial projects	Maintainability improved in 4/5 projects	Limited to automated methods; did not examine code effectiveness
[40]	Sequence-based refactoring navigation	CBO, LCOM	Industrial case study	Improved design quality metrics	Focused on navigation technique, a narrow set of metrics
[41]	“Clean code” refactoring	Comprehensibility	Industrial environment	Mixed results: comprehensibility not always improved	Developer style influenced results; no focus on effectiveness
[42]	Various refactoring	Fault-proneness, change-proneness	Industrial	Significant reduction in faults and changes	Limited metrics; no direct measurement of internal design properties
[43]	11 automated refactoring methods	Understandability, reusability, effectiveness, flexibility (QMOOD)	Open source & industrial	Improved all measured attributes	Broad focus; code effectiveness considered but not deeply analysed
[44]	Various (developer + tool suggestions)	Maintainability	Industrial	Reduced duplicate code, improved maintainability	Single project; limited generalizability
[45]	Extract Method, Move Method, Extract Class	Understandability, maintainability, modifiability, and analysability	9 small projects	All attributes improved	Small scale; only three techniques
[46]	11 methods	Coupling, complexity, cohesion, inheritance, size	23 applications	65% of quality indicators improved	Some attributes unchanged; no link to code effectiveness
[47]	Clone refactoring	Complexity, coupling, size, test code size	1 open-source system (Ant)	All metrics improved; reduced test code size	Single system, few metrics
[48]	5 methods	Security metrics (CCDA, COA, CIDA)	5 projects	Classified techniques for security improvement	Focused only on security, not effectiveness
[49]	Various	Security (information flow, access control)	Not specified	Improved security	Narrow attribute focus

The influence of refactoring methods on key quality attributes such as understandability, extendibility, flexibility, reusability, and code effectiveness has not been thoroughly researched [33]. This is due to a lack of comprehensive research. They leave behind a question that must be answered with additional research on the impact of refactoring on these particular qualities [34]. Our study objective is to address this identified gap in research by examining the effect that refactoring methods have on the code's effectiveness, specifically.

3. RESEARCH METHODOLOGY

The research methodology used in this study entails a more systematic procedure that includes four phases. All phases are well elaborated to make sure that the research purposes are fully investigated and the code effectiveness under the influence of refactoring methods is studied in detail. Four phases are presented as follows:

3.1 Selecting Refactoring Methods

A comprehensive review of existing refactoring methods and techniques is conducted at this phase [5] [7] [33-34]. This involves examining literature, studying best practices in software engineering, and consulting with domain experts to identify a diverse range of refactoring methods. The selection criteria for refactoring methods may include their relevance to the research objectives, applicability to the chosen case study, and potential to impact code effectiveness metrics. A final set of refactoring methods is then chosen based on the criteria established, ensuring that the selected methods represent a well-rounded and representative sample of refactoring methods [2] [9] [31-38]. The selected refactoring methods are Extract Subclass, Extract Class, Introduce Parameter Object, Extract Method, and Move Method [51-52].

To sum up, the selection of refactoring methods followed several steps involving:

- a. Conducting a comprehensive literature review of foundational refactoring works.
- b. Extracting industry best practices and domain experts from the literature review to assess practical relevance.
- c. Establishing selection criteria (relevance, applicability, diversity).
- d. Applying the criteria to shortlist and finalize the five selected methods.

3.2 Selecting a Case Study

During this phase, the jHotDraw case study [53] was chosen due to the aim of this study. It is an appropriate case study under which the study will be carried out as the experimental subject [35-37]. The jHotDraw is a software project or a codebase of the real world with characteristics that are interesting to the research aims. The jHotDraw contains 250 classes and 14866 LOC. In the selection of the jHotDraw case study, factors to be considered are the size and complexity of the codebase, availability of historical data/documentation, and the possibility of doing refactoring experiments under the given study constraints. The jHotDraw offers some examples of how to implement the selected refactoring practices and test their influence on the quality measures in the code.

Here is a summary of the steps taken for the case study selection process:

- a. Identifying potential open-source Java projects.
- b. Evaluating them against defined criteria (size/complexity, documentation, refactoring potential).
- c. Performing a preliminary analysis to confirm suitability.
- d. Selecting jHotDraw as the case study based on its strong fit with the study's aims and constraints.

3.3 Selecting Metrics to Assess Code Effectiveness

This phase carries with it an aspect of attempting to define a set of metrics to measure the quality of the code before and after the implementation of refactoring methods on the code. Five internal design properties (abstraction, encapsulation, composition, inheritance, and polymorphism) and five well-known measures, namely Average Number on Ancestors (ANA), Data Access Metric (DAM), Measure of Aggregation (MOA), Measure of Functional Abstraction (MFA), and Number of Polymorphic Methods (NOP) can be used to measure the code effectiveness [35-

37] [54]. ANA measures the level of abstraction by calculating the average number of ancestor classes in the inheritance hierarchy. DAM measures encapsulation by determining the ratio of private and protected attributes to the total number of attributes in a class. MOA measures composition by counting the number of attributes whose types are user-defined classes. MFA measures inheritance by computing the ratio of inherited methods to the total number of available methods in a class. NOP measures polymorphism by counting the methods that can be overridden in descendant classes. As depicted in Figure 1, the effectiveness of the code can be measured.

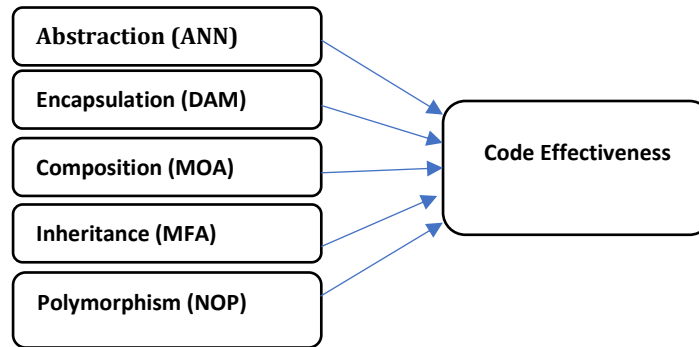


Figure 1. Code Effectiveness Measurements

These measures represent important elements of code effectiveness. The identified metrics are meaningful to the research questions and responsive to the modifications associated with the adoption of refactoring methods. Additionally, they are quantifiable and reproducible to ensure the reliability of the experimental results. Based on these metrics, the code effectiveness is calculated by the following formula [54]:

$$\text{Code Effectiveness} = 0.2 * ANA + 0.2 * DAM + 0.2 * MOA + 0.2 * MFA + 0.2 * NOP \quad (1)$$

This formula is used to compute the code effectiveness before and after applying the refactoring methods to assess their impact on the code effectiveness. The Eclipse Metrics tool [55] was used to automatically collect the software metrics.

3.4 Applying Refactoring Methods

In the final phase, the selected refactoring methods are applied to the chosen jHotDraw case study, following a systematic and controlled process, as shown in Figure 2. Each refactoring method is applied iteratively, with careful documentation of the changes made and their rationale. Throughout the refactoring process, the chosen metrics are measured and recorded to track changes in code effectiveness. Comparative analysis is performed between the pre-refactoring and post-refactoring states to assess the impact of the applied methods. The results of the refactoring experiments are analysed and interpreted to draw conclusions regarding the effect of the selected refactoring methods on improving code effectiveness.

Extract Class was applied to classes with multiple responsibilities, separating relevant fields and methods into a new class, and updating associations accordingly. Extract Method was used to find long/complicated methods and keep them as rational blocks of code in new methods, using clear and descriptive names, to create easy-to-read and easy-to-reuse routines. Extract Subclass was used in cases where the classes contained subsets of features that could be generalized, and subclasses were outlined to recapture these characteristics, and inheritance relationships were modified. The Introduce Parameter Object aspect was implemented to refactor and replace a series of correlated parameters in the methods of a parameter with a single parameter that is an object. The Move Method was used to refactor methods to classes in which they belong to a more relevant context or are used more often with data, i.e., are more encapsulated and have the data responsibility more centralized. A JDeodorant Tool [56] was used to help refactor the Extract Class and Extract Method methods across the jHotDraw efficiently and consistently. Extract Subclass, Introduce Parameter Object, and Move Method were implemented along the manual refactoring path because there were no automated refactorings to address the manual intervention.

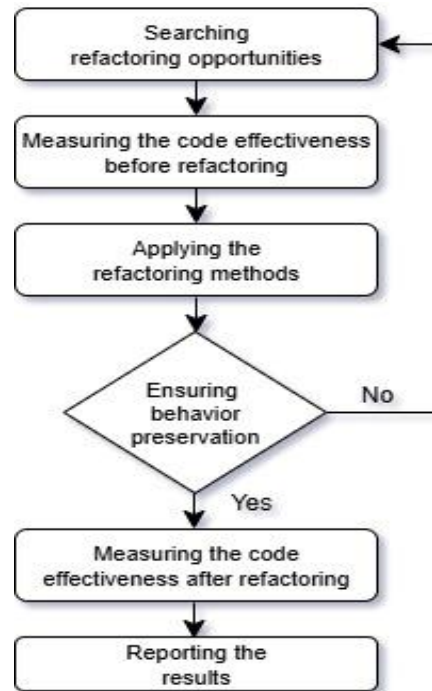


Figure 2. Steps of Applying Refactoring

4. RESULTS AND DISCUSSIONS

This section presents and analyses the results obtained from implementing the five refactoring methods in the jHotDraw case study in order to determine their impact on the code's effectiveness. Prior to implementing the refactoring, the code effectiveness was calculated using the metric values that were gathered and are presented in Table 2.

Table 2. Metrics and Code Effectiveness Values before Refactoring in jHotDraw Case Study

Metrics	Values
ANA	2.268
DAM	156.033
MOA	138
MFA	1
NOP	161
Effectiveness	91.660

Subsequently, the five refactoring methods were implemented a total of 86 times across five experiments in the jHotDraw case study, distributed as follows. The operation Extract Subclass was executed 8 times, Extract Class 13 times, Introduce Parameter Object 8 times, Extract Method 51 times, and Move Method 6 times. Table 3 displays the effectiveness values before and after implementing the refactoring methods.

The results of applying the refactoring methods are depicted in Figure 3. The code's effectiveness is demonstrated both before and after the implementation of each refactoring method.

Table 3. Metrics and Code Effectiveness Values after Refactoring in jHotDraw Case Study

Refactoring Methods	ANA	DAM	MOA	MFA	NOP	Effectiveness
Extract Subclass	2.342	166	138	1	175	96.475
Extract Class	2.192	171	154	1	163	98.245
Introduce Parameter Object	2.229	164	144	1	159	94.0524
Extract Method	2.268	156	138	1	161	91.660
Move Method	2.268	156	138	1	161	91.660

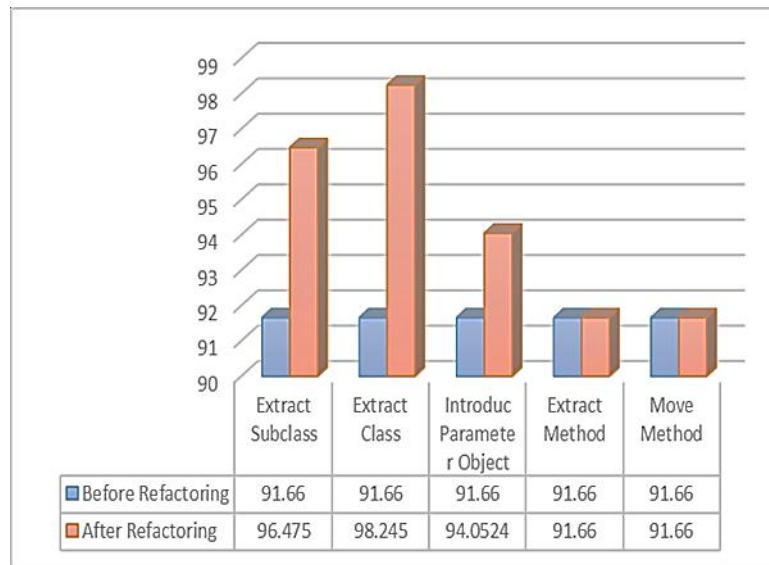


Figure 3. Effects of Refactoring on Code Effectiveness

The percentages in Figure 3 represent the measured code effectiveness values before and after applying five different refactoring techniques. For example, the Extract Class improved the effectiveness score from 91.66% to 98.245%, showing the highest gain among all methods tested. Similarly, Extract Subclass increased from 91.66% to 96.475% and Introduce Parameter Object rose from 91.66% to 94.0524%. In contrast, the Extract Method and Move Method did not show any measurable improvement, maintaining their pre-refactoring value of 91.66%.

The results indicated that the methods of Extract Subclass, Extract Class, and Introduce Parameter Object enhance the effectiveness of the code. Neither the Extract Method nor the Move Method has any impact on the code's effectiveness. Refactoring has a significant impact on code effectiveness, influencing various aspects that contribute to the overall quality, efficiency, and reliability of software code.

Based on the results obtained, the Extract Class and Extract Subclass refactoring methods break up intricate, challenging code blocks so they become more understandable and maintainable. By removing unnecessary complexity, redundant logic, and confusing types, these refactoring methods give more clarity and readable code by making it so. This results in enhanced readability and effectiveness of the code for developers, which makes it possible for them to rapidly comprehend the objective of the code, eventually making the troubleshooting, debugging, and modification processes more effective.

Splitting and refactoring such big code blocks into small and neat units using the Extract Class and Extract Subclass refactoring methods incorporates the design principle of modularity. On the other hand, these modular elements are enclosed, and therefore, they cover only the necessary interfaces as opposed to showing their internal details. This will boost code effectiveness through the provision of code reusability, enhanced by the design principle of separation

of concerns. Moreover, the maintainability is improved, and the individual units can be thoroughly tested and debugged.

The results revealed that the Introduce Parameter Object refactoring method improves the code's effectiveness. This refactoring method goes through the codebase in order to remove duplicated segments of code and build solid design decisions. By deleting duplicated code and rearranging everything in a cleaner, more maintainable way, the lack of errors, inconsistencies, and issues in future improvements will be lessened. This elimination of technical debt increases code efficiency as it ensures that the codebase can run, grow, and be simple to maintain.

The need for refactoring is strong and important, and codebases that are highly dynamic and prone to frequent changes may benefit significantly from it. By breaking code into its own parts, modular, flexible, and loosely coupled terms, refactoring allows developers to make changes more easily and with less risk due to unintended side effects. The improved flexibility warrants code effectiveness, ensuring that the software adapts accordingly to the next requirements without compromising the stability and the quality of the final product.

Refactoring not only avoids inefficiencies but also minimizes the overall computational burden, and this reduces algorithmic complexity, hence leading to performance optimization. In this way, refactoring code so that it is not only more efficient but also scalable also has a positive effect on the software's performance and scalability. The optimization mentioned here, through which software can continue delivering its performance even when there is an increase in the number of users, load of data, and complexity, is to ensure it does not compromise performance or stability.

Refactoring allows coding to standards and best practices and permits the application of pattern design, which instills consistency, uniformity, and design for the codebase. The refactoring practice minimizes the possibility of new errors, inconsistencies, and maintenance complications by providing a unified approach and a common standard code. Consistency implies simplicity, since these codebases are easy to understand, maintain, and extend. The result is a high-quality code that is expressed with effectiveness.

In summary, refactoring changes the code quality in several aspects – first, making the code easier to understand; second, dividing it into reusable modules; third, granting more flexibility; fourth, making it run faster and keeping it up to date with the coding standards. Refactoring enhances the effectiveness and quality of software code because it enables more dependable, extensible, and maintainable software systems on the premise of the systematic refactoring of the code that leads to clearer, simpler-to-maintain, and efficient code.

5. CONCLUSION

Code effectiveness pertains to the capacity of software code to reliably and efficiently accomplish its designated goals. Code quality, performance, and functionality are all components of this concept, which aims to provide value to stakeholders and end-users. This paper aims to enhance the comprehension of the influence of refactoring on the effectiveness of code by examining existing literature and industry practices. Through an analysis of the precise impact of refactoring methods on effectiveness, the goal was to offer valuable insights to developers who wish to harness the potential of effectiveness in their projects.

This study aims to study how refactoring methods affect code effectiveness. Five common refactoring methods (Extract Subclass, Extract Class, Introduce Parameter Object, Extract Method, and Move Method) were selected to investigate their effect on code effectiveness across the jHotDraw case study. The five refactoring methods were implemented 86 times in five different experiments on jHotDraw. The findings suggest that Extract Subclass, Extract Class, and Introduce Parameter Object exert a substantial positive influence on code effectiveness, whereas Extract Method and Move Method do not impact code effectiveness.

The Extract Method and Move Method had no effect on the code's effectiveness because they changed no metrics after being applied. It is clear that moving methods between classes and packages will not affect the metrics. On the other hand, extracting the methods is expected to improve the code's effectiveness. However, we did not see their effects in this study. That can be interpreted through the metrics used to measure effectiveness, which focus on the classes rather than the methods. Thus, it is strongly advised that the effect of the Extract Method on code effectiveness be investigated using other software metrics suites.

This knowledge can be utilized by practitioners and software designers in order to make informed assessments regarding refactoring methods and to produce software systems that are more reliable and effective. In future investigations, additional refactoring methods such as Extract Superclass, Extract Interface, Move Field, Remove Setting Method, and Hide Method can be employed to assess their influence on the code effectiveness across various case studies.

ACKNOWLEDGEMENT

The authors would like to thank the Faculty of Computer Science and Information Technology (FSKTM) at Universiti Tun Hussein Onn Malaysia (UTHM) for their generous provision of facilities. The authors would like to also thank the anonymous reviewers for their valuable comments.

FUNDING INFORMATION

The authors received no funding from any party for the research and publication of this article.

AUTHOR CONTRIBUTIONS

Abdullah Almogahed: Project Administration, Supervision, Conceptualization, Data Curation, Methodology, Validation, Writing – Original Draft Preparation;
Manal Othman: Conceptualization, Data Curation, Methodology, Validation;
Mazni Omar: Methodology, Validation, Writing – Review & Editing;
Samera Obaid Barraood: Methodology, Validation;
Abdul Rehman Gilal: Methodology, Validation.

CONFLICT OF INTERESTS

No conflict of interest were disclosed.

ETHICS STATEMENTS

This study did not involve human participants, animal experiments, or data from social media platforms, and therefore no ethical approval or informed consent was required. Our publication ethics follow The Committee of Publication Ethics (COPE) guideline. <https://publicationethics.org/>.

REFERENCES

- [1] M. Fowler and K. Beck, Refactoring: Improving the Design of Existing Code, 2nd ed. Boston, MA, USA: Addison-Wesley Professional, 2019.
- [2] A. Almogahed, H. Mahdin, M. Omar, N. H. Zakaria, G. Muhammad, and Z. Ali, “Optimized refactoring mechanisms to improve quality characteristics in object-oriented systems,” IEEE Access, vol. 11, pp. 99143–99158, 2023, doi: 10.1109/ACCESS.2023.3313186.
- [3] E. A. AlOmar et al., “How do developers refactor code to improve code reusability?,” in Lecture Notes in Computer Science, vol. 12541. Cham, Switzerland: Springer, 2020, doi: 10.1007/978-3-030-64694-3_16.




- [4] H. Mumtaz, P. Singh, and K. Blincoe, "Identifying refactoring opportunities for large packages by analyzing maintainability characteristics in Java OSS," *J. Syst. Softw.*, vol. 202, no. 111717, 2023, doi: 10.1016/j.jss.2023.111717.
- [5] C. Abid, V. Alizadeh, M. Kessentini, N. Ferreira, and D. Dig, "30 years of software refactoring research: A systematic literature review," *arXiv preprint arXiv:2007.02194*, 2020, doi: 10.48550/arXiv.2007.02194.
- [6] Y. Zhao, Y. Yang, Y. Zhou, and Z. Ding, "DEPICTER: A design-principle guided and heuristic-rule constrained software refactoring approach," *IEEE Trans. Reliab.*, vol. 71, no. 2, pp. 698–715, Jun. 2022, doi: 10.1109/TR.2022.3159851.
- [7] G. Lacerda, F. Petrillo, M. Pimenta, and Y. Gaël, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *J. Syst. Softw.*, vol. 167, no. 110610, 2020, doi: 10.1016/j.jss.2020.110610.
- [8] E. Fernandes et al., "Refactoring effect on internal quality attributes: What haven't they told you yet?," *Inf. Softw. Technol.*, vol. 126, 2020, doi: 10.1016/j.infsof.2020.106347.
- [9] Almogahed, M. Omar, and N. H. Zakaria, "Recent studies on the effects of refactoring in software quality: Challenges and open issues," in *Proc. 2nd Int. Conf. Emerg. Smart Technol. Appl. (eSmarTA)*, 2022, pp. 1–7, doi: 10.1109/eSmarTA56775.2022.9935361.
- [10] A. Almogahed et al., "Revisiting scenarios of using refactoring techniques to improve software systems quality," *IEEE Access*, vol. 11, pp. 28800–28819, 2023, doi: 10.1109/ACCESS.2022.3218007.
- [11] A. Almogahed et al., "Empirical investigation of the diverse refactoring effects on software quality: The role of refactoring tools and software size," in *Proc. 3rd Int. Conf. Emerg. Smart Technol. Appl. (eSmarTA)*, 2023, pp. 1–6, doi: 10.1109/eSmarTA59349.2023.10293407.
- [12] F. Palomba, A. Zaidman, R. Oliveto, and D. A. Lucia, "An exploratory study on the relationship between changes and refactoring," in *Proc. 25th Int. Conf. Program Comprehension (ICPC)*, IEEE, 2017, doi: 10.1109/ICPC.2017.38.
- [13] G. Bavota et al., "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, pp. 1–14, 2015, doi: 10.1016/j.jss.2015.05.024.
- [14] A.S. Nyamawe, "Mining commit messages to enhance software refactorings recommendation: A machine learning approach," *Mach. Learn. Appl.*, vol. 9, 2022, doi: 10.1016/j.mlwa.2022.100316.
- [15] V. Alizadeh et al., "RefBot: Intelligent software refactoring bot," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2019, pp. 823–834, doi: 10.1109/ASE.2019.00081.
- [16] L. Sousa et al., "Characterizing and identifying composite refactorings: Concepts, heuristics and patterns," in *Proc. 17th Int. Conf. Mining Softw. Repositories (ICSE)*, 2020, Seoul, Korea, doi: 10.1145/3379597.3387477.
- [17] A. Almogahed, M. Omar, and N. H. Zakaria, "Categorization refactoring techniques based on their effect on software quality attributes," *Int. J. Innov. Technol. Exploring Eng. (IJITEE)*, vol. 8, no. 8S, pp. 439–445, 2019. [Online]. Available: <https://www.ijitee.org/wp-content/uploads/papers/v8i8s/H10760688S19.pdf>
- [18] A. Almogahed et al., "Empirical studies on software refactoring techniques in the industrial setting," *Turkish J. Comput. Math. Educ. (TURCOMAT)*, vol. 12, no. 3, pp. 1705–1716, 2021, doi: 10.17762/turcomat.v12i3.995.



- [19] A. Almogahed et al., “Software security measurements: A survey,” in Proc. Int. Conf. Intell. Technol. Syst. Service Internet Everything (ITSS-IoE), 2022, pp. 1–6, doi: 10.1109/ITSS-IoE56359.2022.9990968.
- [20] A. Almogahed, M. Omar, and N. H. Zakaria, “Impact of software refactoring on software quality in the industrial environment: A review of empirical studies,” in Proc. Knowl. Manag. Int. Conf. (KMICe), Sarawak, Malaysia, 2018, pp. 229–234. [Online]. Available: <http://soc.uum.edu.my/kmice/proceedings/proc/2018/pdf/CR61.pdf>
- [21] R. Faqih et al., “Empirical analysis of CI/CD tools usage in GitHub Actions workflows,” J. Informatics Web Eng., vol. 3, no. 2, pp. 251–261, 2024, doi: 10.33093/jiwe.2024.3.2.18.
- [22] D. Feitosa et al., “Code reuse in practice: Benefiting or harming technical debt,” J. Syst. Softw., vol. 167, no. 110848, 2020, doi: 10.1016/j.jss.2020.110848.
- [23] N. Mäkitalo et al., “On opportunistic software reuse,” Computing, vol. 102, pp. 2385–2408, 2020, doi: 10.1007/s00607-020-00833-6.
- [24] E. A. AlOmar et al., “Refactoring for reuse: An empirical study,” Innov. Syst. Softw. Eng., vol. 18, pp. 105–135, 2022, doi: 10.1007/s11334-021-.
- [25] R. Moser et al., “A case study on the impact of refactoring on quality and productivity in an agile team,” in Proc. 2nd IFIP Central Eur. Conf. Softw. Eng. Tech. (CEE-SET), 2007, pp. 252–266, doi: 10.1007/978-3-540-85279-7_20.
- [26] C. Dibble and P. Gestwicki, “Refactoring code to increase readability and maintainability: A case study,” J. Comput. Sci. Colleges, vol. 30, no. 1, pp. 41–51, 2014. [Online]. Available: <https://dl.acm.org/doi/abs/10.5555/2667369.2667378>
- [27] N. Niu et al., “Traceability-enabled refactoring for managing just-in-time requirements,” in Proc. IEEE 22nd Int. Requirements Eng. Conf. (RE), 2014, pp. 133–142, doi: 10.1109/RE.2014.6912255.
- [28] C. Abid et al., “X-SBR: On the use of the history of refactorings for explainable search-based refactoring and intelligent change operators,” IEEE Trans. Softw. Eng., vol. 48, no. 10, pp. 3753–3770, Oct. 2022, doi: 10.1109/TSE.2021.3105037.
- [29] R. Malhotra and J. Jain, “Analysis of refactoring effect on software quality of object-oriented systems,” in Proc. Int. Conf. Innov. Comput. Commun., 2019, pp. 197–212, doi: 10.1007/978-981-13-2354-6.
- [30] I. Alazzam, B. Abuata, and G. Mhediat, “Impact of refactoring on OO metrics: A study on extract class, extract superclass, encapsulate field and pull-up method,” Int. J. Mach. Learn. Comput., vol. 10, no. 1, 2020, doi: 10.18178/ijmlc.2020.10.1.913.
- [31] A. Almogahed and M. Omar, “Refactoring techniques for improving software quality: A practitioners’ perspectives,” J. Inf. Commun. Technol. (JICT), vol. 20, no. 4, pp. 511–539, 2021, doi: 10.32890/jict2021.20.4.3.
- [32] J. Abid et al., “How does refactoring impact security when improving quality? A security-aware refactoring approach,” IEEE Trans. Softw. Eng., vol. 48, no. 3, pp. 864–878, Mar. 2022, doi: 10.1109/TSE.2020.3005995.

- [33] Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 44–69, 2018, doi: 10.1109/TSE.2017.2658573.
- [34] S. Kaur and P. Singh, "How does object-oriented code refactoring influence software quality? Research landscape and challenges," *J. Syst. Softw.*, vol. 157, no. 110394, 2019, doi: 10.1016/j.jss.2019.110394.
- [35] A. Almogahed et al., "A refactoring classification framework for efficient software maintenance," *IEEE Access*, vol. 11, pp. 78904–78917, 2023, doi: 10.1109/ACCESS.2023.3298678.
- [36] A. Almogahed et al., "A refactoring categorization model for software quality improvement," *PLoS ONE*, vol. 18, no. 11, e0293742, 2023, doi: 10.1371/journal.pone.0293742.
- [37] A. Almogahed et al., "Multi-classification refactoring framework using Hopfield neural network for sustainable software development," *IEEE Access*, vol. 13, pp. 31785–31808, 2025, doi: 10.1109/ACCESS.2025.3542087.
- [38] Almogahed et al., "Code refactoring for software reusability: An experimental study," in *Proc. 4th Int. Conf. Emerg. Smart Technol. Appl. (eSmarTA)*, 2024, pp. 1–6, doi: 10.1109/eSmarTA62850.2024.10638872.
- [39] G. Szöke et al., "Empirical study on refactoring large-scale industrial systems and its effects on maintainability," *J. Syst. Softw.*, vol. 129, pp. 107–126, 2017, doi: 10.1016/j.jss.2016.08.071.
- [40] Y. Lin et al., "Interactive and guided architectural refactoring with search-based recommendation," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 535–546, doi: 10.1145/2950290.2950317.
- [41] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old habits die hard: Why refactoring for understandability does not give immediate benefits," in *Proc. 22nd IEEE Int. Conf. Softw. Anal. Evol. Reeng. (SANER)*, 2015, pp. 504–507, doi: 10.1109/SANER.2015.7081865.
- [42] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial C# software," *Sci. Comput. Program.*, vol. 102, pp. 44–56, 2015, doi: 10.1016/j.scico.2014.12.002.
- [43] A. Ouni et al., "A multi-objective refactoring approach to introduce design patterns and fix anti-patterns," in *Proc. North Amer. Search-Based Softw. Eng. Symp. (NASBASE)*, 2015. [Online]. Available: <https://kir.ics.es.osaka-u.ac.jp/lab-db/betuzuri/archive/990/990.pdf>
- [44] M. Wahler, U. Drofenik, and W. Snipes, "Improving code maintainability: A case study on the impact of refactoring," in *Proc. IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, 2016, pp. 493–501, doi: 10.1109/ICSME.2016.52.
- [45] R. S. Bashir et al., "A methodology for impact evaluation of refactoring on external quality attributes of a software design," in *Proc. Int. Conf. Front. Inf. Technol. (FIT)*, 2017, pp. 183–188, doi: 10.1109/FIT.2017.00040.
- [46] A. Chavez et al., "How does refactoring affect internal quality attributes? A multi-project study," in *Proc. 31st Brazilian Symp. Softw. Eng.*, 2017, pp. 74–83, doi: 10.1145/3131151.313117.
- [47] B. Mourad et al., "Exploring the impact of clone refactoring on test code size in object-oriented software," in *Proc. 16th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, 2017, pp. 586–592, doi: 10.1109/ICMLA.2017.00098.

- [48] A. Almogahed, M. Omar, and N. H. Zakaria, "Refactoring codes to improve software security requirements," *Procedia Comput. Sci.*, vol. 204, pp. 108–115, 2022, doi: 10.1016/j.procs.2022.08.013.
- [49] H. Mumtaz et al., "An empirical study to improve software security through the application of code refactoring," *Inf. Softw. Technol.*, vol. 96, pp. 112–125, 2018, doi: 10.1016/j.infsof.2017.11.010.
- [50] B. Alshammari, C. Fidge, and D. Corney, "A hierarchical security assessment model for object-oriented programs," in *Proc. 11th Int. Conf. Quality Softw.*, 2011, pp. 218–227, doi: 10.1109/QSIC.2011.31.
- [51] Rajput and A. Chug, "An inclusive survey on automation of refactoring: Challenges and opportunities," *Int. J. Syst. Assur. Eng. Manag.*, 2025, doi: 10.1007/s13198-025-02914-1.
- [52] Y. Zhang et al., "Move method refactoring recommendation based on deep learning and LLM-generated information," *Inf. Sci.*, vol. 697, p. 121753, 2025, doi: 10.1016/j.ins.2024.121753.
- [53] jHotDraw Files. [Online]. Available: <https://sourceforge.net/projects/jhotdraw/files> (accessed Jan. 12, 2025).
- [54] Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, 2002, doi: 10.1109/32.979986.
- [55] Metrics 3 – Eclipse Metrics Plugin Continued 'Again'. [Online]. Available: <https://github.com/qxo/eclipse-metrics-plugin> (accessed Sep. 9, 2024).
- [56] JDeodorant. [Online]. Available: <https://marketplace.eclipse.org/content/jdeodorant> (accessed Apr. 5, 2025).

BIOGRAPHIES OF AUTHORS

	<p>Abdullah Almogahed is an International Senior Lecturer at the Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia (UTHM), Malaysia. He holds a Ph.D. in Computer Science with a specialization in Software Engineering from Universiti Utara Malaysia (UUM). He has authored over 30 peer-reviewed articles in prestigious ISI and Scopus-indexed journals and conferences. His diverse research interests encompass software engineering, software refactoring, software quality, software measurement, software maintenance, software sustainability, wireless networks, data mining, and machine learning. He can be contacted at email: abdullahm@uthm.edu.my.</p>
	<p>Manal Othman holds a B.Sc. degree in Computer Science from Taiz University, Yemen, and the M.S. degree in Information Technology from the School of Computing, Universiti Utara Malaysia (UUM), Malaysia in 2022. She is currently pursuing a PH.D. degree in Computer Science, the School of Computing, Universiti Utara Malaysia (UUM), Malaysia. Her research interests include software engineering, software refactoring, machine learning, optimization, metaheuristics, Bio-inspired algorithms, and feature selection. She can be contacted at email: manal.oshari@taiz.edu.ye.</p>
	<p>Mazni Omar is currently an Associate Professor with the School of Computing (SOC) under the College of Arts and Sciences, Universiti Utara Malaysia (UUM), Malaysia. In addition, she is a Research Fellow with the Institute for Advanced and Smart Digital Opportunities (IASDO) under the SOC, UUM. She has published several articles in Scopus and indexed journals, conference papers, and other publications, such as books with chapters and technical reports. She also managed to secure several research grants from the university, as well as national, international, and industry grants. Her research interests include software engineering, knowledge management, and data mining. She can be contacted at email: mazni@uum.edu.my.</p>

	<p>Samera Obaid Barraood is currently a lecturer at the Department of Computer Science, College of Computers and Information Technology, Hadhramout University, Yemen. She has several publications that have been indexed by Scopus. She also presented several papers at international conferences. Her research interests include software engineering, software quality, testing quality, test case quality, and agile software development. She can be contacted at email: sammorahobaid@gmail.com.</p>
	<p>Abdul Rehman Gilal is currently an assistant teaching professor at the Knight Foundation School of Computing and Information Sciences, Florida International University, Miami, FL, United States. With extensive experience in teaching and research in computer science and software engineering, he has worked at esteemed institutions worldwide. His contributions to the field include postdoctoral research with University College Dublin, where he developed generic models and methodologies for assurance cases, addressing shared assets, components, and threats in software systems. He can be contacted at email: rehman_gilal33@yahoo.com.</p>