**MINAR** International Journal of Applied Sciences and Technology
ISSN: 2717-8234
Article type: Research Article

**Received:** 18/06/2023 **Accepted:** 12/07/2023 **Published:** 01/09/2023

# MEASURING MAINTAINABILITY INDEX BEFORE AND AFTER CODE REFACTORING

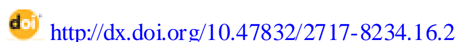
**Shahbaa I. KHALEEL**[1]
University of MosuL, Iraq

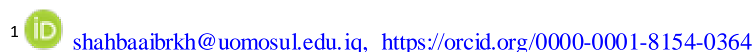
**Ghassan Khaleel AL-KHATOUNI**[2]
University of MosuL, Iraq

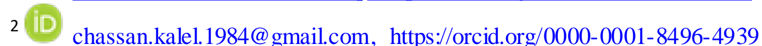
## Abstract

Measuring the maintainability index of software is crucial to ensure better maintenance and improve quality. Refactoring code is important in improving software quality and increasing maintainability. However, understanding the relationship between refactoring code and the maintainability index is crucial for software developers and maintenance engineers. An assistive software tool called MMIBAR was developed, which provides a set of metrics for calculating the maintainability index of the source code before and after refactoring code that contains cloned programming code. This research contributes to the understanding of the relationship between refactoring code and the maintainability index, and proposes a software tool that can help software developers and maintenance engineers improve software quality and increase maintainability. The research involved the development of the MMIBAR software tool and the application of its metrics to measure the maintainability index of the source code before and after refactoring code that contains cloned programming code. The results of the study demonstrate that refactoring code can significantly improve software quality and increase maintainability. The MMIBAR software tool provides a useful set of metrics for measuring the maintainability index of source code, and can be used to identify areas of code that need further refactoring. The research shows that refactoring code is crucial in improving software quality and increasing maintainability. The MMIBAR software tool can help software developers and maintenance engineers identify areas of code that need further refactoring, and ultimately improve the overall quality of software systems.

**Keywords**: *Maintainability; Maintainability Index; Code Refactoring; Cyclomatic Complexity.*


http://dx.doi.org/10.47832/2717-8234.16.2


[1] shahbaaibrkh@uomosul.edu.iq, https://orcid.org/0000-0001-8154-0364


[2] chassan.kalel.1984@gmail.com, https://orcid.org/0000-0001-8496-4939

**Introduction**

The ease with which software can be understood, modified, and tested is a significant factor in enhancing its quality, and this is referred to as maintainability(Macharia & Kimwele, 2022). The Maintainability Index (MI) is a measure of the maintainability of a software system, which can be evaluated using different metrics such as Lines of Code (LOC), Cyclomatic Complexity, and Halstead Volume(M.A.M.Najm, 2014). These metrics will be discussed in detail. By measuring the MI of a software system before and after code refactoring, refactoring techniques can be used to enhance the system's maintainability. This process can be useful in identifying code that may require further refactoring and improving the overall maintainability of the system. A higher MI score indicates that the system is more maintainable, while a lower MI score suggests that the system is less maintainable(Caro et al., 2007). MI is used to identify areas in the system that may require improvement to increase maintainability and helps developers prioritize refactoring efforts(Ouni et al., 2016).

Code cloning, also called duplicate code, refers to parts of code in a software system that are identical or similar and have the same function. This is seen as an issue in software development as it can result in higher maintenance costs, lower code quality, and difficulties in fixing errors. It can have negative effects on maintainability during software development by increasing code complexity, making it hard to comprehend and change, and expanding code size. To eliminate cloned code, developers use code refactoring techniques, which are common methods for extracting and getting rid of duplicated code(Tairas, 2006).

The research topics are structured and organized in a logical and systematic manner to ensure a comprehensive and cohesive study. The first section focuses on previous studies, which provides a background and context for the proposed research. The next section examines the concept of maintainability, which is an essential factor in software development. It highlights the significance of maintaining software systems to ensure their continued functionality and relevance. The subsequent section discusses code restructuring, which is the process of modifying existing software code to improve its quality, efficiency, and maintainability. The implementation of the proposed model follows, where the research presents the details of the software model designed to address the issues of maintainability and code restructuring. The next section involves a practical study of the employees' payroll management system, which serves as a case study for the proposed model. Finally, the research concludes by summarizing the findings, identifying the limitations of the research, and suggesting areas for future research. The overall structure and organization of the research topics provide a clear and concise understanding of the research and its significance.

## Literature Review

Maintenance is a crucial part of the software development life cycle. While maintaining a software, the code may undergo several changes, resulting in reduced code quality and the emergence of issues with cloned or duplicated code. To address these issues, it is essential to employ code refactoring techniques. The following are some previous works done in this area.

In 2010, Hegedűs et al examined how to predict quality characteristics, including maintainability, testability, and error proneness, based on measures that can be quantified in the source code, such as cohesion, code size, and complexity. They also investigated the effect of each refactoring approach on the computed metrics. Applying code restructuring is a reasonable and effective activity to facilitate maintenance and increase the value of program quality characteristics if refactoring operations are used correctly(Hegedűs et al., 2010).

In 2011, Dig introduced a new method for applying refactoring techniques to analyze and transform existing source code. The method suggests changing multiple lines of source code and eliminating errors to ensure parallel operations that increase program performance, maintainability, and portability. The research also presents a set of tools that support many refactoring activities, including increasing scalability and maintaining parallel operations, making processes harmless, and increasing the productivity of sequential operations(Dig, 2011).

In 2012, Meananeatra proposed a method for identifying the best sequence of code refactoring that satisfies several metrics such as maintenance factor, total number of resolved faults, sequence of refactoring dimensions, and total number of program components that have been modified. Additionally, the authors estimate that the results tend to reduce maintenance cost and time and improve program quality. The research method does not produce all refactoring sequences at once, but it regularly discovers a graphical representation of the sequences and uses the polishing method to remove the reverse sequence of refactoring to find the best sequence to be executed(Meananeatra, 2012).

In 2013, Fujiwara and colleagues proposed a technique for assessing refactoring procedures using version archives. The goal of this approach is to enhance software quality by improving maintainability. The method is implemented semi-automatically by analyzing software archives using two algorithms: UMLDiff, which detects differences in UML diagrams, and SZZ, which identifies bug-fixing changes. The researchers used the Columba project as a case study and found that refactoring cycles reduce the occurrence of defects and improve maintainability, as measured by three variables: frequency of refactoring, frequency of bug fixing, and defect density(Fujiwara et al., 2013).

In 2014, Chaparro et al introduced a technique called Refactoring Impact Prediction (RIP) to study the impact of refactoring processes on software code quality metrics. With this

technique, developers can assess the opportunities for refactoring in software maintenance tasks, and also compare the deviation values caused by refactoring processes, especially when refactoring involves multiple transformations and conflicting metrics evaluation of the source code(Chaparro et al., 2014).

In 2015, Han et al introduced the term MIS (Maximal Independent Set) which allows developers to identify multiple refactoring processes that can be executed at the same time. Each MIS has a set of refactoring paths that calculate a delta table, which represents the maintenance value for each initial path. In each round of the refactoring process, multiple operations can be applied to increase the maintenance value through sets of MIS. The proposed model was implemented in several case studies and the results show that it can increase the maintenance factor. Additionally, developers can apply multiple refactoring processes at the same time(Han et al., 2015).

In 2016, Malhotra and Chug conducted a study on the impact of software restructuring on maintainability using five proprietary software systems. They evaluated internal quality attributes using a set of design metrics, while external quality attributes such as understandability, abstraction, extensibility, modifiability, and reusability were assessed by experts. The original program versions were compared to the restructured versions, and changes in quality attributes for maintainability were analyzed. Results showed a significant improvement in program quality and expected lifespan with software restructuring and improvement. However, they also found that restructuring could be tedious and lead to errors if not carefully implemented. Therefore, they recommended frequent code restructuring to enhance maintainability while balancing engineering and over-engineering. The study's findings can assist management in identifying opportunities for restructuring while maintaining an ideal balance between engineering and over-engineering(Malhotra & Chug, 2016).

Mohan and Greer proposed a new way to automate software maintenance in 2017. This tool is capable of carrying out 26 different rebuilding processes and has a wide range of options for evaluating the impact of the rebuilding on the software. It also has six search-based methods for optimizing software, including both single-objective and multi-objective approaches. The tool has been fully automated and the researchers have highlighted its diverse abilities and unique features, while presenting results from a study. The effectiveness of various metrics has been tested on five different codebases to determine the best measures for improving software quality(Mohan & Greer, 2017).

In 2019, Mohan et al. present a study on a many-objective genetic algorithm for automating software refactoring, implemented as the Java tool Multi-Refactor. The tool incorporates four software quality measures, including code priority, element recency, refactoring coverage, and software quality metrics. The many-objective algorithm combines the four measures to improve software quality holistically. The study compares the many-objective method with a mono-objective approach using only one objective to measure

software quality. Several objective permutations were tested on six open-source Java programs. The many-objective approach provided more effective objective score values on average and was faster than the mono-objective method. However, the study found that element recency and priority measures had lower success rates when used in combination with other objectives in many-objective setups. The authors conclude that the many-objective approach is suitable for optimizing automated refactoring to improve software quality, although the addition of other objectives may be less efficient than using a mono-objective method(Mohan & Greer, 2019)

In 2020, Morales and others conducted an experimental study to investigate the effect of automated code refactoring on the system's understandability during comprehension tasks. They conducted a survey of 80 developers, asking them to identify a set of 20 refactoring changes in code that were produced either by a tool or by developers, as well as to provide a rating of the quality of the refactoring changes. They also asked 30 developers to perform code comprehension tasks on 10 systems that had been refactored either by an independent compiler or through automated refactoring tools. They measured the developers' performance using a NASA agency metric for their efforts, the time they spent on tasks, and the percentage of accurate responses. despite the current limitations imposed on technology, the results they found showed that it is reasonable to expect that a refactoring tool would match the developer's code. In fact, the results showed that for 3/5 of the anti-patterns studied, developers did not have the ability to identify the origin of the refactoring, whether it was executed through an automated tool or not. Additionally, they noted that developers do not prefer manual refactoring processes over automated refactoring processes(Morales et al., 2020).

In 2021, Draz et al. presents a comprehensive study on the role and effects of code refactoring in improving software quality. The study highlights positive improvements in the quality of code concerning internal attributes like complexity, inheritance, coupling, and cohesion, with the exception of size, after applying refactoring operations(Draz et al., 2021).

In 2022, Fernandes et al. introduces a Live Refactoring Environment, a real-time Integrated Development Environment (IDE) tool that identifies, suggests, and automates 'Extract Method' refactorings to enhance code quality and streamline programming solutions. To validate the effectiveness of this tool, an empirical experiment was conducted with 42 participants across three open-source projects. Results indicated that the live refactoring tool fosters an awareness of potential code flaws, promotes better quality software, and is comparatively more efficient than manual refactoring(S. Fernandes et al., 2022).

In 2023, Fernandes et al. presents an empirical study on the use of a Live Refactoring Environment to improve code quality. The tool is designed to detect "code smells," which are indicators of deeper problems in software code, and suggests refactorings in real time. The results showed that the Live Refactoring Environment increased developers' awareness,

leading to higher code quality and faster coding compared to manual refactoring(S. Fernandes et al., 2023).

Table 1, Summarizes the factors studied by the mentioned researchers in previous works and their respective impacts on quality characteristics.

Table (1) Summarizes the findings of the literature review

| Year | Authors | Factors Studied | Quality Characteristics Impacted |
|---|---|---|---|
| 2010 | Hegedűs et al | Cohesion, LOC, and complexity | Reconstruction, maintainability, testability, and fault identification |
| 2011 | Dig | A multiple circles of refactoring | Maintainability, navigability, productivity, scalability, and performance |
| 2012 | Meananeatra | A multiple circles of refactoring | Maintainability, total number of eliminated faults, sequence of reconstruction dimensions, and total number of changed software components |
| 2013 | Fujiwara et al | Version archives | Refactoring and maintainability |
| 2014 | Chaparro et al | RIPE “Refactoring Impact PrEdiction” | Code quality metrics (RFC, CBO, DAC, MPC, LOC, NOM, CYCLO, LCOM2, LCOM5, NOC, DIT( |
| 2015 | Han et al | MIS “Maximal Independent Set” applying multiple processes of refactoring simultaneously. | Refactoring and maintainability |
| 2016 | Malhotra and Chug | Abstraction level, comprehensibility, scalability, modifiability, and reusability. | Refactoring and maintainability |
| 2017 | Mohan and Greer | Evaluation of the approach using metrics such as QMOOD and CK, and improvement based on six different search algorithms. | MultiRefactoring and maintainability |
| 2019 | Mohan et al | Code prioritization measurement, reconstruction coverage, element novelty, and improvement of automatic reconstruction. | MultiRefactoring and maintainability |
| 2020 | Morales et al | "Rebuilding by Imitating People's Operations" (RePOR), an automated reconstruction approach based on partial demand reduction techniques. | Restructuring, understandability, and maintainability |
| 2021 | Draz et al | Complexity, Inheritance, Coupling, Cohesion | Improving software quality (except size) through code refactoring |

| | | | |
|---|---|---|---|
| | | | operations |
| 2022 | Fernandes et al | 'Extract Method' refactorings | Enhancing code quality and streamlining programming solutions through a Live Refactoring Environment |
| 2023 | Fernandes et al | 'Code Smells' detection and real-time suggestions | Increasing developers' awareness and improving code quality and coding efficiency compared to manual refactoring |

Through the above table, which contains 10 studies that have investigated and summarized the impact of software reconstruction and its effect on quality characteristics, especially maintainability. Through the review, the study summarizes the following points: (1) Applying reconstruction activities leads to an increase in the values of some quality characteristics, such as understandability and maintainability. (2) There are many factors that affect reconstruction activities, including cohesion, coupling, information hiding, and packaging. (3) Reconstruction helps improve the source code without changing the program's behavior. (4) Many reconstruction processes can be applied.

## MAINTAINABILITY

"Maintainability" is a term in software engineering that refers to the ease with which a software system or its components can be modified or maintained(Di Biase et al., 2019). This includes tasks such as bug fixing, adding new features, and adapting to new environments or requirements. Maintainability is extremely important because it directly affects the cost and efficiency of maintaining a software system throughout its life cycle(Heričko & Šumak, 2023). It also serves as a measure of the ease of modifying, updating, and repairing software systems. It is easy to change and update a software system that has high maintainability, which leads to reduced errors and maintenance costs. On the other hand, it is difficult to change and update a software system with low maintainability, which leads to higher maintenance costs and more errors(Gradišnik et al., 2020; Ogheneovo, 2014).

The Maintainability Index (MI) is a measure of the maintainability of a software system, and is strongly correlated with maintainability)Kaur & Singh, 2017(. MI is expressed as a numerical scale ranging from 0 to 100, where higher numbers indicate a higher level of maintainability. It is calculated using various metrics such as lines of code (LOC), code complexity (CC), and Halstead complexity(Hu et al., 2023). A high MI score means that the system is more maintainable, which leads to lower maintenance costs and improved efficiency, while a low MI score indicates the opposite. Modifying and maintaining systems with low MI scores may be more difficult, leading to increased maintenance costs

and reduced efficiency(Molnar & Motogna, 2020). To ensure maintainability, the MI for the software system should be improved(Hong-liang, 2020; M.A.M.Najm, 2014).

The maintenance index(MI) is determined by calculating various metrics such as Lines of Code (LOC), Halstead Volume (HV), Cyclomatic Complexity (CC), and Comment to Code Ratio (CM) )Heitlager et al., 2007(.

The maintenance index can be calculated using three different formulas (M.A.M.Najm, 2014):

- The original formula:

MI = 171 - 5.2 * ln(HV) - 0.23 * (CC) - 16.2 * ln(LOC) ……………………………………………(1)

- The formula derived by the Software Engineering Institute (SEI):

MI = 171 - 5.2 * log2(HV) - 0.23 * (CC) - 16.2 * log2(LOC) + 50 * sin(sqrt(2.4 * CM))...(2)

- The formula derived by Microsoft Visual Studio since 2008:

MI = MAX(0, (171 - 5.2 * ln(HV) - 0.23 * (CC) - 16.2 * ln(LOC)) * 100 / 171) ............. (3)

In equation (2), the Comment Ratio metric is used, which affects maintainability since comments help to clarify the purpose of the code and ease the maintenance process. The threshold value for the maintenance index is specified in Table 2.

**Table (2)** Threshold values for MI (M.A.M.Najm, 2014)

| **Maintainability Index** | **Assessment** |
|---|---|
| Less than 65 | Low |
| 65 to 85 | Medium |
| Greater than 85 | High |

In short, having a high level of maintainability in software systems is of utmost importance because it can lead to reduced maintenance costs, fewer errors, and improved overall quality. To achieve this, techniques such as code refactoring can be used. It is recommended to continuously improve the program's maintainability index and make necessary modifications accordingly.

## LINE OF CODE (LOC)

This measure is one of the simpler metrics, as it is easy to calculate and can be used to measure the complexity of a program(Sotonwa et al., 2023). The larger the number of lines of code, the more complex the program becomes. The metric for the number of lines of code can be divided into several types(Aswini & Yazhini, 2017; Sotonwa et al., 2023).

• The first type is Physical Lines of Code (LOC), which deals with all the source code lines of the software without any consideration for its content. This scale calculates only the software that will be delivered to the customer and is sometimes referred to as Non-Comment Lines of Code (NCLOC) or effective Lines of Code (eLOC), excluding blank spaces and comments(Parsa et al., 2023).

• The second type is Logical Line of Code (iLOC), which calculates the number of statements in the program. For example, in C and Java, the program statement ends with a semicolon, but this scale does not apply to all languages.

• The third type is Comment Lines of Code (CLOC), which calculates the number of comment lines in the source code. This scale does not consider the content or quality of these comments, whether they provide good information about the program or not. For example, in C and Java, single-line comments start with the symbol (//), and block comments start with the symbol (/*) and end with the symbol .(*/)

• Finally, the Comment Rate (CR) scale calculates the ratio of CLOC to LOC. This scale provides a percentage index of the amount of commenting in the program. As the ratio increases (to a certain extent), the understandability of the program increases. The CR can be calculated using the following formula: $CR = (CLOC / LOC) * 100$ ......................... (4)

## HALSTEAD METRICS

Halstead introduced a set of metrics through static analysis of programming code that can be calculated using a fixed formula. These metrics rely on extracting what is known as operators and operands [23]. Operators include all mathematical operations as well as special characters used in the language, such as brackets and parentheses, while operands include variables, constants, and character strings. Through a set of equations, difficulty and size can be calculated as well as estimation of effort and time(Tashtoush et al., 2023). These equations primarily depend on four basic variables(Ardito et al., 2020; Posnett et al., 2011):

- Unique Operator Count, denoted by n1
- Unique Operand Count, denoted by n2
- Total Operator Count, denoted by N1
- Total Operand Count, denoted by N2

Through these variables, Halstead derived the following equations (metrics)(Kencana et al., 2020; Madi et al., 2013; Navas-Su & Gonzalez-Torres, 2022):

1. Program length: represents the total length of the program:

$N = N1 + N2$ ............................. (5)

2. Program vocabulary: represents the total number of unique program statements (operands and operators) without repetition:

$n = n1 + n2$ ................................. (6)

3. Program volume: represents the minimum number of bits required to represent the program:

$V = (N1+N2)\log2(n1+n2)$ ........... (7)

4. Program difficulty: gives an indication of the difficulty of developing and understanding the program:

$D = [(n1)/2]\ (N2/n2)$ .................... (8)

5. Programming effort: is an indicator of the effort required to understand and develop the program:

$E = V * D$ .................................... (9)

6. Time to implement: this metric estimates the time required to represent the program in seconds and was derived through experiments and studies:

$T = E / 18$ ................................. (10)

7. Program level: is the inverse of error proneness, meaning that the lower the level, the more likely errors are to occur:

$L = 1 / D$ ................................. (11)

8. Estimated number of errors: is an estimation of the number of errors that might exist in the program module:

$B = [E ^ {(2/3)}] / 3000$ ............... (12)

## CYCLOMATIC COMPLEXITY(CC)

McCabe's Cyclomatic Complexity is a well-known and popular measure of structural complexity(Alraddadi, 2023)(Lenarduzzi et al., 2023). It is a measure of the number of control flows in a Control Flow Graph (CFG) of a programming module. The number of paths in the program is calculated through this graph. The higher the number of paths in the program, the higher the complexity. The Cyclomatic Complexity can be calculated by counting the number of decision points, represented by jump statements and loops (if else, switch cases, while, do while, for each, for loops) and adding one(Sarwar et al., 2013). The Cyclomatic Complexity can also be calculated by representing the program module using the basic structures of a flowchart, where it can be calculated in two ways (which give the same result), either by counting the nodes and edges of the CFG or by counting the number of binary decision points, which are the nodes where two branches emerge(Lavazza et al., 2023). The cyclomatic complexity can be calculated using one of two equations)Garg, 2013(:

$CC = e - n + 2$ .................................... (13)

where e represents the number of edges and n represents the number of nodes.

CC = NO. of Decisions + 1 ............... (14)

where NO. of Decisions represents the number of loops that have branching paths.

Generally, the cyclomatic complexity (CC) is an important measure of code complexity and can be useful in determining the maintainability of a program.

## CODE REFACTORING

Code Refactoring refers to the act of restructuring or reorganizing software code without altering its functionality. Its primary objective is to enhance the code's quality by making it more scalable, readable, and maintainable. There exist several refactoring strategies that can be employed to improve software maintainability)Kaur & Singh, 2017(. The practice is considered a fundamental step in software development, especially in agile methodologies. Refactoring eliminates code smells, reduces technical debt, and enhances the overall software architecture. It helps developers to identify issues in the codebase and enables them to address these problems systematically. The process of refactoring may involve changing the code's structure, design patterns, or class hierarchy while ensuring that the output remains the same. It is an iterative process that requires continuous testing and monitoring to ensure that the code meets the intended specifications. Refactoring is a crucial activity that helps to maintain software sustainability and improve the overall quality of the code (Fernandes et al., 2020; Jonsson, 2017; Wahler et al., 2017).

- Methods, classes, and variables can be used to improve code organization and readability.
- Replace complex expressions with simple, reusable components.
- Remove unnecessary or redundant code instructions.
- Rename variables, methods, and classes to improve clarity and consistency.
- Improve the use of data structures and algorithms to increase performance.

By applying these techniques, it is possible to improve the maintainability index of a software system, making it easier to understand, modify, and maintain.

## IMPLEMENTING THE PROPOSED MODEL (MMIBAR)

The purpose of measuring the maintainability index before and after restructuring or rebuilding the source code using the proposed MMIBAR model is to:

- Evaluate the impact of rebuilding on the maintainability of the software system.
- Identify areas in the code that are difficult to maintain and require improvement or further rebuilding.
- Evaluate the impact of different code restructuring techniques and strategies.
- Compare the cost and effort of rebuilding with the resulting improvements in maintainability.
- Provide future maintenance and development efforts.

And the maintenance index was calculated using three formulas: the original formula, the derivative formula according to the Software Engineering Institute, and the formula adopted by Microsoft Visual Studio, followed by comparing the results.

## ALGORITHM STEPS FOR (MMIBAR) MODEL

Step 1: In this step, all files (classes) of Python or Java type are read by specifying the path of the folder or directory. This is done by clicking on the button where the interface appears, and the folder is specified.

Step 2: In this step, files with the extension java or Py are searched for and the file is selected to calculate the metrics for each file (class) in the folder or directory.

Step 3: The number of lines for each class is calculated by the LOC_Counter function, and the information obtained is stored in a matrix of the Class_LOC class.

Step 4: The cyclomatic complexity CC of each class is calculated by analyzing the source code and extracting decision-making statements using equation (14), and the information is stored in a matrix of the Class_CC class.

Step 5: The ratio of the number of comment lines to the number of code lines is calculated according to equation (4)

Step 6: The number of operators and operands for each class is calculated by dividing the code into Tokens, comparing them using a set of functions in the Halstead class. The class size metric is calculated according to equation (7).

Step 7: The three equations (3)(2)(1) for the maintainability index are calculated for each class based on the metrics obtained from the previous steps.

Step 8: In this final step, the final metrics are displayed in tables.

The figure 1, Illustrates the steps of the proposed model (MMIBAR) which include inputs, processing, and outputs.

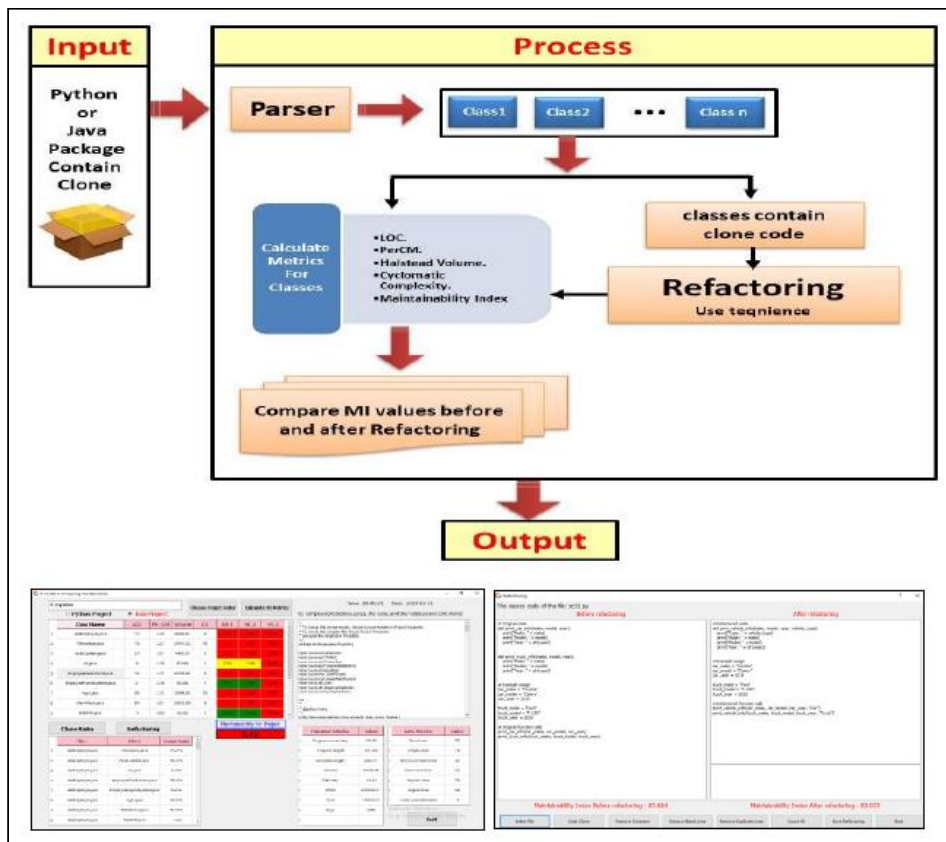



**Figure (1) Shows the operation of the proposed model**

The flowchart in Figure 2, Illustrates the methodology for measuring the maintainability index (MI) of a software system before and after code refactoring.

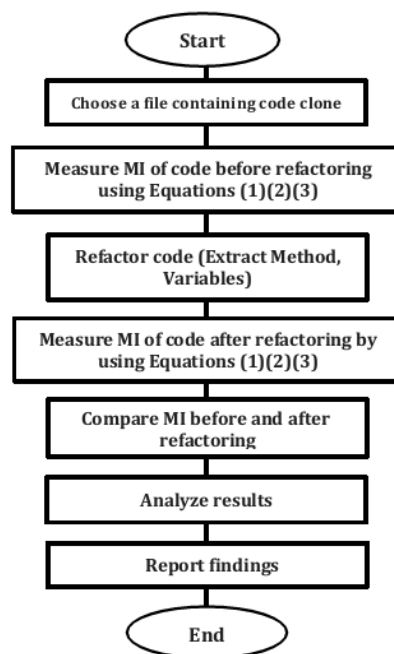



**Figure (2) Flowchart for the operation of the proposed model.**

**CASE STUDY**

The designed model was tested on a Payroll Management System, a program created in Java language that aims to help the company efficiently manage its employees' salaries. Using the system can significantly improve the company's work by monitoring its employees' performance from time to time and calculating salaries automatically without using paper-based work. It provides the company with the ability to manage information for all employees and add, update, delete, create, print reports, and more. The program consists of 13 Java file types. The test results of the model showed that there is a difference between the maintainability index value before and after code refactoring and code clone removal, which is an increase in the MI value. This leads to improving the quality and maintenance of the program.

Table (3) Shows the results of the measurements for three files

| **File Name** | **LOC** | **CR** | **HV** | **CC** |
|---|---|---|---|---|
| EmployeePayrollSystem.java | 21 | 4.76 | 292.842 | 1 |
| db.java | 36 | 2.78 | 885.804 | 1 |
| Audit_details.java | 279 | 2.87 | 11400.381 | 3 |

Table 3, showing different metrics for three different files in a case study (Payroll Management System). include Lines of Code (LOC), Comment Rate (CR), Halstead Volume (HV), and Cyclomatic Complexity (CC).

- LOC: It measures the number of lines of code in a file.
- CR: It measures the proportion of lines of comments to the total number of lines of code.
- HV: It measures the complexity of a program by analyzing the operators and operands used in it.
- CC: It measures the number of independent paths through a program's source code.

For example, EmployeePayrollSystem.java has 21 lines of code, a comment rate of 4.76, an HV of 292.842, and a CC of 1. Similarly, db.java has 36 lines of code, a comment rate of 2.78, an HV of 885.804, and a CC of 1. Finally, Audit_details.java has 279 lines of code, a comment rate of 2.87, an HV of 11400.381, and a CC of 3.

These metrics are useful in identifying areas of the code that may be more prone to errors or more difficult to maintain. For example, a high HV and CC could indicate that a file is more complex and may require more effort to modify or debug in the future. On the other hand, a high CR indicates that the code is well-documented and easier to understand, which could lead to fewer errors and faster maintenance.

Table (4) Shows the results for three files.

| **File Name** | **MI(Before Refactoring)** | **MI(After Refactoring)** | **Maintainability** |
|---|---|---|---|
| EmployeePayrollSystem.java | 91.91 | 100 | High to maintain |
| db.java | 77.43 | 83.780 | Moderate to maintain |
| Audit_details.java | 30.51 | 40.348 | Difficult to maintain |

Table 4, summarizes the measurement of maintainability index for three files and the effect of rebuilding on their maintainability. Through analysis, developers can identify areas of improvement in the code and take steps to make it more maintainable.

## CONCLUSION

The conclusion of the research on measuring the maintainability index before and after code restructuring should summarize the main findings and discuss their impact on software development. Some potential points to consider may include:

- The study results show that code restructuring can significantly improve the maintainability index of software systems. This indicates that restructuring is an important technique for improving code quality and maintenance in the long term.
- The specific restructuring techniques used in the study, such as variable extraction or function renaming, had different effects on the maintainability index. This suggests that some techniques may be more effective in improving maintainability in certain situations.
- The results obtained have positive effects on the development process, as the restructuring processes reduce cost and provide appropriate time for development.
- Further research should be conducted to find a deeper or more extensive relationship between code restructuring and maintainability, such as the effects of different restructuring techniques on various types of software systems.

**REFERENCES**

Alraddadi, S. (2023). Detecting Security Bad Smells of Software by using Data Mining.

Ardito, L., Coppola, R., Barbato, L., & Verga, D. (2020). A tool-based perspective on software code maintainability metrics: a systematic literature review. Scientific Programming, 2020, 1–26.

Aswini, S., & Yazhini, M. (2017). An assessment framework of routing complexities using LOC metrics. 2017 Innovations in Power and Advanced Computing Technologies, i-PACT 2017, 2017-Janua, 1–6. https://doi.org/10.1109/IPACT.2017.8245022

Caro, A., Calero, C., Mendes, E., & Piattini, M. (2007). A probabilistic approach to web portal's data quality evaluation. QUATIC 2007 - 6th International Conference on the Quality of Information and Communications Technology, March 2015, 143–153. https://doi.org/10.1109/QUATIC.2007.8

Chaparro, O., Bavota, G., Marcus, A., & Di Penta, M. (2014). On the impact of refactoring operations on code quality metrics. 2014 IEEE International Conference on Software Maintenance and Evolution, 456–460.

Di Biase, M., Rastogi, A., Bruntink, M., & Van Deursen, A. (2019). The delta maintainability model: Measuring maintainability of fine-grained code changes. Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019, 113–122. https://doi.org/10.1109/TechDebt.2019.00030

Dig, D. (2011). A refactoring approach to parallelism. IEEE Software, 28(1), 17–22. https://doi.org/10.1109/MS.2011.1

Draz, Farhan, M. S., Eldefrawi, M. M., Draz, A. M. E. S. M., Farhan, M. S., & Eldefrawi, M. M. (2021). A Survey of Refactoring Impact on Code Quality. FCI-H Informatics Bulletin, 3(1), 16–22. https://doi.org/10.21608/FCIHIB.2021.54539.1007

Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., & Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? Information and Software Technology, 126. https://doi.org/10.1016/j.infsof.2020.106347

Fernandes, S., Aguiar, A., & Restivo, A. (2022). LiveRef: a Tool for Live Refactoring Java Code. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 1–4.

Fernandes, S., Aguiar, A., & Restivo, A. (2023). Empirical Evaluation of a Live Environment for Extract Method Refactoring. ArXiv Preprint ArXiv:2307.11010.

Fujiwara, K., Fushida, K., Yoshida, N., & Iida, H. (2013). Assessing refactoring instances and the maintainability benefits of them from version archives. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence

and Lecture Notes in Bioinformatics), 7983 LNCS, 313–323. https://doi.org/10.1007/978-3-642-39259-7_25

Garg, M. (2013). Analysing the quality attributes of AOP using CYVIS tool. International Journal of Computers & Technology, 4, c2.

Gradišnik, M., Beranič, T., & Karakatič, S. (2020). Impact of historical software metric changes in predicting future maintainability trends in open-source software development. Applied Sciences, 10(13), 4624.

Han, A. R., Bae, D. H., & Cha, S. (2015). An efficient approach to identify multiple and independent Move Method refactoring candidates. Information and Software Technology, 59, 53–66. https://doi.org/10.1016/j.infsof.2014.10.007

Hegedűs, G., Hrabovszki, G., Hegedűs, D., & Siket, I. (2010). Effect of object oriented refactorings on testability, error proneness and other maintainability attributes. 1–7. https://doi.org/10.1145/1890692.1890700

Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability - A preliminary report. QUATIC 2007 - 6th International Conference on the Quality of Information and Communications Technology, 30–39. https://doi.org/10.1109/QUATIC.2007.7

Heričko, T., & Šumak, B. (2023). Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems. Applied Sciences, 13(5), 2972.

Hong-liang, C. (2020). The Verification Method of Maintainability Indexes of Equipment Based on UML. Journal of Physics: Conference Series, 1678(1), 12068.

Hu, Y., Jiang, H., & Hu, Z. (2023). Measuring code maintainability with deep neural networks. Frontiers of Computer Science, 17(6), 176214.

Jonsson, A. (2017). The Impact of Refactoring Legacy Systems on Code Quality Metrics. http://www.diva-portal.se/smash/get/diva2:1114582/FULLTEXT01.pdf

Kaur, G., & Singh, B. (2017). Improving the quality of software by refactoring. Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems, ICICCS 2017, 2018-Janua, 185–191. https://doi.org/10.1109/ICCONS.2017.8250707

Kencana, G. H., Saleh, A., Darwito, H. A., Rachmadi, R. R., & Sari, E. M. (2020). Comparison of maintainability index measurement from Microsoft Codelens and line of code. 2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI), 235–239.

Lavazza, L., Abualkishik, A. Z., Liu, G., & Morasca, S. (2023). An empirical evaluation of the “Cognitive Complexity” measure as a predictor of code understandability. Journal of Systems and Software, 197, 111561.

Lenarduzzi, V., Kilamo, T., & Janes, A. (2023). Does Cyclomatic or Cognitive Complexity Better Represents Code Understandability? An Empirical Investigation on the Developers Perception. ArXiv Preprint ArXiv:2303.07722.

M.A.M.Najm, N. (2014). Measuring Maintainability Index of a Software Depending on Line of Code Only. IOSR Journal of Computer Engineering, 16(2), 64–69. https://doi.org/10.9790/0661-16276469

Macharia, E. M., & Kimwele, M. (2022). A Metrics-Based Maintainability Estimation Framework for Object Oriented Software in Design Phase. 1–12.

Madi, A., Zein, O. K., & Kadry, S. (2013). On the improvement of cyclomatic complexity metric. International Journal of Software Engineering and Its Applications, 7, 67–82.

Malhotra, R., & Chug, A. (2016). An empirical study to assess the effects of refactoring on software maintainability. 2016 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2016, 110–117. https://doi.org/10.1109/ICACCI.2016.7732033

Meananeatra, P. (2012). Identifying refactoring sequences for improving software maintainability. 2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings, 406–409. https://doi.org/10.1145/2351676.2351760

Mohan, M., & Greer, D. (2017). MultiRefactor: Automated refactoring to improve software quality. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 10611 LNCS, 556–572. https://doi.org/10.1007/978-3-319-69926-4_46

Mohan, M., & Greer, D. (2019). Using a many-objective approach to investigate automated refactoring. Information and Software Technology, 112, 83–101. https://doi.org/10.1016/j.infsof.2019.04.009

Molnar, A.-J., & Motogna, S. (2020). Longitudinal Evaluation of Open-Source Software Maintainability. ArXiv Preprint ArXiv:2003.00447.

Morales, R., Khomh, F., & Antoniol, G. (2020). RePOR: Mimicking humans on refactoring tasks. Are we there yet? Empirical Software Engineering, 25(4), 2960–2996. https://doi.org/10.1007/s10664-020-09826-7

Navas-Su, J., & Gonzalez-Torres, A. (2022). An approach for the forecasting of the maintainability of system functionalities. Proceedings of the 2022 European Symposium on Software Engineering, 33–42.

Ogheneovo, E. E. (2014). On the Relationship between Software Complexity and Maintenance Costs. Journal of Computer and Communications, 02(14), 1–16. https://doi.org/10.4236/jcc.2014.214001

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., & Deb, K. (2016). Multi-criteria code refactoring using search-based software engineering: An industrial case study. ACM Transactions on Software Engineering and Methodology, 25(3). https://doi.org/10.1145/2932631

Parsa, S., Zakeri-Nasrabadi, M., Ekhtiarzadeh, M., & Ramezani, M. (2023). Method name recommendation based on source code metrics. Journal of Computer Languages, 74, 101177.

Posnett, D., Hindle, A., & Devanbu, P. (2011). A simpler model of software readability. Proceedings of the 8th Working Conference on Mining Software Repositories, 73–82. https://doi.org/10.1145/1985441.1985454

Sarwar, M. M. S., Shahzad, S., & Ahmad, I. (2013). Cyclomatic complexity: The nesting problem. Eighth International Conference on Digital Information Management (ICDIM 2013), 274–279.

Sotonwa, K., Adeyiga, J., Adenibuyan, M., & Dosunmu, M. (2023). Survey of Schema Languages: On a Software Complexity Metric. Advances in Information and Communication: Proceedings of the 2023 Future of Information and Communication Conference (FICC), Volume 2, 349–361.

Tairas, R. (2006). Clone detection and refactoring. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, 2006, 780–781. https://doi.org/10.1145/1176617.1176722

Tashtoush, Y., Abu-El-Rub, N., Darwish, O., Al-Eidi, S., Darweesh, D., & Karajeh, O. (2023). A Notional Understanding of the Relationship between Code Readability and Software Complexity. Information, 14(2), 81.

Wahler, M., Drofenik, U., & Snipes, W. (2017). Improving code maintainability: A case study on the impact of refactoring. Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, 493–501. https://doi.org/10.1109/ICSME.2016.52