International Journal of Informatics and Computation (IJICOM)
Vol. 7, No. 2, 2025
ISSN: 2685-8711, E-ISSN: 2714-5263 DOI: 10.35842/ijicom

Design-Level Refactoring Using Genetic Algorithms in an Academic System

Bayu Priyambadha[1], Nurudin Santoso[2]

Abstract

Legacy software systems undergo structural degradation due to continuous evolution, which makes maintenance increasingly complex. This study investigates the effectiveness of design-level refactoring using a genetic algorithm (GA)-based class decomposition method in a legacy academic information system of FILKOM. We utilize static analysis via CodeMR to detect code smells and evaluate maintainability using the Maintainability Index (MI). Our findings reveal a significant increase in median MI from 28.25 to 70.90 post-refactoring. While formal statistical significance was not reached ($p = 0.10$), the effect size was consistently strong ($r = 1.00$), confirming the positive impact of the proposed approach. The study contributes a practical, replicable method for improving maintainability in real-world systems, supported by semantic similarity and usability-based decomposition. This work highlights the potential of design-level optimization using evolutionary algorithms and calls for further multi-domain validation.

Keywords:
Software Design, Academic System, Genetic Algorithm, Maintainability Index

1. Introduction

Software systems are in a constant state of evolution, driven by various environmental changes such as hardware upgrades, operating system updates, shifting user needs, and new stakeholder requirements [1], [2]. While this evolution is essential to keep the software relevant and valuable, it also introduces significant challenges. Without careful management, continuous changes can lead to the gradual degradation of the software's internal structure, making it increasingly difficult to maintain, extend, or adapt over time. According to Lehman's law, the complexity of software systems tends to increase as they evolve, regardless of efforts to control it [1], [2]. This rising complexity can result in reduced system quality, higher maintenance costs, and greater risks of introducing defects during updates. To counteract these issues, it is essential to adopt a structured, systematic, and metric-driven maintenance strategy. Such an approach helps monitor, evaluate, and preserve the system's quality as it evolves, ensuring long-term sustainability.

The internal structure of software plays an important role in maintaining maintainability, readability, and flexibility of the system to changes [3], [4], [5], [6], [7]. A well-maintained internal structure allows developers to easily understand the system's design, make modifications safely, and extend its functionality without introducing unintended side effects. However, when this structure is neglected or compromised due to the accumulation of uncontrolled or poorly managed changes, the software begins to experience a gradual decline in quality, a phenomenon commonly referred to as structural degradation. This

Corresponding Author: Bayu Priyambadha, Faculty of Computer Science, Universitas Brawijaya, Indonesia (bayu_priyambadha@ub.ac.id)
1 Bayu Priyambadha, Faculty of Computer Science, Universitas Brawijaya, Indonesia (bayu_priyambadha@ub.ac.id)
2 Nurudin Santoso, Faculty of Computer Science, Universitas Brawijaya, Indonesia (nurudin.santoso@ub.ac.id)

degradation has several significant and tangible consequences. One major impact is the increasing difficulty developers face when trying to comprehend the codebase, making tasks such as bug fixing or feature enhancement more time-consuming and error-prone. Additionally, the likelihood of introducing new defects or breaking existing functionalities rises as the system becomes more fragile. Over time, this leads to higher maintenance costs, longer development cycles, and greater resource demands for even minor updates or improvements. Ultimately, without deliberate efforts to preserve or restore the internal structure, the software risks becoming unmanageable, reducing its lifespan, and negatively affecting the organization's ability to respond to new business or technical needs efficiently.

In the context of software engineering, the quality of internal structure can be measured objectively using software metrics, such as Lines of Code (LOC), Cyclomatic Complexity (CC), Halstead Volume (HV), and Maintainability Index (MI) [8]. These metrics help developers identify parts of a system that are experiencing quality degradation, or what is often referred to as a code smell or smell. Smell is an early indication visible on the surface of the code that indicates the potential for deeper problems in the structure or design of the software [9]. Although not always an error or bug that directly impacts the functionality of the system, the presence of a smell indicates that a section of the code is likely suboptimal, challenging to understand, or will make future maintenance and development difficult. In addition, the presence of smells such as Blob Class, Feature Envy, and other types is also an important indicator of potential design problems that must be addressed.

One approach to improving the quality of internal structure is through refactoring, which is a systematic process of modifying the internal structure of software without changing its external behavior [9]. Refactoring is not only helpful in improving code readability but also helps reduce technical debt, increase modularity, and support continuous development.

In its development, smells can no longer be recognized at the source code level, but can also be recognized at the design level [10], [11], [12], [13], [14], [15]. Refactoring techniques that focus on the design level, especially class decomposition in class diagrams, aim to break down large and complex classes into several smaller, more specific, and cohesive classes [3], [16], [17], [18]. This decomposition has a direct impact on increasing cohesion and reducing coupling, which are key indicators of a healthy software structure.

This study focuses on a real case, namely the FILKOM Apps system. This academic management information system has been in use for more than a decade at the Faculty of Computer Science, Brawijaya University. Over time, this application has undergone numerous changes to accommodate the evolving dynamics of academic policies, internal business processes, and technological advancements. However, the incremental and sporadic changes that have occurred over the years have caused some parts of the system, especially in crucial modules such as the thesis module, to become increasingly complex.

Considering the importance of maintaining the quality of the system developed over time, this study aims to evaluate the quality of the internal structure of the FILKOM Apps system and to make improvements through a class decomposition-based refactoring approach. The evaluation was conducted using static analysis tools, such as CodeMR, to detect code smells and measure quality metrics. Meanwhile, the refactoring process was carried out using the Genetic Algorithm-based Class Decomposition approach [18], an artificial intelligence-based technique developed in previous studies that aims to optimize the division of responsibilities between classes automatically.

Unlike previous studies that have primarily focused on experimental case studies, such as ArgoUML, this study tests the GA approach on a real-world academic information system that has been operational for over a decade. This study aims not only to

demonstrate the effectiveness of refactoring in improving the MI value of problematic parts of the system but also to contribute theoretically by proving that model-based design-level refactoring can be applied systematically to handle software structure degradation in real-world contexts. The findings of this study are expected to provide a practical foundation for development teams to manage legacy systems in a more structured manner, as well as extend the life of software more efficiently and measurably.

2. Related Works

Early works on software engineering have long emphasized the importance of maintainable and evolvable system design. Sommerville [1] and Pressman [2] established foundational principles for structuring modular, cohesive, and low-coupled software systems, setting the theoretical basis for modern refactoring and maintainability research. These seminal works introduced the conceptual framework for understanding software quality attributes such as maintainability, scalability, and reliability through systematic design methodologies. However, while these classical models provide clear process-oriented perspectives, they lack automated techniques for improving design-level structures dynamically, leaving a research gap for approaches leveraging optimization and artificial intelligence to enhance software maintainability.

Recent studies have advanced these foundations by empirically evaluating the maintainability of design-level structures. Priyambadha and Katayama [3] examined the impact of class decomposition on maintainability, demonstrating that decomposing complex classes can significantly improve modularity and reduce code smells. Similarly, Palomba et al. [4] analyzed the influence of code smells on maintainability in large-scale software systems, showing that design anomalies such as “God classes” and “Feature Envy” are reliable indicators of low maintainability. Szőke et al. [5] reinforced these findings in an industrial context, where large-scale refactoring efforts measurably improved system maintainability metrics. Nonetheless, these works focus mainly on empirical measurement rather than algorithmic optimization, indicating a need for automated methods that can refactor designs while preserving behavioral consistency.

Efforts to classify and quantify maintainability have also been extensive. Saraiva et al. [6] proposed a family of metrics catalogs for evaluating object-oriented maintainability, offering a taxonomy of metric-based indicators for cohesion, coupling, and complexity. Yamashita and Counsell [7] further explored system-level indicators by identifying how code smells correlate with maintainability degradation. Earlier work by Coleman et al. [8] already emphasized the utility of metrics-based evaluation for predicting software longevity. However, while these studies contributed to the measurement dimension of maintainability, they did not propose automated corrective mechanisms, leaving the practical problem of refactoring largely manual and heuristic-driven.

The concept of refactoring itself was systematized by Fowler et al. [9], who defined structured procedures to improve existing code design without altering external behavior. Subsequent works extended refactoring beyond source code to non-source and model-level artifacts. Rochimah et al. [10] and Misbhauddin and Alshayeb [11] provided comprehensive literature reviews on non-source code refactoring, particularly focusing on UML model refactoring. These studies underscored the growing relevance of refactoring at the design level before implementation, but they did not address how to algorithmically optimize the decomposition process for design diagrams. This omission motivates the exploration of heuristic optimization methods, such as genetic algorithms, to automate design-level restructuring.

The detection of design anomalies in UML class diagrams has become a focal point for improving software quality at the model level. Priyambadha et al. [12], [13] introduced an approach for detecting “Blob” and “Feature Envy” smells using class feature analysis, highlighting how semantic and structural indicators can reveal potential refactoring targets.

Saranya et al. [14] proposed an EGAPSO-based model for detecting code smells, utilizing evolutionary and similarity-based measures to enhance accuracy. Similarly, Rattan et al. [15] explored model clone detection using tree comparison techniques to identify redundancy within UML diagrams. These studies validate the utility of applying computational intelligence to model-level analysis, but they primarily focus on smell detection rather than automated design improvement.

Several studies have contributed directly to class decomposition and refactoring optimization. Priyambadha and Katayama [16], [17] emphasized semantic approaches for assessing class cohesion and explored how design-level decomposition affects maintainability. Their findings demonstrated that decomposition guided by semantic cohesion can substantially improve maintainability metrics. Subsequent works by the same authors [18] applied genetic algorithms to automate design-level class decomposition, presenting a novel framework that adapts evolutionary computation to optimize class structures. Hamdi et al. [19] and Priyambadha and Katayama [20], [21] expanded on this by proposing threshold-driven and hierarchical clustering methods for decomposition, demonstrating measurable maintainability improvements. Nonetheless, these heuristic-based approaches sometimes face challenges in balancing performance optimization and structural interpretability.

Beyond decomposition and refactoring, similarity-based measures play an important role in evaluating design-level transformations. Wu and Palmer [22] proposed one of the earliest semantic similarity metrics, which later informed design similarity assessments in software models. Dijkman et al. [23] extended this concept by introducing quantitative metrics for evaluating the similarity of business process models, providing a mathematical foundation for comparing refactored and original designs. While these models provide critical evaluation tools, they were not explicitly integrated into evolutionary optimization frameworks, presenting an opportunity to merge similarity evaluation with genetic algorithm-driven refactoring for more adaptive and automated design improvement.

In summary, the related works collectively reveal significant progress in software maintainability evaluation, refactoring techniques, and model-level analysis. Traditional metrics-based and manual refactoring approaches [1]–[9] have evolved into intelligent systems employing heuristics, clustering, and evolutionary computation [14]–[21]. However, a major gap persists in integrating design-level decomposition with adaptive optimization algorithms that dynamically balance cohesion and coupling. This research addresses that gap by proposing a Genetic Algorithm-based design-level refactoring method, specifically tailored for academic systems, where modular maintainability and class-level evolution are essential for long-term system sustainability and adaptability.

3. Proposed Method

In the GA-based approach, the class decomposition process is modeled as an optimization problem. Class elements are represented as genes in a chromosome, while the decomposed clusters are positioned as individuals in the population. An evaluation function (fitness function) is designed to measure the quality of the decomposed clusters by considering two leading indicators, including Silhouette Coefficient ($s(i)$), which represents the compactness and separation between elements in a cluster. Class Usability, which refers to the functionality of a cluster as a class, as indicated by the existence of at least one public method.

A simple explanation of class usability is that a class in a software design will be projected into an object when the software is running. The objects formed will collaborate to fulfill a specific functionality. If an object only has a private class element, then it is assumed that the object cannot collaborate with other objects. Therefore, class usability can be

understood as follows [21].

$$CUsability = \begin{cases} 0 & , mpub = 0 \\ 1 & , mpub \geq 1 \end{cases} \quad (1)$$

Where $mpub$ is the number of public methods in a class. By considering the class usability, the evaluation function is formulated as follows. Furthermore, this evaluation function will also serve as the fitness function in executing the genetic algorithm, ensuring that candidate solutions are quantitatively assessed and optimized based on their contribution to class usability.

$$Eval = a.s(i) + b.CUsability \quad (2)$$

which a dan b as the weights of each component.

The evolution process in GA involves several stages: population initialization, linear ranking selection, single-point crossover, and mutation using the swap mutation approach. This process is repeated iteratively until it meets the termination condition, namely the stagnation of fitness values for several generations.

A. Linear Ranking Selection

Instead of selecting individuals purely by fitness proportion, linear ranking assigns probabilities based on the rank of each individual within the population. The best individuals receive higher selection probabilities, but even lower-ranked individuals still retain a chance of being selected. This reduces the risk of premature convergence and maintains population diversity. The selection probability for individual i with rank $r(i)$ is:

$$P_{sel}(i) = \frac{r(i)}{\sum_{j=1}^{n} r(j)} \quad (3)$$

It is to ensure a smoother selection pressure compared to traditional fitness-proportionate selection.

B. Single-Point Crossover

Crossover is the primary mechanism for combining genetic material from two parent solutions. In the single-point crossover strategy, a random cut point k is chosen within the chromosome. The offspring are generated by exchanging the segments of the two parents at that point:

$$o_1 = [p_1\,[1{:}k], p_2\,[k+1{:}N]], o_2 = [p_2\,[1{:}k], p_1\,[k+1{:}N]] \quad (4)$$

This operator introduces new combinations of class assignments and promotes exploration of the solution space.

C. Swap Mutation

Mutation is applied to maintain diversity and avoid local optima. In the swap mutation strategy, two random genes in the chromosome are selected, and their positions are exchanged:

$$o' = swap(o, i, j) \quad (5)$$

This operation introduces small perturbations in the decomposition structure, allowing the search to escape from stagnation and discover better class partitioning.

D. Elitism

To prevent the loss of high-quality solutions across generations, elitism is applied. The best EEE individuals, based on their fitness values, are directly copied into the next

generation without modification. This guarantees that the evolutionary process never degrades the best solution found so far.

In this study, we calculate relation capacity as the maximum number of possible relationships that can exist within a class, serving as an upper bound for assessing its internal connectivity. In this context, a class is conceptualized as a relational space or area that holds a finite capacity for relationships between its elements, specifically between methods and attributes. Understanding the relation capacity is essential for evaluating the class's structural limits and identifying potential design issues. The maximum number of possible relations is calculated using a specific mathematical equation, which provides a baseline for comparing the actual number of observed relations within the class. The equation is described as follows:

$$MaxRelation = \frac{(m + a)((m + a) - 1)}{2} \quad (6)$$

where m is the number of methods and a is the number of attributes.

We also measure Cohesion Value by combining the values of MAR (method-attribute relations), MMR (inter-method relations), and AAR (inter-attribute relations). These relations include both direct connections and transitive (indirect) connections within a class, offering a comprehensive view of how tightly its elements are linked. To calculate cohesion, the total number of actual relations between methods and attributes is divided by the maximum possible number of relations that can exist within the class (3). The resulting value provides an indicator of the class's internal unity, as defined by the following equation:

$$Cohesion = \frac{MAR + MMR + AAR}{MaxRelation} \quad (7)$$

4. Experimental Setup

In this section, we outline a structured sequence of activities, combining source code analysis, class diagram generation, metric evaluation, and automated refactoring. The scenario begins with the collection of PHP source code, sourced from official repositories or trusted documentation. This code is then converted into a class diagram in XML format, enabling a structural view of the system's design. The resulting class diagram serves as the input for calculating the initial Maintainability Index (MI), which reflects the current maintainability state of the software.

To enhance this maintainability, a genetic algorithm-based refactoring is applied to the class diagram. This evolutionary approach explores alternative class structures to improve design quality. The refactored structure is then used to modify the source code, aligning the implementation with the optimized design. Subsequently, the Maintainability Index is measured again using the modified code, producing a new value that represents the post-refactoring maintainability level. Finally, the research scenario concludes with a comparison between the initial and final maintainability indexes, providing quantitative evidence of the effectiveness of the applied refactoring strategy. Fig 1 illustrates the research scenario or experimental setup.

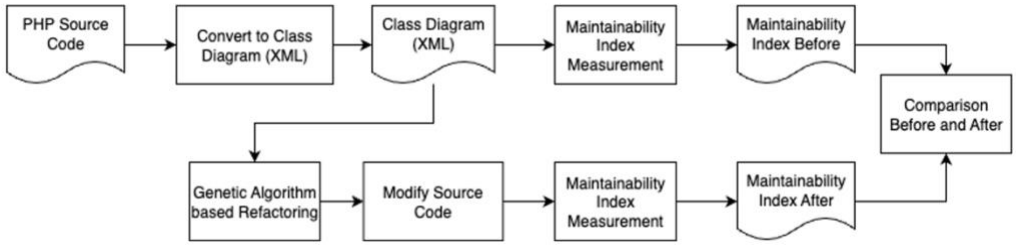

Fig 1 Research scenario of the study

Fig.1 illustrates the workflow of the Genetic Algorithm-based Refactoring process for improving software maintainability. It begins with the PHP source code, which is first converted into a Class Diagram in XML format to represent the system's structural design. This class diagram serves as the foundation for analyzing the software architecture and computing the initial maintainability index, referred to as the Maintainability Index Before Refactoring. This stage establishes a baseline measure of how maintainable the current software is before any optimization is applied.

The next stage is the class diagram will be fed into the Genetic Algorithm-based Refactoring module, which performs automated refactoring operations aimed at enhancing the design quality. The refactored class diagram is then used to modify the source code accordingly, ensuring consistency between the model and the implementation. A second maintainability index measurement is conducted to evaluate the Maintainability Index After Refactoring. Finally, a comparison between the before and after measurements is made to assess the effectiveness of the genetic algorithm in improving software maintainability. This end-to-end process integrates model-based analysis with intelligent refactoring, allowing for quantitative validation of design improvement.

5. Results and Analysis

The source code analyzed in this study is part of the thesis module in the FILKOM Apps application, comprising 36 classes. FILKOM Apps is a web-based application developed using the PHP programming language, one of the most commonly used programming languages for developing dynamic web applications. The source code was collected from a version management system managed by PSIK FILKOM, which is fully responsible for the management, maintenance, and development of the application. Since it was first implemented more than ten years ago, this application has undergone various significant changes. These changes include the addition of new features to support the academic process, fixing bugs that appear over time, and adjusting to the latest technology infrastructure.

The PHP source code used in the FILKOM Apps application often requires adaptation due to environmental changes, such as server operating system updates, PHP version updates, or integration with other web technologies, including MySQL databases and web frameworks. With a class-based modular architecture, each application module, including the thesis module, is managed within a class structure that encompasses various essential functionalities, such as student data management, title submission processes, and thesis evaluation. However, as the application grows and evolves, the complexity of the code also increases, especially in classes that are frequently updated, raising concerns about the decline in the software's internal structure quality. Therefore, it is essential to evaluate and improve this source code through a refactoring approach to maintain its quality and ease of maintenance in the future.

A. Smell Detection and Maintainability Index Measurement

The process of smell identification in a software system is a crucial step to ensure that code quality is maintained and to facilitate future maintenance. In this approach, detection is not only done through direct analysis of the source code but starts from class information that has been previously extracted from the class diagram. This information includes essential elements such as attributes, operations (methods), relationships between classes, and the complexity of relationships within the system. Table 1 describes the results of smell identification (Blob) based on class information.

Table 1. List of Smell Class

No.	Class Name	File Name	LOC
1.	skripsi_detail	detail.php	11620
2.	skripsi_kemajuan	kemajuan.php	1564
3.	skripsi_penjadwalan	penjadwalan.php	604
4.	skripsi_progres	progres.php	1612
5.	skripsi_report	report.php	7923
6.	SkripsiSebaran	sebaran.php	440
7.	SkripsiSetting	setting.php	795
8.	model_master	master.php	1505
9.	ModelPengajuan	pengajuan.php	285
10.	ModelPenjadwalan	penjadwalan.php	244
11.	ModelReport	report.php	4136

In addition, to strengthen the analysis process, MI calculations are also performed on each class. MI is a composite metric that combines various aspects of code quality. The results of this MI calculation are then combined with data obtained from the class diagram, allowing for a more accurate identification of classes that have the potential to become problematic. Table 2 shows the calculation of MI values using the tool.

Table 2. MI Measurement Result

No.	Class Name	MI
1.	skripsi_kemajuan	18,49
2.	skripsi_penjadwalan	28,25
3.	SkripsiSebaran	33,98
4.	SkripsiSetting	25,18
5.	ModelReport	24,29
6.	ModelPengajuan	70,9
7.	ModelPenjadwalan	49,81

Table 2 shows a list of classes after going through the MI calculation process using the calculation tool. Based on the results of the MI calculation above, seven classes were identified that were included in the Difficult to Maintain and Moderately Maintainable categories. This finding shows that not all classes that are indicated to have smells can be directly categorized as problematic classes according to MI. Therefore, the refactoring process is focused on seven classes that show low to moderate maintainability levels. These classes include skripsi_kemajuan, skripsi_penjadwalan, skripsi_report, SkripsiSebaran, SkripsiSetting, ModelReport, ModelPengujian, ModelPanjadwaalan, and model_master.

B. Refactoring

The refactoring process in this study is specifically focused on the decomposition of classes that are identified as problematic, particularly those falling into the Blob or God Class category. To achieve this, the study employs a Genetic Algorithm-based Clustering approach, which systematically decomposes large, overloaded classes into several smaller classes, each assigned with more focused and specific responsibilities. By breaking down these Blob classes, the refactoring process aims to improve the clarity, cohesion, and modularity of the software, ultimately enhancing the quality of the code and making it easier to maintain, test, and extend in future development cycles. Following the refactoring, the study conducts Maintainability Index (MI) measurements to assess the impact of the refactoring activities on the maintainability level of each class. Table 3 shows a list of classes after refactoring and the MI measurement result

.

Table 3. List of Classes After Refactoring and MI Measurement Result

No.	Class	New Class	MI
1.	skripsi_kemajuan	skripsi_kemajuan	117,34
		skripsi_kemajuan1	18,22
2.	skripsi_penjadwalan	skripsi_penjadwalan	27,47
		skripsi_penjadwalan1	117,34
3.	SkripsiSebaran	skripsi_sebaran	25,18
		skripsi_sebaran1	117,34
4.	SkripsiSetting	skripsi_setting	25,18
5.	ModelReport	model_report	24,21
		model_report1	135,09
6.	ModelPengajuan	model_pengajuan	70,9
7.	ModelPenjadwalan	model_penjadwalan	49,81

The MI values obtained after refactoring are averaged across the refactored classes to produce an aggregated MI score. This averaged post-refactoring MI is then compared directly with the pre-refactoring MI values to determine the overall improvement achieved through the refactoring efforts. Table 4 explains the results of the comparison, detailing the MI values before and after refactoring for each class.

Table 4. MI Value Before and After Refactoring

No.	Class	MI Before	MI After
1.	skripsi_kemajuan	18,49	67,78
2.	skripsi_penjadwalan	28,25	72.4
3.	SkripsiSebaran	33,98	71.26
4.	SkripsiSetting	25,18	25,18
5.	ModelReport	24,29	79.65
6.	ModelPengajuan	70,9	70,9
7.	ModelPenjadwalan	49,81	49,81

C. Result Analysis

One technique used to improve maintainability is class decomposition, which is breaking down an overly complex class into several smaller and more specific classes. This technique aims to reduce complexity, enhance readability, and facilitate easier code maintenance. In this presentation, we will analyze the impact of class decomposition on the MI value, a metric for measuring code maintainability. Based on the results of the data visualization, we will see the changes in the MI value before and after the decomposition process and evaluate the significance of these changes, both in terms of statistics and their practical implications for software maintainability. Fig.2 shows a comparison of the MI value

before and after class decomposition on the source code.

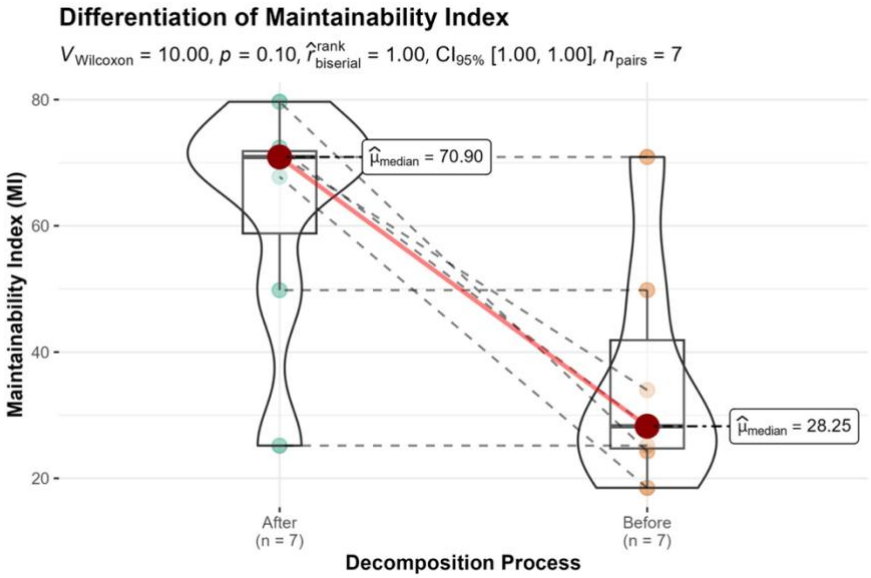

Fig.2 Comparison of the MI value before and after class decomposition

Fig. 2 displays the distribution of MI values before decomposition, which tends to narrow in the low range. In contrast, after decomposition, the distribution widens towards high values, with a median that increases almost threefold. The red line connecting the two medians shows an upward trend, while the dashed lines connecting the individual points show that each entity experienced an improvement in MI values. These findings support the main hypothesis of the study, namely that the application of class decomposition, preceded by a smell detection and refactoring process, can improve the internal quality of the code, especially in terms of maintainability.

The MI value is one of the important indicators in measuring software quality, especially in terms of maintainability or ease of maintenance. The plot used is a combination of a violin plot and a box plot, which allows visualization of the distribution of values, medians, spreads, and individual data spreads. On the “Before” side (before decomposition), the median MI was recorded at 28.25, indicating that maintainability was at a low or poor level. Conversely, on the “After” side (after decomposition), the median MI jumped sharply to 70.90, indicating a significant improvement in code maintainability after the class decomposition process was implemented.

Statistical analysis was performed using the Wilcoxon signed-rank test, producing a statistical value of $V_{Wilcoxon} = 10{,}00$ with $p = 0{,}10$. The p (p-value) indicates that statistically formal (with a conventional significance level of $\alpha = 0{,}05$.The difference is not significant. However, the effect size calculated using the rank biserial coefficient of $\hat{r}_{rankbiserial} = 1{,}00$ shows a very large effect, meaning that all data pairs experience a consistent increase in MI values. This is reinforced by the very narrow 95% confidence interval, which ranges from 1.00 to 1.00, indicating no doubt about the direction of the effect. The number of data pairs used in this analysis was 7 (n_pairs = 7), which illustrates the existence of 7 code entities tested before and after decomposition. The small sample size is a limitation, yet the consistency of improvement across all samples underscores the practical benefit of the method.

This study also recognizes several potential threats that may affect the validity of its results. In terms of internal validity, all refactoring decisions were generated automatically by the Genetic Algorithm-based tool without any manual intervention. This automation minimizes subjective bias and improves the reliability of the experimental process. However, internal validity may still be influenced by possible hidden bugs or technical limitations within the implementation, particularly in the fitness evaluation or clustering process, which could alter the accuracy of refactoring results. Regarding construct validity, maintainability was assessed using the Maintainability Index (MI), which primarily focuses on code complexity, size, and the presence of comments. Although MI is widely accepted in software engineering research, it does not comprehensively capture all aspects of maintainability, such as testability, documentation quality, or adaptability. In addition, the detection of design smells was based on class diagram features, potentially overlooking certain design flaws not represented in static diagrams.

External validity may also pose a challenge, as the study utilized only a single module from FILKOM Apps, a PHP-based academic system. Consequently, the findings may not generalize to software systems developed in different programming languages, frameworks, or domains. Further empirical studies involving more diverse systems are necessary to confirm the broader applicability of the proposed approach. Finally, in terms of statistical conclusion validity, the limited number of analyzed classes before and after refactoring ($n = 7$) constrains the statistical strength of the results. Although the effect size observed was relatively strong, the Wilcoxon signed-rank test yielded a non-significant p-value ($p = 0.10$), suggesting that the dataset size may not have been sufficient to achieve statistical significance. Future work should involve larger datasets and additional statistical analyses to enhance the robustness and generalizability of the conclusions.

According to experimental results, this study makes a significant contribution to the understanding of the benefits of class decomposition-based refactoring and can serve as a basis for further research involving larger datasets to obtain stronger statistical evidence.

6. Conclusion

This study evaluates the impact of class decomposition on software maintainability using the Maintainability Index (MI) metric. The analysis shows a clear and substantial improvement in maintainability after the decomposition process. Specifically, the median MI value increases from approximately 28.25 before decomposition to about 70.90 after decomposition, reflecting a significant enhancement in code quality. Although the Wilcoxon signed-rank test produces a p-value of 0.10 that slightly above the conventional 0.05 threshold. In this study, the rank biserial effect size reaches 1.00 with a confidence interval of [1.00, 1.00], indicating a strong and consistent improvement across all analyzed samples. While the small sample size limits the statistical power of the results, the uniform upward trend in MI values confirms the practical effectiveness of the proposed class decomposition method in improving maintainability at the design level.

For future research, it is essential to test the approach on a larger and more diverse dataset to achieve greater statistical validity and reliability. Expanding the dataset will enable more comprehensive analyses, allowing researchers to identify complex relationships and maintainability patterns that smaller datasets might obscure. Future studies should also involve various types of software projects from different application domains to enhance the generalizability of the findings. Moreover, the integration of optimization-based decomposition methods, such as those utilizing genetic algorithms or hybrid metaheuristics. It should be explored to make the decomposition process more efficient, adaptive, and scalable to complex systems. Through these advancements, future research can provide deeper insights and stronger empirical evidence to support the continuous improvement of software maintainability and refactoring strategies.

References

1. Sommerville, *Software Engineering*, 9th ed. Harlow, England: Addison-Wesley Professional, 2010.
2. R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. USA: McGraw-Hill, Inc., 2009.
3. B. Priyambadha and T. Katayama, "The Impact of Design-level Class Decomposition on Software Maintainability," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 4, p. 2023, 2023, doi: 10.14569/IJACSA.2023.0140445.
4. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018, doi: 10.1007/s10664-017-9535-z.
5. G. Szőke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Empirical Study on Refactoring Large-scale Industrial Systems and Its Effects on Maintainability," *Journal of Systems and Software*, vol. 129, pp. 107–126, 2017, doi: 10.1016/j.jss.2016.08.071.
6. J. D. A. G. Saraiva, M. S. De França, S. C. B. Soares, F. J. C. L. Filho, and R. M. C. R. De Souza, "Classifying Metrics for Assessing Object-Oriented Software Maintainability: A Family of Metrics' Catalogs," *Journal of Systems and Software*, vol. 103, pp. 85–101, 2015, doi: 10.1016/j.jss.2015.01.014.
7. Yamashita and S. Counsell, "Code Smells as System-Level Indicators of Maintainability: An Empirical Study," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2639–2653, 2013, doi: 10.1016/j.jss.2013.05.007.
8. D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using Metrics to Evaluate Software System Maintainability," *IEEE Computer*, vol. 27, no. 8, pp. 44–49, 1994.
9. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, 2nd ed. United States of America: Pearson Education - Wesley, 2019.
10. S. Rochimah, S. Arifiani, and V. F. Insanittaqwa, "Non-source Code Refactoring: A Systematic Literature Review," *International Journal of Software Engineering and its Applications*, vol. 9, no. 6, pp. 197–214, 2015, doi: 10.14257/ijseia.2015.9.6.19.
11. M. Misbhauddin and M. Alshayeb, "UML Model Refactoring: A Systematic Literature Review," *Empirical Software Engineering*, vol. 20, no. 1, pp. 206–251, 2013, doi: 10.1007/s10664-013-9283-7.
12. B. Priyambadha, T. Katayama, Y. Kita, H. Yamaba, K. Aburada, and N. Okazaki, "The Seven Information Features of Class for Blob and Feature Envy Smell Detection in a Class Diagram," *The 2021 International Conference on Artificial Life and Robotics (ICAROB2021)*, pp. 348–351, 2021.
13. B. Priyambadha et al., "Detection of Blob and Feature Envy Smells in a Class Diagram Using Class's Features," *Journal of Robotics, Networking and Artificial Life*, vol. 9, no. 1, pp. 43–48, 2022, doi: 10.57417/JRNAL.9.1_43.
14. G. Saranya, H. Khanna Nehemiah, A. Kannan, and V. Nithya, "Model Level Code Smell Detection Using EGAPSO Based on Similarity Measures," *Alexandria Engineering Journal*, vol. 57, no. 3, pp. 1631–1642, 2018, doi: 10.1016/j.aej.2017.07.006.
15. D. Rattan, R. Bhatia, and M. Singh, "Model Clone Detection Based on Tree Comparison," *INDICON 2012, Annual IEEE India Conference*, pp. 1041–1046, 2012, doi: 10.1109/INDCON.2012.6420770.
16. B. Priyambadha and T. Katayama, "Considering the Semantic Approach to Assess Class Cohesion," *Japan Symposium on Software Testing 2020 Tokyo (JaSST'20 Tokyo)*, Tokyo, Mar. 2020, pp. 236–246.
17. B. Priyambadha and T. Katayama, "The Impact of Design-Level Class Decomposition on Software Maintainability," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 4, pp. 405–413, 2023, doi: 10.14569/IJACSA.2023.0140445.
18. B. Priyambadha, N. Takahashi, and T. Katayama, "A Genetic Algorithm-based Approach for Design-level Class Decomposition," 2024. [Online]. Available: www.ijacsa.thesai.org.
19. M. Hamdi, R. Pethe, A. S. Chetty, and D. K. Kim, "Threshold-driven Class Decomposition," *Proceedings - International Computer Software and Applications Conference*, vol. 1, pp. 884–887, 2019, doi: 10.1109/COMPSAC.2019.00130.

20. B. Priyambadha and T. Katayama, “Design-Level Class Decomposition Using Threshold-based Hierarchical Agglomerative Clustering,” *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 3, pp. 57–64, 2022, doi: 10.14569/IJACSA.2022.0130310.
21. B. Priyambadha and T. Katayama, “Enhancement of Design-Level Class Decomposition Using Evaluation Process,” *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 8, pp. 130–139, 2022, doi: 10.14569/IJACSA.2022.0130816.
22. Z. Wu and M. Palmer, “Verb Semantics and Lexical Selection,” *32nd Annual Meeting on Association for Computational Linguistics*, pp. 133–138, 1994, doi: 10.3115/981732.981751.
23. R. Dijkman, M. Dumas, B. van Dongen, R. Käärik, and J. Mendling, “Similarity of Business Process Models: Metrics and Evaluation,” *Information Systems*, vol. 36, no. 2, pp. 498–516, Apr. 2011, doi: 10.1016/j.is.2010.09.006.