Software Quality Enhancement Through Code Refactoring: An Empirical Evaluation of Roblox Game Development

Enrico Almer Tahara
Department of Informatics
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
6025241062@student.its.ac.id

Abdullah Faqih Septiyanto
Department of Informatics
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
7025221022@student.its.ac.id

Riyanarto Sarno
Department of Informatics
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
riyanarto@its.ac.id

Shoffi Izza Sabilla
Department of Medical Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
shoffi@its.ac.id

2025 International Conference on Computer Sciences, Engineering, and Technology Innovation (ICoCSETI) | 979-8-3315-0861-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICoCSETI63724.2025.11019538

***Abstract*-Roblox, a prominent platform in the metaverse, is recognized for its intuitive interface, robust developer support, and customizable features, making it ideal for creating virtual experiences. However, maintaining a clean and efficient code structure in game development is crucial as projects scale. Roblox Lua-based scripting environment presents challenges when managing complex codebases, particularly when critical functions are housed within a monolithic script, leading to "code smell" and lower software quality. This research focuses on addressing these challenges through code refactoring, a process that improves software quality by restructuring without altering its external behavior. The study applies various refactoring methods to a monolithic game script to enhance its software quality, using metrics such as the Number of Operations, Lack of Cohesion Metrics, Instability, Cyclomatic Complexity, Service Granularity Metrics, and Maintainability Metrics to assess improvements. The results indicate substantial improvements post-refactoring, with the NOO reduced from 17 to 5, LCOM from 0.900 to 0.556, Instability from 1.000 to 0.376, CC from 102.000 to 17.667, and MI increased from 41.897 to 84.505. These enhancements demonstrate that refactoring effectively boosts the scalability, adaptability, and quality of the game codebase, supporting its continued growth and potential expansion into additional educational subjects by maintaining a well-structured, modular codebase.**

Keywords—code refactoring, software quality, software quality metrics, Roblox, game development

I. INTRODUCTION

Roblox, a prominent platform in the metaverse, fosters interactive learning, creativity, and collaboration through a vast ecosystem of user-generated content and a global community [1]. However, for game development, maintaining high software quality is crucial for performance, updates, and player experience. While Roblox Lua-based scripting environment supports creativity, it also creates challenges with complex codebases, especially when core functions are confined to monolithic script. Over time, this can lead to "code smell," where problematic patterns emerge, increasing complexity and lowering software quality [2].

As games encounter these challenges, refactoring stands as a key solution. Refactoring involves restructuring the source code without changing its behavior [3][13][14]. By addressing code smells through refactoring, it helps developers manage complexity and improve software quality [3]. It is a proactive approach to ensure robust and extensible codebases that adapts alongside the game development.

Software quality ensures compliance with requirements and standards through criteria like functionality, usability, reliability, performance, supportability, usability, efficiency, and maintainability [17]. Software quality is assessed via measurable metrics, like the Maintainability Index (MI) [4][10], Cyclomatic Complexity [9][15][18], Granularity [5][7], and Instability [5][8] help assess and predict software quality, providing insights for refactoring [5]. Refactoring, when guided by these metrics, can significantly improve internal structure, and readability without changing external behavior, ultimately enhancing the overall quality of the software [6][16].

Previous research highlights the role of software quality, particularly on maintainability metrics. For example, Yenduri and Veeranjaneyulu [4] researched the maintainability of open-source gaming applications written in C, C++, and Java, analyzing how the choice of programming language affects key metrics such as Lines of Code (LOC), Cyclomatic Complexity (G), and Halstead's Volume (V). Their findings suggest that while maintainability is generally sufficient, there are significant variations between programming languages, making accurate predictions challenging. This highlights the need to explore how maintainability metrics impacted in niche environments like Roblox Lua, which has not been extensively researched in relation to game development. Additionally, Estevam et al. [12], which investigated the effect of design patterns on Unity games, found a 5.5% improvement in code maintainability after applying design patterns to the study's limited scope. This highlights a need for larger-scale studies, especially in platforms like Roblox, where the development practices and scripting environments differ considerably from more traditional game engines.

Building upon previous research on software quality and refactoring, this study introduces novel contributions by applying refactoring techniques to Roblox to address issues like monolithic scripting that hinder game scalability. Using software quality metrics such as Maintainability Index, Cyclomatic Complexity, and Instability, it evaluates the impact of refactoring on Roblox game scripts. The case study of this research focuses on an educational Roblox game titled "Jelajah Warisan ASEAN," accessible through https://www.roblox.com/games/17589073335/, which provides an alternative tool for elementary and middle school students to learn about the cultures of ASEAN member countries through interactive activities. This research

demonstrates how refactoring improves both the technical quality and scalability of the game.

II. Literature Review

A. Number of Operations (NOO)

The Number of Operations (NOO) metric counts the operations in a script, where the greater number of methods in the script results to increased complexity [5][7]. NOO can be calculated using (1).

$$NOO = \sum_{i=1}^{n} M \quad (1)$$

where:
M : Number of operations

B. Lack of Cohesion (LCOM)

LCOM evaluates the cohesiveness level of operations within a script module by evaluating their similarity and interrelation, based on Henderson-Sellers's metric [5][7], as seen on (2).

$$LCOM = 1 - \left(\frac{\sum_{i=1}^{n} MF}{M \times F}\right) \quad (2)$$

where:
M : Number of operations
F : Number of unique parameters
MF : Occurrence of a parameter
n : Number of operations in the script

The average LCOM value reflects the overall cohesion of the application, with a score closer to unity indicating less cohesion and higher complexity, while a score of 0 represents perfect cohesion, meaning the operations are fully related [7]. To achieve the overall system cohesion, the ALCOM formula of (3) is used.

$$ALCOM = 1 - \left(\frac{\sum_{i=1}^{n} LCOM}{NS}\right) \quad (3)$$

where:
NS : Number of scripts
n : Number of operations in a script

C. Instability (I)

Instability measures the stability of a package by counting the dependencies that enter and leave it [5][8]. It is calculated using (4):

$$I = \frac{Ce}{Ce + Ca} \quad (4)$$

where:
Ce : Efferent coupling
Ca : Afferent coupling

$$Ce = \sum E_{dep} \quad (5)$$

$$Ca = \sum A_{dep} \quad (6)$$

where:
E_{dep} : Number of outgoing dependencies
A_{dep} : Number of incoming dependencies

The instability metric ranges from 0 to 1, with a score of 0 indicating a maximally stable package and a score of 1 indicating a maximally unstable package with a higher score suggests the package is more likely to change, while a lower score indicates greater resistance to change [8][11].

D. Cyclomatic Complexity (CC)

Cyclomatic Complexity (CC), introduced by McCabe, measures the structural complexity of a module by assessing its control flow [9][15][18]. CC is calculated by (7)

$$CC = d + 1 \quad (7)$$

where:
d : Decision points (loop, conditional, switch case)

It calculates the total amount of decision points in the code, with higher values indicating greater complexity and requiring more test cases and maintenance effort [9].

E. Service Granularity Metrics (SGM)

The Service Granularity Metric (SGM) evaluates a granularity of the game through Data Granularity of a Service (DGS) and Functional Granularity of a Service (FGS) to assess each overall granularity of the script, ensuring it is neither too fine nor too coarse-grained, indicating a well-designed service [5][7]. SGM is computed using (8).

$$SGM = \sum_{i=1}^{n} DGS \times FGS \quad (8)$$

where:
DGS : Data Granularity of a Service
FGS : Functional Granularity of a Service
n : Number of operations in a script

DGS measures the input and output data size for each operation, with values closer to 1 indicating coarser data use and values near 0 indicating finer granularity [7]. DGS is computed using (9).

$$DGS = \frac{IPR}{\sum_{i=1}^{n} FP} + \frac{OPR}{\sum_{i=1}^{n} FP} \quad (9)$$

where:
IPR : Input parameters in an operation
OPR : Output parameters in an operation
FP : Total input parameters in a script
n : Number of operations in a script

FGS assesses functional granularity by assigning weights to Create, Read, Update, Delete (CRUD) operations based on data manipulation levels, with create operations weighted at 4, read operations at 3, delete operations at 2, and read operations at 1, with the values indicates functional complexity of each operation within the microservice [7]. FGS is computed using (10).

$$FGS = \frac{OT}{\sum_{i=1}^{n} O} \quad (10)$$

where:
OT : Weight for a specific operation in a script
O : Total weights in a specific script

ASGM calculates the average SGM across all microservices in an application, providing an overall granularity score to evaluate the structure of the application, where a balanced ASGM score reflects a well-designed application [7]. ASGM is computed using (11).

$$ASGM = 1 - \frac{\sum_{i=1}^{n} SGM}{NS} \quad (11)$$

where:
SGM : Service Granularity Metrics
NS : Number of scripts
n : Number of operations in a script

F. Maintainability Index (MI)

The Maintainability Index (MI) assesses software maintainability, with higher values indicating easier maintenance. Refined over time to better capture maintainability factors, the improved codebase, $MI_{impr.}$, replaces Halstead's Effort metric with Halstead's Volume for greater consistency. MI is calculated using (12), while Halstead's Volume is calculated using (13)

$$MI_{impr.} = 171 - 5.2 \times \ln(aveV) - 0.23 \times \text{aveV(g)} - 16.2 \times \ln(aveLOC) \quad (12)$$

where:
aveV : Avg. Halstead's Volume per module
aveV(g) : Avg. Cyclomatic Complexity per module
aveLOC : Avg. Lines of Code per module

$$V = N \times log_2 n \quad (13)$$

where:
N : Number of tokens (operators + operands)
n : Number of unique tokens (operators + operands)

III. Methodology

This study aims to enhance the software quality of a Roblox game by refactoring its monolithic Lua script. The initial disorganized structure of the script hinders modifications and expansion. By refactoring, the code readability is improved, redundancy is reduced, and functionality is streamlined, ensuring scalability and easier future updates. The study compares pre-refactoring and post-refactoring software quality metrics, highlighting the advantages of modularity and improved code structure. Fig. 1 depicts the flowchart of the proposed method.

A. Code Renaming

Code renaming is a process of renaming the script to reflect its function, as well as multiple functions and variables in the script to better reflect their uses. This enhances readability and understandability, contributing to maintainability and usability.

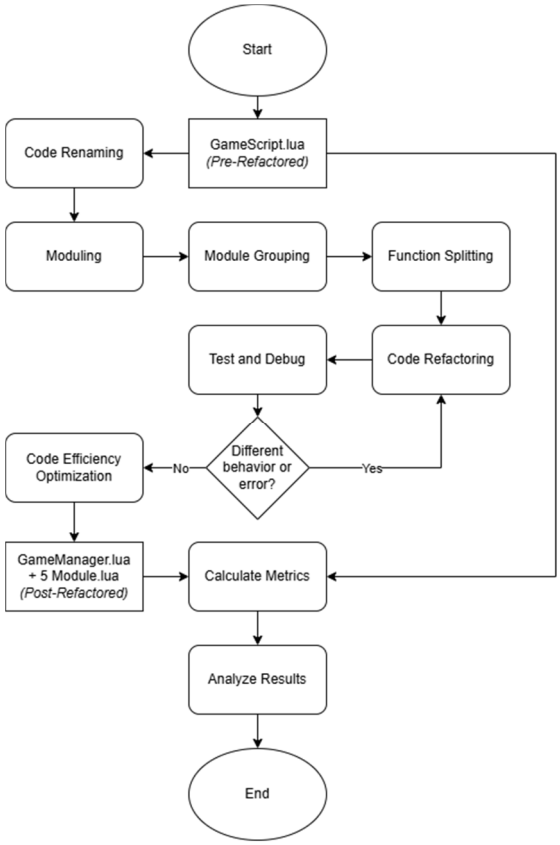

Fig. 1. Proposed method

B. Moduling

Moduling is a process of breaking down large functions in GameManager into smaller, more specific modules. This means organizing the code into separate, focused modules that handle specific tasks, making the code becomes easier to manage, more maintainable, and better optimized for extensibility and efficiency.

C. Module Grouping

Module grouping is a process of organizing modules that are focused on similar tasks into separate ModuleScripts. This means grouping related functionalities together into distinct modules or scripts based on their objectives, improving the structure, maintainability, and usability by aligning with logical objectives.

D. Function Splitting

Function splitting is a process of splitting large functions into smaller, single-purpose functions for better maintainability and reliability.

E. Code Refactoring

Code refactoring involves enhancing the structure, design, and readability of existing code while preserving its original behavior and functionality. This process supports maintainability, reliability, and efficiency by addressing technical debt and improving internal design.

F. Test and Debugging

Testing and debugging are ongoing processes, as we perform them each time code refactoring is initiated. We test whether the game behaves the same as the pre-refactoring

codebase and debug any errors that arise, contributing to reliability and correctness.

G. Code Efficiency Optimization

Code Efficiency Optimization is a process of optimizing the code for better performance by reducing redundancy, improving algorithms, and enhancing resource usage. This process improves efficiency and reliability, ensuring smoother functionality.

H. Calculate Metrics

The pre-refactored and post-refactored codebases were compared using NOO, ALCOM, Instability, CC, ASGM, and MI metrics, manually computed in Google Sheets due to the lack of automated tools for analyzing Roblox Lua scripts. While this manual approach required more effort, it allowed for a flexibility in handling the codebase's specific structure. To ensure accuracy, iterative validation and recalculation of metrics were performed prior to data entry in Google Sheets. Although this approach offers flexibility, it comes with potential human error. Future research could explore developing automated tools for Roblox script analysis. Furthermore, a custom script was created to automate the counting of LOC and Halstead's Volume, streamlining the calculation process.

Table I provides criteria for evaluating software quality metrics in the refactored game codebase, specifying the scales and relevant quality attributes. The post-refactored codebase should show lower values for most metrics being used, while MI being higher.

TABLE I. CRITERIA INFORMATION OF EACH METRICS

Criteria	*Scale*	*Quality Attributes*	*Description*
Average Number of Operations (ANOO)	$0 < ANOO < 10$	Maintainability Readability	In the range, more better
Average Lack of Cohesion Metrics (ALCOM)	$0 < ALCOM \leq 0.8$	Maintainability Reusability	More lower, more better
Average Instability (AvgI)	$0 < AvgI < 0.5$	Maintainability Reliability	Towards stability 0, more better
Average Cyclomatic Complexity (AvgCC)	$CC > 0$	Complexity	More higher, more complex
Average Service Granularity Metrics (ASGM)	$0.2 \leq ASGM \leq 0.6$	Granularity	In the range, more better
Maintainability Index (MI)	$MI > 65$	Maintainability	More higher, more better

TABLE II. COMPARISON OF PRE-REFACTORED AND POST-REFACTORED CODEBASE

Pre-Refaactored		*Post-Refactored*	
Script Name (.lua)	*Script Type*	*Script Name (.lua)*	*Script Type*
GameScript	Server Script	GameManager	Server Script
		FlagsHandler Module	Module Script
		PhotosHandler Module	Module Script
		QuizHandler Module	Module Script
		PlayerUtility Module	Module Script
		TweenModule	Module Script

IV. RESULT AND DISCUSSION

This section discusses the refactoring outcome of the educational game, focusing on GameScript, which manages core processes. The script was modularized to enhance readability and maintainability. A comparison of software quality metrics between the pre-refactored and post-refactored codebases highlights the improvements made.

A. Code Refactoring

Table II compares the pre-refactored and post-refactored game scripts. The original monolithic GameScript, which contained all game functions, was refactored into smaller, task-specific modules. This modular approach enhances maintainability and scalability, with the GameManager script handling core process and other module scripts handling individual game features.

B. Software Quality Metrics

Table III presents the NOO comparison, showing the pre-refactored codebase at 17, while the post-refactored scripts range from 2 to 12. Lack of Cohesion Metrics (LCOM): Table IV highlights the LCOM results, where the pre-refactored script has a high lack of cohesion at 0.9, and the post-refactored modules show values between 0.333 and 0.885. Table V displays the instability metric, with the pre-refactored script showing high instability at 1, while the post-refactored modules achieve stability with values from 0.111 to 0.4. Table VI compares CC, showing the pre-refactored codebase at 102, while post-refactored modules range from 1 to 39, indicating reduced complexity. Table VII focuses only on the post-refactored scripts, showing SGM values between 0.124 and 1; the pre-refactored script is not shown, as it results in a fine-grained SGM value despite being a monolithic script. Table VIII compares the MI, where the pre-refactored codebase scores 101.660, while the post-refactored modules range from 118.332 to 144.896, reflecting improved maintainability. To automate the calculation of Lines of Code and Tokens per module, two custom scripts were created manually in Roblox Studio. Finally, Table IX provides an overview of both codebases, averaging the post-refactored metrics for a clear comparison across all evaluated criteria.

C. Discussion

The refactoring process has led to significant improvements across all evaluated metrics, with each metric for the post-refactored scripts shown in Table IX meeting or exceeding the goals and scales defined in Table I, confirming the effectiveness of adopting the modularization approach. These outcomes align with previous studies, such as Yenduri and Veeranjaneyulu [4], who emphasized the importance of maintainability metrics in gaming applications. While their research highlighted the differences across various programming languages, this study extends their work by exploring how these metrics are influenced in the niche environment of Roblox Lua, which has not been extensively explored in game development research.

TABLE VII. NOO COMPARISON OF PRE-REFACTORED AND POST-REFACTORED CODEBASE

Pre-Refactored		*Post-Refactored*	
Script Name (.lua)	***NOO***	***Script Name (.lua)***	***NOO***
GameScript	17	GameManager	6
		FlagsHandler Module	6
		PhotosHandler Module	12
		QuizHandler Module	2
		PlayerUtility Module	4
		TweenModule	2

TABLE VIII. LCOM COMPARISON OF PRE-REFACTORED AND POST-REFACTORED CODEBASE

Pre-Refactored		*Post-Refactored*	
Script Name (.lua)	***LCOM***	***Script Name (.lua)***	***LCOM***
GameScript	0.900	GameManager	0.738
		FlagsHandler Module	0.736
		PhotosHandler Module	0.885
		QuizHandler Module	0.300
		PlayerUtility Module	0.675
		TweenModule	0.333

TABLE IX. INSTABILITY COMPARISON OF PRE-REFACTORED AND POST-REFACTORED CODEBASE

Pre-Refactored		*Post-Refactored*	
Script Name (.lua)	***I***	***Script Name (.lua)***	***I***
GameScript	1.000	GameManager	1.000
		FlagsHandler Module	0.214
		PhotosHandler Module	0.333
		QuizHandler Module	0.400
		PlayerUtility Module	0.200
		TweenModule	0.111

The improvements observed, such as the reduction in Average Number of Operations (ANOO) to an average of 5 and Average Cyclomatic Complexity (AvgCC) to 17.667 both demonstrates significant improvements in code simplicity and maintainability. The lower ANOO highlights enhanced script readability, as each module now serves a singular purpose. This modular approach aligns with best practices in game development and supports findings from previous research [4][12]. Meanwhile, the reduction in AvgCC indicates simplified decision-making pathways within the scripts, improving testability, reducing error risk, and making the scripts easier to understand and maintain. Together, these changes lead to a more manageable and scalable codebase.

Similarly, the decrease in the Average Lack of Cohesion Metrics (ALCOM) to an average of 5 and Average Instability (AvgI) to 0.376 highlights improved module cohesion and stability. The lower ALCOM indicates stronger cohesion within the modules, minimizing internal dependencies and making the code easier to adapt and reuse. Meanwhile, the reduced AvgI reflects reduced interdependence between modules, allowing for simpler modifications with minimal risk of unintended consequences. These improvements

TABLE III. CC COMPARISON OF PRE-REFACTORED AND POST-REFACTORED CODEBASE

Pre-Refactored		*Post-Refactored*	
Script Name (.lua)	***CC***	***Script Name (.lua)***	***CC***
GameScript	102	GameManager	29
		FlagsHandler Module	39
		PhotosHandlerModule	28
		QuizHandler Module	1
		PlayerUtility Module	6
		TweenModule	3

TABLE IV. SGM COMPARISON OF PRE-REFACTORED AND POST-REFACTORED CODEBASE

Post-Refactored	
Script Name (.lua)	***SGM***
GameManager	0.124
FlagsHandlerModule	0.401
PhotosHandlerModule	0.191
QuizHandlerModule	1.000
PlayerUtilityModule	0.428
TweenModule	0.500

TABLE V. MI COMPARISON OF PRE-REFACTORED AND POST-REFACTORED CODEBASE

Pre-Refactored		*Post-Refactored*	
Script Name (.lua)	***MI***	***Script Name (.lua)***	***MI***
GameScript	41.897	GameManager	65.030
		FlagsHandler Module	61.413
		PhotosHandlerModule	74.365
		QuizHandler Module	111.192
		PlayerUtility Module	104.808
		TweenModule	99.224

TABLE VI. COMPARISON METRICS

Metrics	*Pre-Refactored*	*Post-Refactored*
ANOO	17	5
ALCOM	0.900	0.556
AvgI	1.000	0.376
AvgCC	102.000	17.667
ASGM	N/A	0.441
AvgMI	41.897	84.505

collectively contribute to a more reliable, maintainable, and scalable codebase.

The improvements in Average Service Granularity Metrics (ASGM) and Maintainability Index (MI) underscore the balance achieved between granularity and maintainability. The ASGM value of 0.441 reflects optimal modularity without over-fragmentation, while the MI value of 84.505 confirm that the codebase is now more accessible and easier to extend, with new features can be implemented with minimal impact on existing modules. This improvement aligns with the findings of Yenduri and Veeranjaneyulu [4], which emphasized the potential of the MI, despite challenges in its consistency across various programming languages.

Overall, the refactored codebase meets game development standards for software quality, positioning it as a robust foundation for future iterations of the educational game being researched. This approach could also serve as a model for similar projects, where code refactoring is needed

to increase its scalability. Future work could explore the scalability of this approach in more complex projects or evaluate its effectiveness across other programming environments, as suggested by previous studies [4][12].

V. Conclusion

Developing an educational game in Roblox presents challenges in maintaining software quality, especially when the main game script is overly monolithic. This structure complicates debugging, modifying, and scaling, as changes risk disrupting the functionality of the game. Refactoring is essential to enhance the software quality and support future developments without affecting its original behavior.

This study improved the software quality of a Roblox educational game by refactoring its monolithic main script using techniques such as code renaming, function splitting, and module grouping, which improved modularity, cohesion, and MI. Simplified decision paths lowered CC, and minimized ALCOM and Instability, resulting in a more reliable codebase, allowing for easier future modifications with minimal risk.

In summary, refactoring significantly improved the scalability, adaptability, and maintainability of the game. This research highlights the critical role of code refactoring in software development, particularly for complex and evolving projects like games, where maintaining a clean and manageable codebase is essential for long-term success.

Acknowledgement

This research was funded by Institut Teknologi Sepuluh Nopember (ITS) under Penelitian Kolaborasi Pusat and under the Project Scheme of the Publication Writing and Intellectual Property Rights (IPR) Incentive (Penulisan Publikasi dan Hak Kekayaan Intelektual/PPHKI) Program 2024.

References

[1] Ho, W.; Lee, D. (2023). Enhancing Engineering Education in the Roblox Metaverse: Utilizing ChatGPT for Game Development for Electrical Machine Course. *International Journal on Advanced Science Engineering and Information Technology*, vol. 13, no. 3, Insight Society, 1052–52. https://doi.org/10.18517/ijaseit.13.3.18458.

[2] Lacerda, G.; Petrillo, F.; (2020) Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations. *Journal of Systems and Software*, vol. 167, Elsevier BV. 110610–10. https://doi.org/10.1016/j.jss.2020.110610.

[3] Händler, T.; Neumann, G. (2020). Ontology-Based Analysis and Design of Educational Games for Software Refactoring. *Communications in Computer and Information Science*, Springer Science+Business Media, 602–28. https://doi.org/10.1007/978-3-030-58459-7_29.

[4] Yenduri, G.; Veeranjaneyulu, N. (2019). An Analysis of Maintainability Index Influencing Metrics and Their Behavior on Similar Open Source Gaming Application Developed in C, C++ And, JAVA. *Ingénierie Des Systèmes D Information*, vol. 24, no. 1, 83–87. https://doi.org/10.18280/isi.240112.

[5] Vera-Rivera, F. H.; Gaona, C.; Astudillo, H. (2021). Defining and Measuring Microservice Granularity—a Literature Overview. *PeerJ Computer Science*, vol. 7, PeerJ, Inc., e695–95. https://doi.org/10.7717/peerj-cs.695.

[6] Gradišnik, M.; Karakatič, S.; Beranič, T.; Heričko, M.; Mauša, G.; Grbac, T. G. (2019). The impact of refactoring on maintability of Java code: A preliminary review. *SQMIA*, vol. 2508. https://doi.org/10.1109/tse.2004.1265817.

[7] Al-Debagy, O.; Martinek, P. (2020). A Metrics Framework for Evaluating Microservices Architecture Designs. *Journal of Web Engineering, River Publishers*. https://doi.org/10.13052/jwe1540-9589.19341.

[8] Martin, R. (2014). Agile Software Development, Principles, Patterns, and Practices, 1st ed., *Pearson Education Limited*.

[9] Wijendra, D. R.; Hewagamage, K. P. (2021) Analysis of Cognitive Complexity with Cyclomatic Complexity Metric of Software. *International Journal of Computer Applications*, vol. 174, no. 19, 14–19. https://doi.org/10.5120/ijca2021921066.

[10] Heričko, T.; Šumak. B. (2023). Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems. *Applied Sciences*, vol. 13, no. 5, MDPI AG, 2972. https://doi.org/10.3390/app13052972.

[11] Santos, D., B.; Resende, A. M. P.; Lima, E. C.; Freire, A. P. (2017) Software Instability Analysis Based on Afferent and Efferent Coupling Measures. *Journal of Software*, vol. 12, no. 1, 19–34. https://doi.org/10.17706/jsw.12.1.19-34.

[12] Estevam, V. B.; dos Santos, A. D.; Paixao, M. (2023). Design Patterns and Code Maintainability in Games: A Case Study. *Sociedade Brasileira de Computação*, 299–304. https://doi.org/10.5753/sbgames_estendido.2023.234103.

[13] Almogahed, A.; Omar, M.; Zakaria, N. H.; Muhammad, G.; AlQahtani, S. A. (2022). Revisiting Scenarios of Using Refactoring Techniques to Improve Software Systems Quality. *IEEE Access*, vol. 11, Institute of Electrical and Electronics Engineers, 28800–19. https://doi.org/10.1109/access.2022.3218007.

[14] Alshayeb. M. (2009) Empirical Investigation of Refactoring Effect on Software Quality. *Information and Software Technology*, vol. 51, no. 9, Elsevier BV, 1319–26. https://doi.org/10.1016/j.infsof.2009.04.002.

[15] Stroud, R. O.; Ertas, A.; Mengel, S. (2019). Application of Cyclomatic Complexity in Enterprise Architecture Frameworks. *IEEE Systems Journal*, vol. 13, no. 3. https://doi.org/10.1109/jsyst.2019.2897592.

[16] Agnihotri, M.; Chug, A. (2022). Understanding Refactoring Tactics and their Effect on Software Quality. *International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. doi: 10.1109/Confluence52989.2022.9734141.

[17] Papakitsos, E. C. (2022). Robust Software Quality Assurance. *Bulletin of the Georgian National Academy of Sciences*, 16(2), 23–31.

[18] Subandri, S. A.; Sarno, R. (2017). Cyclomatic Complexity for Determining Product Complexity Level in COCOMO II. *Information Systems International Conference (ISICO)*, Procedia Computer Science, vol. 124, 478–486. doi: https://doi.org/10.1016/j.procs.2017.12.180.

[19] Yonia, D. L.; Anggraini, R. N. E.; Sarno, R.; Haryono. A. T.; Septiyanto, A. F.; Mulyanto, S.. E-Learning Evaluation Using Information Technology Infrastructure Library 4 with ISO/IEC 25010 Indicators. *2024 IEEE International Conference on Artificial Intelligence and Mechatronics Systems (AIMS)*, 1–5. https://doi.org/10.1109/AIMS61812.2024.10512570.

[20] Basori, A. H.; Sarno, R.; Widyanto, S. (2008). The development of 3D multiplayer mobile racing games based on 3D photo satellite map. *Wireless and Optical Communications Networks*, 2008. WOCN '08. 5th IFIP International Conference on, 1–5. 10.1109/WOCN.2008.4542540.

[21] Putri, R. A.; Nevin, M.; Ansori, D. B.; Septiyanto, A. F.; Sarno, R. (2024). Automated Extraction of Microservice Architecture Using a Rule-Based Approach. *2024 2nd International Conference on Technology Innovation and Its Applications (ICTIIA)*. 1–6. https://doi.org/10.1109/ictiia61827.2024.10761417.

[22] Wardhiana, I. N. G. A. M.; Ghufron, M. Z.; Jati, S. M.; Septiyanto, A. F.; Sarno, R. (2024). Optimizing Microservices Architecture Using Multi-Criteria Decision Making (MCDM): A Case Study Of PayPal REST API. *2024 11th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, 308–313. https://doi.org/10.1109/icitacee62763.2024.10762818