DOI: https://doi.org/10.15276/hait.09.2026.37
UDC 004.43:004.89

Evaluating the effectiveness of multi-step automated source code refactoring using a large language model

Andrii I. Hryshchenko[1)]
ORCID: https://orcid.org/0009-0007-1191-7948; andrew.hryshchenko@gmail.com. Scopus Author ID: 60278347900
Nataliia O. Komleva[2)]
ORCID: https://orcid.org/0000-0001-9627-8530, komleva@op.edu.ua.Scopus Author ID: 57191858904
[1)] Lowe's Companies, Inc. Mooresville, NC, USA
[2)] Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine

ABSTRACT

Topicality: modern large language models open up new opportunities for automating source code refactoring; however, most existing studies focus on individual refactoring operations, while the automated execution of sequences of such operations remains insufficiently studied. **The aim of the article** is to study the effectiveness of using a large language model for automated multi-step refactoring of Java program code. **Objectives**: to develop multi-step refactoring scenarios, conduct an experimental evaluation of their execution by the large language model, analyze the influence of the initial characteristics of the source code on the success of refactoring, compare the effectiveness of different scenarios and identify typical errors that occur during the execution of multi-step transformations. **Methods**: the effectiveness was assessed by the Maintainability Index, Cyclomatic Complexity, Cognitive Complexity, Lines of Code, the number of Code Smells, as well as successful compilation and unit test execution. **Scientific novelty**: lies in assessing the effectiveness of a large language model for performing multi-step refactoring, which involves a sequence of interrelated operations, as well as studying the influence of program code characteristics and the complexity of scenarios on the success of their automated execution. **Practical significance**: the results obtained can be used in development of intelligent automated refactoring tools, integration of large language models into software development environments, and improvement of source code quality assurance processes. **Results**: the experimental study showed that the large language model successfully performed multi-step refactoring scenarios, ensuring an improvement in the primary indicators of source code quality. **Conclusions**: it was established that the effectiveness of automated refactoring depends on both the initial characteristics of the source code and the complexity of the refactoring scenario, and the main limitations remain the execution of long sequences of interrelated operations and ensuring full functional equivalence of the program code.

Keywords: Multi-step refactoring; code refactoring; large language models; Java, code quality; programming automation; artificial intelligence

For citation: **Hryshchenko A. I., Komleva N. O. "Evaluating the effectiveness of multi-step automated source code refactoring using a large language model". *Herald of Advanced Information Technology*. 2026. Vol.9. No.4: 605–619. DOI:https://doi.org/10.15276/hait.09.2026.37**

INTRODUCTION

The modern development of large language models (LLM) significantly transforms approaches to software development. In general, the use of modern information technologies, methods of data quality assessment, their visualization and digital decision support tools is an important factor in increasing the efficiency of information systems in various subject areas [1]. Due to the ability to analyze source code, generate program solutions, identify potential errors and suggest ways to eliminate them, large language models are becoming a promising tool for automating software engineering processes. Their use contributes to increasing developer productivity, reducing development time and improving software quality.

One of the important stages of software maintenance is refactoring of the program code, aimed at improving its internal structure without changing functional behavior. Refactoring makes it possible to improve the readability, maintainability and scalability of software systems, reduce technical debt and simplify further software development. Traditionally, such transformations are performed manually or using integrated development environments, the capabilities of which are limited mainly to typical local operations.

The emergence of large language models has opened up new opportunities for automating refactoring of source code. Unlike traditional tools, they are able to take into account the context of the program, analyze the relationships between its components, and perform more complex transformations. However, its effectiveness may depend on many factors, including the structural complexity of the software code, its size, the presence of design defects, and the number and sequence of refactoring operations required to improve the structure of the software code.

© Hryshchenko A., Komleva N., 2026

This is an open access article under the CC BY license (https://creativecommons.org/licenses/by/4.0/deed.uk)

Despite the active use of large language models in software engineering tasks, the issue of the influence of program code characteristics and the sequence of refactoring operations on the effectiveness of automated improvement of the structure of the software code remains insufficiently explored. The lack of systematic research in this area makes it difficult to determine the limits of the effective application of large language models, to formulate practical recommendations for their use, and to choose the optimal sequence of refactoring operations.

RELATED WORKS AND PROBLEM STATEMENT

Refactoring is a systematic process of changing the internal structure of software without intentionally changing its external behavior. The main purpose of refactoring is to improve the readability, modularity, extensibility, and maintainability of software code. Typical operations include extract method, renaming of program elements, moving functionality between classes, class isolation, eliminating duplication, and simplifying conditional constructs [2].

Traditional automated refactoring tools integrated into development environments implement predefined algorithms for transforming software code. Such tools are able to safely perform local operations provided that the developer has independently identified the problematic fragment, the required type of refactoring, and the transformation parameters. At the same time, traditional tools have limited capabilities in interpreting the purpose of software components, analyzing informal requirements, and selecting refactorings taking into account the broader context of the software system.

The development of pre-trained code models has laid the foundation for the application of natural language processing methods in software engineering. Models trained simultaneously on source code and natural language descriptions have demonstrated the ability to search for code, generate descriptions of code, find correspondences between natural language and program constructs, and solve other problems of program understanding [3], [4]. Further development of encoder–decoder models has expanded support for the tasks of generating, translating, correcting, and generalizing code [5]. The results of evaluating generative models have also shown that the correctness of the resulting code should be established by performing tests, and not only by analyzing syntactic similarity to a reference solution [6].

Systematic reviews of the use of large language models in software engineering mainly cover code generation, defect correction, testing, documentation, requirements analysis, vulnerability scanning, and refactoring [7]. These studies emphasize that large language models are able to take into account the semantic context and form non-trivial program transformations. However, their probabilistic nature causes the risk of syntactic errors, functionally incorrect solutions, incomplete changes, and dependence of the result on prompt formulation. Therefore, one of the promising directions is the combination of large language models, compilers, static analysis tools, and automated testing [8].

A systematic review of deep learning applications in software refactoring shows software refactoring shows that a significant portion of the existing methods is focused on detecting code smells or recommending possible transformations [9]. Less attention is paid to the end-to-end process that includes problem detection, operation selection, generation of changed code, verification of functional equivalence, and evaluation of the structural quality of the result. This indicates the need to consider automated refactoring as a multi-criteria task, and not only as generation of an alternative version of the program code.

In a study of program refactoring using several prompt examples, it was found that large language models can generate functionally correct program variants with lower cyclomatic complexity and line count [10]. Meanwhile, undesirable changes to comments, formatting, and individual elements of the original program were recorded. The results also showed that generating multiple candidate variants and subsequent automated filtering can increase the probability of obtaining a desired result.

Empirical evaluation of ChatGPT capabilities has shown that the model is able to suggest useful changes aimed at improving the quality of the program code [11]. However, the success of refactoring cannot be determined only by the presence of structural differences between the initial and generated versions. It is necessary to separately check compilation, passing tests, preservation of behavior, actual execution of the required operation and change in quality indicators.

Analysis of real developer dialogues with ChatGPT shows that requests for refactoring can be both general and aimed at performing a specific operation [12], [13]. General formulations leave the model more freedom in the choice of transformations, but make it difficult to objectively establish whether the obtained result corresponds to

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

the original intention of the user. Specialized queries make it easier to verify the execution of a particular refactoring method, but may limit the model's ability to suggest an alternative structural solution.

An iterative approach to refactoring of real open source projects proved the possibility of sequentially performing several refactoring operations with gradual improvement of individual indicators of software code quality [14]. However, changing one metric after each step does not guarantee an overall improvement in the program structure. The reduction of a single method may be accompanied by the appearance of new auxiliary methods, a change in the relationships between classes, or the transfer of complexity to other components. This indicates the need for a comprehensive assessment of the results of multi-step refactoring after each performed operation.

A study of real-world refactoring cases from software repositories has shown that the performance of large language models varies depending on the category of structural transformation [15]. Models can correctly identify individual refactoring opportunities, but sometimes generate syntactically incorrect code or change its functionality. To increase reliability, a hybrid approach is proposed, in which the language model determines the location and type of refactoring, and a specialized tool performs the direct transformation.

Evaluation of a model specialized in program code has shown that it can effectively eliminate systematic local problems, in particular magic numbers or excessively long constructs [16]. Similar structural improvements are also important for modern software systems based on microservice architecture, where maintainability and quality of program code directly affect the efficiency of system operation [17]. More complex context-dependent changes that require understanding of class responsibilities and component interactions are performed less reliably. Using an example in a prompt and generating multiple candidate variants increases the proportion of solutions that pass tests.

An important step in automated refactoring is the detection of code smells. Experimental evaluation of ChatGPT on such code smells as Blob, Data Class, Feature Envy, and Long Method examples showed that the detection accuracy depends on the type of smell and the prompt formulation [18]. This suggests that the result can be determined not only by the properties of the model, but also by how clearly the analysis criteria are described in the prompt.

A promising direction is the combination of large language models with a set of specialized tools for detecting code smells and supporting their further refactoring [19], [20]. In this approach, the language model performs a coordinating and interpretive function, while individual expert tools provide a formalized analysis of different types of structural problems. This makes it possible to partially compensate for the limitations of LLM related to the context window, the lack of complete information about the project, and the instability of the responses.

Code intelligence tools can also support the process of reviewing proposed changes. In particular, automated code reviews systems help identify potential smells, violations of programming practices, and fragments that require additional analysis [21]. However, the results of such tools do not replace compilation and testing, since structural improvement does not guarantee the preservation of functional behavior of the program. A separate area of research is the use of large language models to detect and eliminate smells in test code, which expands the capabilities of automated software quality analysis [22].

The use of quantitatively defined quality indicators is also important in modern software engineering, where measurable metrics can provide a formalized basis for relating software quality requirements to engineering decisions and evaluating the resulting software characteristics [23].

Research on repeated, refactoring of cyclic constructs shows that the performance of large language models can vary significantly between successive refactoring iterations [24]. Reapplication of the model sometimes leads to further improvements in the structure of the program code, but in other cases it causes the accumulation of unwanted changes or the emergence of alternative implementations without a noticeable improvement in quality. This indicates the feasibility of studying multi-step automated refactoring and determining the rational sequence of execution of individual operations.

New agent-based approaches combine context-aware search, the interaction of multiple specialized agents, compilation, and test execution [25]. The results of this approach demonstrate a significant increase in the proportion of solutions that compile and pass tests compared to a direct one-time query to the language model. However, this work is available as a preprint, so its results require further independent verification.

A systematic review of LLM-oriented refactoring studies found a significant diversity in experimental sets, programming languages, refactoring types, models, queries, and evaluation metrics [26]. The lack of a unified protocol makes it

difficult to compare results across studies. Furthermore, existing works mostly evaluate the overall ability of a particular model to perform refactoring, the effectiveness of a particular query strategy, or the change in a specific metric, but rarely examine initial code characteristics as possible predictors of success.

Thus, the analysis of scientific works confirms the possibility of using large language models to identify structural problems, select refactoring operations, and generate modified program code. At the same time, the quality of the obtained result depends not only on the type of a single refactoring operation, but also on the initial characteristics of the program code, the completeness of the context, the structure of the query, and the sequence of execution of several refactoring operations.

The research problem concerns the unclear influence of the characteristics of the initial program code and the sequence of execution of refactoring operations on the effectiveness of automated refactoring using a large language model. In particular, the issue of changing source code quality indicators after each stage of multi-step refactoring, as well as determining more effective sequences of refactoring operations to achieve the optimal improvement of the program code structure without disrupting its functional behavior, remains insufficiently researched. Solving this problem involves conducting a controlled experiment with sequential execution of several refactoring operations, evaluation of quality indicators after each stage, and statistical analysis of the obtained results.

RESEARCH AIM AND OBJECTIVES

The purpose of the study is to evaluate the effectiveness of multi-step automated refactoring of software code using a large language model and determine the influence of software code characteristics and the sequence of refactoring operations on the quality of the resulting code.

To achieve the goal, it is necessary to complete the following tasks:

– analyze modern approaches to automated refactoring of software code using large language models;

– form an experimental set of software fragments with different structural characteristics and scenarios of multi-step refactoring;

– determine the characteristics of the software code that can influence the effectiveness of automated refactoring, in particular the number of lines of software code, cyclomatic and cognitive complexity, as well as the presence of structural flaws (Code Smells);

– develop scenarios for sequential execution of refactoring operations using a large language model;

– conduct multi-step automated refactoring of software code using a single query generation method;

– evaluate the results of each stage of refactoring according to the criteria of functional equivalence, success of the refactoring scenario execution, and changes in software code quality indicators;

– analyze the impact of program code characteristics and the sequence of refactoring operations on the effectiveness of multi-step automated refactoring and identify the characteristics of a successful outcome.

METHODOLOGY OF AUTOMATED SOFTWARE CODE REFACTORING

Refactoring methods used in the study

In modern software engineering, there are a significant number of refactoring methods aimed at improving the structure of the program code without changing its functional behavior. The choice of a specific method depends on the nature of the identified structural defect, the architectural features of the software system and the goals of modification. The second edition of the book by M. Fowler describes more than 70 refactoring methods that cover various aspects of improving the program code [2].

In this study, individual refactoring methods were used as basic operations for constructing multi-step automated refactoring scenarios. Each scenario involved sequential execution of several interconnected refactoring operations aimed at gradually improving the structure of the program code without changing its functional behavior.

The scenarios included the following refactoring methods:

– ExtractMethod – separating a piece of code into a separate method to reduce the size of complex methods;

– Inline Method – moving the contents of a helper method to the place where it is called in cases where a separate method is impractical;

– Rename Variable – renaming variables to improve the clarity of the program code and more accurately reflect their purpose;

– Rename Method – renaming methods to improve code readability and clearly reflect the function they perform;

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

– Introduce Parameter Object – combining a group of related method parameters into a single object to reduce the parameter list and improve the structure of the program code;

– Move Method – moving a method to the class to which it logically belongs;

– Extract Class – separating part of the functionality into a separate class to reduce excessive responsibility of the existing class;

– Replace Conditional with Polymorphism – replacing complex conditional constructs with the polymorphism mechanism;

– Remove Dead Code – removing unused program code;

– Simplify Conditional Expressions – simplification of nested conditional constructs.

These methods were selected because they are among the most common refactoring operations, are supported by modern development tools and can be combined into various sequential automated refactoring scenarios. This made it possible to evaluate the effectiveness of performing not only individual operations, but also their sequential application using the large GPT language model.

Using the GPT model for automated refactoring

Large language models belong to the class of generative artificial intelligence models trained on large amounts of text and program data. Thanks to the self-attention mechanism, they are able to analyze the context of the program code, establish relationships between its elements, and generate modified fragments according to the task at hand.

In this study, the GPT model, which interacts with the user through natural language prompts, was used as an automated refactoring tool. Unlike traditional refactoring tools that perform separate predefined operations, GPT is able to analyze the context of the program code, interpret the user's intent, and perform a sequence of interconnected refactoring operations within a single scenario. This approach allows for incremental improvements to the structure of the program code while preserving its functional behavior.

The main advantages of using GPT for refactoring include:

– automatic analysis of program code without the need for a formal description of rules;

– the ability to perform different types of refactoring within a single query;

– taking into account the context of the program fragment when forming recommendations;

– support for various programming languages;

– generation of explanations for proposed changes.

However, the use of large language models has a number of limitations. During multi-step refactoring, the generated code may contain syntax or logic errors, change the functional behavior of the program, or perform certain operations only partially. In addition, the result of the model's work largely depends on the quality of the formulated query, the amount of context provided, the complexity of the program code, and the sequence of refactoring operations. That is why the results of automated refactoring require further verification by compiling, testing, and analyzing the quality metrics of the program code after each refactoring stage.

Formation and characterization of an experimental set of program fragments

The experimental study was conducted on a specially formed set of program fragments, which covered the most common scenarios of multi-step refactoring and different levels of program code complexity. The sample formation was carried out in accordance with the research goal - to determine the influence of program code characteristics and the sequence of refactoring operations on the effectiveness of automated refactoring using the large language model GPT.

The program fragments were developed by the authors specifically for experimental research. During their creation, typical structural flaws in the program code and the possibility of applying one or more refactoring methods within a defined scenario without changing the functional behavior of the program were taken into account.

The experimental set included developed program fragments that contained one or more typical structural flaws (Code Smells) and could be refactored without changing the functional behavior of the program. To ensure the representativeness of the sample, program fragments were used that differed in volume, structural complexity, and multi-step refactoring scenarios. All program fragments were self-sufficient, compiled independently of each other, and allowed for functional equivalence testing after refactoring.

The following criteria were taken into account when forming the sample:

– presence of one or more structural defects;

– the possibility of using one of the classic refactoring methods;

– different levels of structural complexity of the program code;

– possibility of automatic testing after refactoring;

– using the same prompt template for the GPT model;

– the ability to sequentially perform multiple refactoring operations.

The experimental set contained 30 program fragments, evenly distributed among ten multi-step refactoring scenarios. Each scenario consisted of three to five consecutive refactoring operations, formed using the methods examined in this study. Three program fragments were prepared for each scenario, which ensured a balanced distribution of program fragments across the scenarios and the possibility of correct comparison of results.

For each software fragment, characteristics were determined that were subsequently used as independent variables during statistical analysis. These include programming language, refactoring scenario, number of lines of code (LOC), cyclomatic complexity (CC), cognitive complexity (CogC), number and type of structural flaws Code Smells (CS).

The generalized structure of the experimental sample is given in Table 1.

To simulate the process of multi-step automated refactoring, ten typical scenarios were formed, each of which contained a sequence of interrelated refactoring operations. The scenarios reflect typical approaches to eliminating structural flaws in the software code and were used during the experimental study. The characteristics of the generated scenarios are given in the Table. 2.

For each scenario, three program fragments were generated, which ensured an even distribution of the experimental sample between the studied multi-step refactoring scenarios. An example of the characteristics of program fragments is given in Table 3.

Table 1. **Structure of the experimental set of program fragments**

Characteristic	Value
Number of program fragments	30
Programming language	Java
Number of multi-step refactoring scenarios	10
Number of operations in the scenario	3-5
Basic refactoring techniques	Extract Method, Inline Method, Rename Method, Rename Variable, Move Method, Extract Class, Remove Dead Code, Replace Conditional with Polymorphism, Simplify Conditional Expressions, Introduce Parameter Object
LOCrange	25-210
CCrange	3-28
CogCrange	4-42
Number of CS	1-4
CS types	Long Method, Large Class, Duplicate Code, Feature Envy, Long Parameter List, Data Class, Dead Code, Complex Conditional, Poor Naming, Lazy Class, Switch Statements

***Source*: compiled by the authors**

Table 2. **Multi-step refactoring scenarios**

Script code	Sequence of refactoring operations	The main structural flaw
S1	Rename Variable → Extract Method → Remove Dead Code	Long Method
S2	Extract Method → Simplify Conditional Expressions → Rename Method	Complex Conditional
S3	Move Method → Extract Class → Rename Variable	Feature Envy
S4	Simplify Conditional Expressions → Replace Conditional with Polymorphism → Move Method	Switch Statements
S5	Introduce Parameter Object → Rename Method → Extract Method	Long Parameter List
S6	Rename Variable → Extract Method → Move Method → Remove Dead Code	Long Method, Dead Code
S7	Inline Method → Rename Method → Remove Dead Code	Lazy Class
S8	Extract Class → Move Method → Rename Variable	Large Class
S9	Extract Method → Simplify Conditional Expressions → Remove Dead Code	Long Method, Complex Conditional
S10	Extract Method → Move Method → Extract Class → Remove Dead Code	Large Class, Feature Envy

***Source*: compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

*Table 3.***The example of the characteristics of program fragments of an experimental sample**

ID	Script code	LOC	CC	CogC	Code Smells
F1	S1	84	16	24	Long Method
F2	S2	47	7	10	Complex Conditional
F3	S3	96	14	21	Feature Envy
F4	S8	168	22	36	Large Class
F5	S7	31	4	5	Lazy Class
…	…	…	…	…	…

***Source*: compiled by the authors**

Each program fragment was processed according to a defined multi-step refactoring scenario and a single experiment methodology.

Experiment methodology

The experimental study was conducted using a single methodology for all program fragments, which ensured comparability of the results obtained and minimized the influence of extraneous factors. The independent variables of the study were the characteristics of the program code, while the language model, prompt template, and evaluation procedure remained unchanged for all experiments.

The general sequence of conducting the experimental study is shown in Fig. 1.

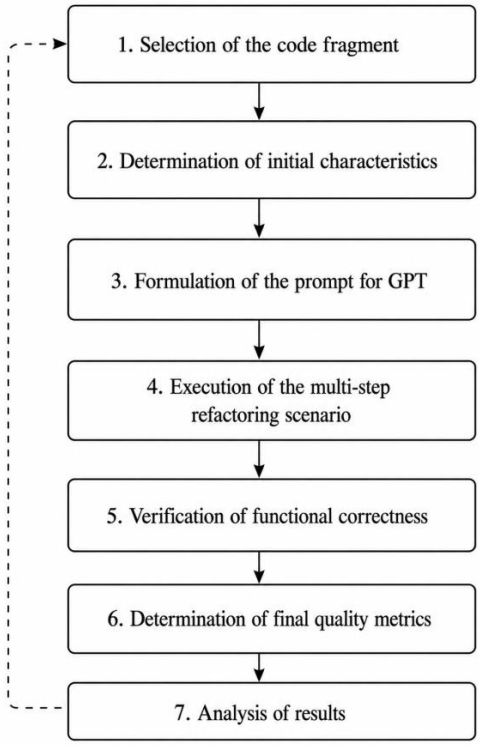


Fig. 1. **General scheme of conducting the experiment**

***Source*: compiled by the authors**

The presented sequence was applied to each software fragment without changing the model, prompt template, and verification procedure, which ensured comparability of the obtained results.

The research methodology involved the sequential implementation of the following stages.

1. Selection of the code fragment. For each experiment, the code fragment with a defined multi-step refactoring scenario and known structural characteristics was selected.

2. Determination of initial characteristics. Before performing refactoring, the values of the LOC, Cyclomatic Complexity, Cognitive Complexity, Maintainability Index indicators, as well as the type and number of detected Code Smells, were determined for each program fragment.

3. Formulation of the prompt for GPT. For all experiments, the same request template was used, which contained the program code, a description of the multi-step refactoring scenario, and instructions for preserving the functional behavior of the program.

Using a single template allowed minimizing the impact of differences in the formulation of instructions on the results of the experiment.

The GPT-5.5 model was used for the experimental study via the ChatGPT web interface. All experiments were performed using the same model and the same query template: “Perform a multi-step refactoring of the given Java code according to the following scenario: [scenario description]. Do not change the functional behavior of the program. Consistently perform all the specified refactoring operations. Preserve public interfaces and business logic. Return the complete refactored code without additional explanations.”

The scenario description was formed according to a single template that contained a sequence of refactoring operations, for example: “Rename Variable → Extract Method → Remove Dead Code”. For all experiments, only the scenarios listed in Table 2 were used.

For each piece of software, a single version of the refactored code was generated without repeated requests or clarification of instructions. This ensured uniform experimental conditions and minimized the impact of different interaction strategies with the model.

4. Execution of the multi-step refactoring scenario. The program code was passed to the GPT model along with a description of the corresponding multi-step refactoring scenario. The model sequentially performed the specified refactoring

operations, after which a refactored version of the program fragment was formed.

5. Verification of functional correctness. The resulting program code was compiled and tested using unit tests. If the compilation was successful and all unit tests passed, the program fragment was considered functionally equivalent to the original version.

6. Determination of final quality metrics. After the execution of the multi-step refactoring scenario for the refactored software code, all the studied characteristics were determined again and compared with their initial values.

7. Analyzing the results. At the final stage, the characteristics of the software code were compared before and after the execution of the multi-step refactoring scenario, the success of the scenario execution and the influence of the characteristics of the software code on the obtained results were determined.

Criteria for evaluating refactoring results

The effectiveness of multi-step automated refactoring was assessed using a combination of functional and structural criteria. This approach allows us to take into account not only the fact of executing the refactoring script, but also its impact on the quality of the program code.

The main criterion is functional equivalence, which determines whether the behavior of the program is preserved after refactoring. Verification was carried out by compiling the program code and running unit tests. A multi-step refactoring scenario was considered functionally correct if the program code compiled successfully and all unit tests passed.

The second criterion is the successful execution of the multi-step refactoring scenario, which characterizes the compliance of the obtained result with the task. It was evaluated whether the model performed all the refactoring operations provided by the corresponding scenario without disturbing the functional behavior of the program code.

To quantify the results of multi-step automated refactoring, the metrics MI, CC, CogC, and LOC, as well as the number of eliminated Code Smells, were used. The indicators MI, CC, CogC, and LOC were determined before and after the refactoring, and then their changes were analyzed. The effectiveness of eliminating structural defects was assessed by the change in the number of CS before and after the automated refactoring.

Table 4 was used to systematize the evaluation criteria.

Table 4. **Criteria for evaluating the results of automated refactoring**

Criterion	Indicator	Desired result
Functional equivalence	Compiling and passing tests	Successfully
Success of the refactoring scenario	Performing all operations of the refactoring script	Done
Maintainability Index	MI	Increase
Cyclomatic Complexity	CC	Decrease
Cognitive Complexity	CogC	Decrease
Lines of Code	LOC	Decrease or no significant change*
Code Smells	Number of structural defects	Decrease

***Source*: compiled by the authors**

*For some multi-step refactoring scenarios that involve Extract Method or Extract Class operations, the number of lines of code may increase, but this does not indicate a deterioration in code quality. Therefore, LOC is only analyzed in conjunction with other metrics

The combined use of functional and structural criteria allowed us to separate the assessment of the functional correctness of the refactoring scenario from the assessment of changes in the quality indicators of the program code.

EXPERIMENTAL RESEARCH RESULTS

Overall success rate of multi-step refactoring scenarios

The experimental study was conducted using Java 21 in the IntelliJ IDEA 2025.1 environment. Automated refactoring was performed using the GPT-5.5 model via the ChatGPT web interface. The program code was compiled using OpenJDK 21, functional equivalence was checked using JUnit 5, and MI, CC, CogC, and Code Smells were determined using SonarQube 10.8. The same software tools, query template, and evaluation methodology were used for all program fragments, which ensured the reproducibility of the experiment and the correctness of the comparison of the obtained results.

At the first stage of the experimental study, the overall success of the automated execution of multi-step refactoring scenarios using the GPT-5.5 model was evaluated. For each program fragment, the functional correctness of the obtained result, the

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

completeness of the execution of all operations provided for by the scenario, as well as the compliance of the refactored code with the specified requirements were analyzed.

Functional correctness was determined by the results of compiling the program code and executing unit tests. A script was considered successfully executed if the functional behavior of the program code was preserved, all tests were successfully passed, and all refactoring operations provided for by the corresponding script were performed.

The general results of executing multi-step refactoring scenarios are given in Table 5.

Table 5. **General results of running multi-step refactoring scenarios**

Indicator	Value
Total number of program fragments	30
Total number of executed scenarios	30
Successfully executed scenarios	26
Error scenarios	4
Successful compilation	28 (93.3 %)
Successful completion of module tests	26 (86.7 %)
Functionally correct results	26 (86.7 %)

***Source*: compiled by the authors**

As seen in Table 5, the GPT-5.5 model successfully performed 26 out of 30 multi-step refactoring scenarios, which is 86.7 % of the total number of experiments. In 28 cases (93.3 %), the resulting program code compiled successfully, while in two cases syntax errors were detected. According to the results of unit tests, the functional equivalence of the original and refactored program code was confirmed for 26 program fragments. Four scenarios were classified as unsuccessful due to violation of the functional behavior of the program or incomplete execution of the refactoring operations provided by the scenario.

The impact of multi-step refactoring on code quality metrics

After confirming the functional correctness of the results, an analysis of the impact of multi-step automated refactoring on the quality indicators of the program code was performed. For each program fragment, the values of the metrics MI, CC, CogC, LOC and the number of Code Smells were determined before and after the refactoring script was executed.

To assess the overall effectiveness of the proposed approach, the average values of each metric before and after refactoring were determined, and the absolute and relative changes in the corresponding indicators were calculated. The results are presented in Table 6.

As seen in Table 6, the implementation of multi-step refactoring scenarios had a positive effect on all the studied indicators of the quality of the program code. For the Maintainability Index indicator, an increase in the value of which indicates an improvement in the quality of the code, the average increase was 17.4 %. For the remaining indicators, on the contrary, the improvement was characterized by a decrease in their values. In particular, the average number of Code Smells decreased by 65.2 %, Cyclomatic Complexity – by 26.3 %, Cognitive Complexity – by 28.6 %, and the average number of lines of program code – by 3.9 %.

Table 6. **Change in code quality metrics after multi-step refactoring**

Indicator	Before refactoring	After refactoring	Average change	Improvement, %
MI	63.8	74.9	+11.1	+17.4
CC	11.8	8.7	−3.1	−26,3
CogC	18.9	13.5	−5.4	−28.6
LOC	102	98	−4	−3.9
CS	2.3	0.8	−1.5	−65.2

***Source*: compiled by the authors**

The decrease in the values of Cyclomatic Complexity and Cognitive Complexity indicates a simplification of the logical structure of the program code and a decrease in the complexity of its analysis. The slight decrease in the Lines of Code indicator is consistent with the specifics of individual refactoring scenarios, since the execution of Extract Method or Extract Class operations may be accompanied by a local increase in the volume of the program code without a deterioration in its quality.

Overall, the results obtained indicate that the use of multi-step automated refactoring scenarios using the GPT-5.5 model improves key indicators of software code quality while maintaining its functional correctness.

The impact of code characteristics on refactoring success

To determine the factors affecting the success of automated multi-step refactoring, the dependence of the results of the scenarios on the initial characteristics of the software code was analyzed. The analysis was carried out according to LOC, CC, CogC and the number of CS.

For each program fragment, a correspondence was established between the initial values of the specified indicators and the result of the refactoring scenario execution (successful or unsuccessful execution). The summarized results are given in Table 7.

Table 7. **Characteristics of the program code for successful and unsuccessful scenarios**

Indicator	Successful scenarios (26)	Failed scenarios (4)
LOC	94	151
CC	9.4	17.3
CogC	14.6	26.1
CS	1.8	4.3

***Source*: compiled by the authors**

As seen in Table 7, unsuccessful execution of refactoring scenarios was mainly observed for program fragments with a larger amount of code, higher cyclomatic and cognitive complexity, as well as a larger number of structural flaws. The largest differences between the groups are observed for the Cognitive Complexity and Code Smells indicators, which indicates their significant impact on the success of automated multi-step refactoring.

The obtained results also demonstrate a tendency to decrease the success rate of script execution with increasing complexity of the program code. For program fragments with Cyclomatic Complexity over 15, Cognitive Complexity over 25, or the presence of more than four Code Smells, the frequency of unsuccessful script execution increased significantly. Meanwhile, for program fragments with moderate values of the indicated indicators, the GPT-5.5 model ensured stable and functionally correct execution of refactoring scripts. The results of the experimental study indicate that the initial characteristics of the program code are one of the key factors determining the effectiveness of automated multi-step refactoring. With an increase in the volume and complexity of the program code, as well as the number of structural flaws, the probability of successful execution of the refactoring script decreases.

For a more detailed analysis of the influence of the initial characteristics of the software code on the success of the automated multi-step refactoring, graphical dependency charts were built comparing the key metrics of code quality and the success of the refactoring scenarios. Fig. 2 shows the results of the analysis for CC, CogC, number of CS and LOC.

As the cyclomatic complexity of the program code increases, the success rate of multi-step refactoring scenarios gradually decreases. The highest success rates are observed for program fragments with a low level of Cyclomatic Complexity, while for more complex code, the probability of successful execution of the scenarios decreases significantly.

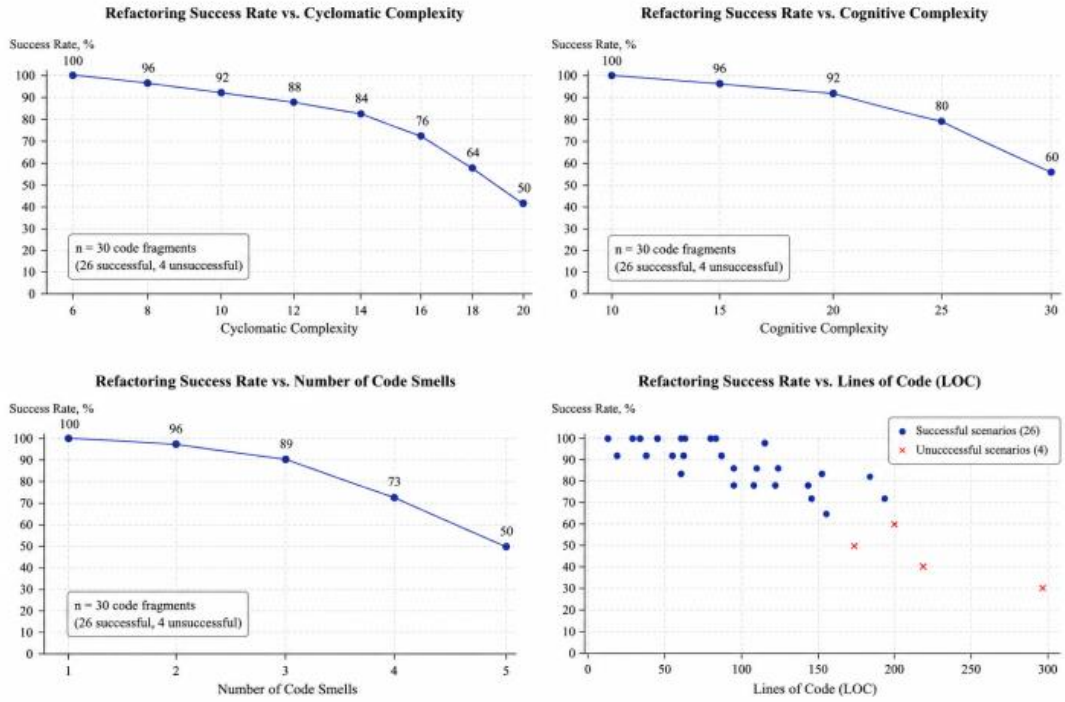


Fig. 2. **The dependence of the success of multi-step refactoring on the initial characteristics of the program code**

***Source*: compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

A similar trend is observed for the Cognitive Complexity indicator. As the cognitive complexity of the program code increases, the success of automated refactoring decreases, which indicates the difficulty of correctly converting program fragments with a more complicated logical structure.

Increasing the number of Code Smells also negatively affects the success rate of refactoring scenarios. For program fragments with a small number of structural flaws, the GPT-5.5 model demonstrates a high level of success, while as the number of Code Smells increases, the proportion of successfully executed scenarios gradually decreases.

In addition, the distribution of successful and unsuccessful scenarios depends on the size of the program code (LOC). Unsuccessful scenarios are mainly observed for larger program fragments, which indicates a decrease in the effectiveness of automated multi-step refactoring with increasing code size.

Overall, the results shown in Fig. 2 indicate that the initial characteristics of the program code are significantly related to the success of automated multi-step refactoring. As cyclomatic and cognitive complexity, the number of Code Smells, and the volume of the program code increase, the probability of successful execution of refactoring scenarios decreases. The most pronounced impact on the success of scenario execution is observed for the indicators CC, CogC, and the number of CS.

Analysis of the effectiveness of various multi-step refactoring scenarios

To determine the most effective sequences of automated refactoring, a comparative analysis of ten multi-step refactoring scenarios (S1–S10) was conducted. For each scenario, the success of its execution, the average improvement in source code quality indicators, and the most typical problems that arose during execution were evaluated. The summarized results are given in Table 8.

As shown in Table 8, the highest efficiency was demonstrated by scenarios S1–S6, which were successfully executed in all experiments. At the same time, scenarios S7–S10 were characterized by lower success (2 out of 3 executions), which was associated with more complex multi-step transformations, in particular, transferring methods, eliminating code duplication, and performing interconnected refactoring operations. Analysis of successfully executed scenarios showed that even in cases where there was a need to check individual transformations or minor manual refinement, most scenarios provided an average or high improvement in the quality indicators of the program code. This indicates the potential of using large language models to automate multi-step refactoring, provided that the results of executing more complex scenarios are controlled.

Table 8. **Comparison of multi-step refactoring scenarios**

Sce na-rio	Success-fully / total	Level of improvement in quality indicators*	The most common performance features
S1	3/3	High	–
S2	3/3	High	–
S3	3/3	High	The need to verify individual transactions
S4	3/3	Average	The need to clarify the names of individual elements
S5	3/3	Average	Needs minor manual editing
S6	3/3	High	Minor structural deviations
S7	2/3	Average	Incomplete execution of the script
S8	2/3	High	Partial preservation of duplicate code
S9	2/3	Average	Incorrect transfer of methods
S10	2/3	Moderate	Errors when performing complex multi-step transformations

***Source*: compiled by the authors**

*The average improvement in quality metrics is only given for experiments where the scenario was successfully executed. The score takes into account the combined change in MI, CC, CogC, LOC, and CS metrics

Common mistakes during automated multi-step refactoring

To identify the main limitations of GPT-5.5, we analyzed all cases of unsuccessful execution of multi-step refactoring scenarios. The analysis showed that most errors occurred during execution of scenarios that contained a large number of interconnected operations or required significant restructuring of the program code structure.

The most common scenario was incomplete execution of the refactoring script. In such cases, the model successfully performed only the initial operations, after which it completed the refactoring without implementing all the transformations provided by the script.

Another common type of error was skipping individual refactoring operations. For example, after successfully executing the Extract Method and Rename Variable operations, the model might not execute the final Remove Dead Code, even though this operation was required by the script.

In some experiments, a violation of the scenario logic was observed. Instead of performing operations in a specific sequence, the model changed their order or replaced individual transformations with alternative ones, which led to a deviation from the specified multi-step refactoring scenario.

Some results were characterized by an increase in the complexity of the program code. Despite the successful execution of most operations, in some cases the values of Cyclomatic Complexity or Cognitive Complexity increased after refactoring, which indicated a suboptimal structuring of individual sections of the program code.

Also, in rare cases, new Code Smells were observed after refactoring. For example, after transferring individual methods between classes, there were code duplications or excessively long methods that were not present in the original version of the program.

The least common but most critical error was functional equivalence violations. These cases manifested themselves in the form of compilation errors or unit test failures after refactoring, indicating a change in the functional behavior of the code.

Below are some typical examples of such errors.

Example 1. Incomplete script execution. For a script that involved the sequence Extract Method → Move Method → Extract Class → Remove Dead Code, the model only executed the first two operations, not completing the full refactoring script.

Example 2. Increasing the complexity of the program code. After refactoring, the Cognitive Complexity score increased due to the appearance of additional nested conditional constructs, although the functional behavior of the program remained unchanged.

Example 3. Violation of functional equivalence. When moving a method to another class, the model incorrectly updated its calls, which led to a compilation error and failed unit tests.

Example 4. Skipping a single refactoring operation. After performing the Rename Method and Extract Method operations, the model did not apply the planned Inline Variable, resulting in unnecessary intermediate variables in the code, and the script was only partially executed.

Example 5. A new Code Smell appears. During the execution of a script that included Move Method and Extract Class, the model moved some of the functionality to a new class, but left duplicate code fragments in both classes, which led to the appearance of a new Code Smell (Duplicate Code).

The results obtained indicate that the main errors in automated multi-step refactoring are primarily associated with the execution of long sequences of interdependent operations. However, their number was insignificant compared to the total number of successfully executed scenarios, which confirms the practical suitability of GPT-5.5 for automated multi-step refactoring of software code.

CONCLUSIONS

The paper presents an experimental study of the effectiveness of using the GPT-5.5 model for automated multi-step refactoring of Java code. Unlike most existing studies, the evaluation was performed not for individual refactoring operations, but for complete scenarios consisting of three to five interconnected transformations.

Experimental results showed that GPT-5.5 successfully completed 26 out of 30 multi-step refactoring scenarios (86.7 %), ensuring successful compilation of programs in 93.3 % of cases. The performed transformations contributed to the improvement of the main indicators of source code quality: an increase in the Maintainability Index, a decrease in Cyclomatic Complexity, Cognitive Complexity, the number of Code Smells, and a slight reduction in the volume of program code.

Analysis of the obtained results showed that the effectiveness of automated multi-step refactoring largely depends on the initial characteristics of the software code. With increasing cyclomatic and cognitive complexity, the number of structural flaws and the amount of code, the probability of successful execution of scenarios gradually decreases. In addition, as the number of interdependent operations in the script increases, the risk of errors during refactoring increases.

The analysis also allowed us to identify the most typical errors of automated multi-step refactoring, including incomplete script execution, skipping of individual operations, violation of script execution logic, increase in program code complexity, emergence of new Code Smells, and, in rare cases, violation of functional equivalence.

The results obtained confirm the promising use of large language models for automating multi-step refactoring of software code. Promising directions for further research are expanding the set of refactoring scenarios, studying other programming languages, and comparing the effectiveness of different large language models when performing complex sequences of refactoring operations.

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

Conflicts of Interest: The author Andrii Hryshchenko declares that they have no conflicts of interest regarding this study, including financial, personal, authorship, or other interests that could influence the research and the results presented in this article. Author Nataliia Komleva is a member of the Editorial Board of this journal. This role had no influence on the peer-review process or the editorial decision regarding this manuscript.

Financing: Research was carried out without financial support

Accessibility data: All data available in digital or graphic form in the main text manuscript

Using artificial intelligence tools: The GPT-5.5 model was used as a tool for implementing the multi-step refactoring technology developed by the authors on author-created program fragments and scenarios. All obtained results were independently verified and analyzed by the authors. The scientific contributions, interpretation of the results, and conclusions are the authors' own

REFERENCES

1. Komleva, N., Liubchenko, V. & Zinovatna, S. "Evaluation of the quality of survey data and its visualization using dashboards". *Computer Science and Information Technologies*. 2020; 2: 234–237, https://www.scopus.com/pages/publications/85100509851?origin=resultslist.
DOI: https://doi.org/10.1109/CSIT49958.2020.9321970.

2. Fowler, M. "Refactoring: improving the design of existing code". 2nd ed. *Addison-Wesley Professional*. Boston. 2018. ISBN: 978-0-134-75759-9.

3. Feng, Z., Guo, D., Tang, D. et al. "CodeBERT: a pre-trained model for programming and natural languages". *Findings of the Association for Computational Linguistics: EMNLP*. 2020. p. 1536–1547. DOI: https://doi.org/10.18653/v1/2020.findings-emnlp.139.

4. Komleva, N. O. & Tereshchenko, O. I. "Requirements for the development of smart contracts and an overview of smart contract vulnerabilities at the solidity code level on the ethereum platform". *Herald of Advanced Information Technology*. 2023; 6 (1): 54–68, https://www.scopus.com/pages/publications/105024774594?origin=resultslist. DOI: https://doi.org/10.15276/hait.06.2023.4.

5. Wang, Y., Wang, W., Joty, S. & Hoi, S. C. H. "CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation". *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 2021. p. 8696–8708, https://www.scopus.com/pages/publications/85125780341?origin=resultslist. DOI: https://doi.org/10.18653/v1/2021.emnlp-main.685.

6. Chen, M., Tworek, J., Jun, H., et al. "Evaluating large language models trained on code". *arXiv*. 2021. DOI: https://doi.org/10.48550/arXiv.2107.03374.

7. Fan, A., Gokkaya, B., Harman, M. et al. "Large language models for software engineering: survey and open problems". *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering*. 2023. p. 31–53, https://www.scopus.com/pages/publications/85185604518?origin=resultslist. DOI: https://doi.org/10.1109/ICSE-FoSE59343.2023.00008.

8. Hou, X., Zhao, Y., Liu, Y. et al. "Large language models for software engineering: a systematic literature review". *ACM Transactions on Software Engineering and Methodology*. 2024; 33 (8): 1–79, https://www.scopus.com/pages/publications/85212610342?origin=resultslist.
DOI: https://doi.org/10.1145/3695988.

9. Nyirongo, B., Jiang, Y., Jiang, H. & Liu, H. "A survey of deep learning based software refactoring". *arXiv*. 2024. DOI: https://doi.org/10.48550/arXiv.2404.19226.

10. Shirafuji, A., Oda, Y., Suzuki, J., Morishita, M. & Watanobe, Y. "Refactoring programs using large language models with few-shot examples". *30th Asia-Pacific Software Engineering Conference*. 2023. p. 151–160, https://www.scopus.com/pages/publications/85190537096?origin=resultslist.
DOI: https://doi.org/10.1109/APSEC60848.2023.00025.

11. DePalma, K. et al. "Exploring ChatGPT's code refactoring capabilities: an empirical study". *Expert Systems with Applications*. 2024; 249: 123602, https://www.scopus.com/pages/publications/85188169544?origin=resultslist. DOI: https://doi.org/10.1016/j.eswa.2024.123602.

12. AlOmar, E. A., Venkatakrishnan, A., Mkaouer, M. W., Newman, C. D. & Ouni, A. "How to refactor this code? An exploratory study on developer–ChatGPT refactoring conversations". *Proceedings of the 21st International Conference on Mining Software Repositories*. 2024. p. 202–206, https://www.scopus.com/pages/publications/85197428680?origin=resultslist.
DOI: https://doi.org/10.1145/3643991.3645081.

13. Soham, D., et al. "Analyzing developer–ChatGPT conversations for software refactoring: an exploratory study". *Proceedings of the 21st International Conference on Mining Software Repositories*. 2024: 207–211, https://www.scopus.com/pages/publications/85197381511?origin=resultslist. DOI: https://doi.org/10.1145/3643991.3645082.

14. Choi, J., An, G. & Yoo, S."Iterative refactoring of real-world open-source programs with large language models". *Search-Based Software Engineering. Lecture Notes in Computer Science*. 2024; 14767: 49–55, https://www.scopus.com/pages/publications/85200245650?origin=resultslist.
DOI: https://doi.org/10.1007/978-3-031-64573-0_4.

15. Liu, B., Jiang, Y., Zhang, Y., Niu, N., Li, G. & Liu, H. "Exploring the potential of general-purpose llms in automated software refactoring: an empirical study". *Automated Software Engineering*. 2025; 32 (1): 26, https://www.scopus.com/pages/publications/105000010073?origin=resultslist.
DOI: https://doi.org/10.1007/s10515-025-00500-0.

16. Cordeiro, J., Noei, S. & Zou, Y. "An empirical study on the code refactoring capability of large language models". *arXiv*. 2024. DOI: https://doi.org/10.48550/arXiv.2411.02320.

17. Hryshchenko, A. I. & Komleva, N. O. "Game-theoretic method of decentralized load balancing in microservice architectures". *Herald of Advanced Information Technology*. 2025; 8 (4): 488–496, https://www.scopus.com/pages/publications/105026618516?origin=resultslist.
DOI: https://doi.org/10.15276/hait.08.2025.31.

18. Silva, L. L., Silva, J. R., Montandon, J. E., Andrade, M. & Valente, M. T. "Detecting code smells using ChatGPT: initial insights". *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2024. p. 400–406, https://www.scopus.com/pages/publications/85210594641?origin=resultslist. DOI: https://doi.org/10.1145/3674805.3690742.

19. Wu, D., Mu, F., Shi, L. et al. "iSMELL: assembling LLMs with expert toolsets for code smell detection and refactoring". *Proceedings of the 39th ACM/IEEE International Conference on Automated Software Engineering*. 2024. p. 1345–1357, https://www.scopus.com/pages/publications/85212444025?origin=resultslist. DOI: https://doi.org/10.1145/3691620.3695508.

20. Pandini, G., Martini, A., Videsjorden, A. N. & Arcelli Fontana, F. "An exploratory study on architectural smell refactoring using large languages models". *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*. 2025. p. 462–471. https://www.scopus.com/pages/publications/105007919273?origin=resultslist. DOI: https://doi.org/10.1109/ICSA-C65153.2025.00070.

21. Bellur, A., Batole, F., Ullah, M., Dilhara, M., Zharov, Y., Bryksin, T., Ishikawa, K., Chen, H., Morimoto, M., Hosomi, T., Nguyen, T., Rajan, H. & Tsantalis, N. "Together we are better: LLM, IDE and semantic embedding to assist move method refactoring". *IEEE International Conference on Software Maintenance and Evolution*. 2025. p. 1–13, https://www.scopus.com/pages/publications/105022449590?origin=resultslist. DOI: https://doi.org/10.1109/ICSME64153.2025.00046.

22. Santana Jr., E. G., Santos Junior, J. P., Almeida, E. P., Ahmed, I., Silveira Neto, P. A. M. & Almeida, E. S. "Evaluating LLMs effectiveness in detecting and correcting test smells: an empirical study". *arXiv*. 2025. DOI: https://doi.org/10.48550/arXiv.2506.07594.

23. Hryshchenko, A. "A Method for Requirements Engineering and Web System Design Using Core Web Vitals Metrics". Transactions of Kremenchuk Mykhailo Ostrohradskyi National University. 2025; 6 (155): 156–164. DOI: https://doi.org/10.32782/1995-0519.2025.6.20.

24. Chand, S., Kilic, M., Würsching, R., Kumar Pandey, S. & Pretschner, A. "Automated extract method refactoring with open-source LLMs: a comparative study". *IEEE/ACM International Conference on AI-powered Software*. 2025. p. 113–122, https://www.scopus.com/pages/publications/105035215735?origin=resultslist. DOI: https://doi.org/10.1109/AIware69974.2025.00020.

25. Pandiyavathi, T. & Sivakumar, B. "Ensemble deep network for secured refactoring framework by predicting code-bad smells in software projects". *Journal of Software: Evolution and Process*. 2025; 37 (2): e2749, https://www.scopus.com/pages/publications/85212072694?origin=resultslist.
DOI: https://doi.org/10.1002/smr.2749.

26. Martinez, S., Xu, L., Elnaggar, M. & Alomar, E. A. "Software Refactoring Research with Large Language Models: A Systematic Literature Review". *Journal of Systems and Software*. 2025; 223: 112762, https://www.scopus.com/pages/publications/105027029724?origin=resultslist.
DOI: https://doi.org/10.1016/j.jss.2025.112762.

Received 25.08.2026
Received after revision 20.10.2026
Accepted 27.10.2026

DOI: https://doi.org/10.15276/hait.09.2026.37
УДК 004.43:004.89

Оцінювання ефективності багатокрокового автоматизованого рефакторингу програмного коду з використанням великої мовної моделі

Грищенко Андрій Ігорович[1)]
ORCID: https://orcid.org/0009-0007-1191-7948; andrew.hryshchenko@gmail.com. Scopus Author ID: 60278347900
Комлева Наталія Олегівна[2)]
ORCID: https://orcid.org/0000-0001-9627-8530; komleva@op.edu.ua. Scopus Author ID: 57191858904
[1)] Компанія Lowe's, Inc., м. Мурсвілл, штат Північна Кароліна, США
[2)] Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, Україна, 65044

АНОТАЦІЯ

Актуальність: сучасні великі мовні моделі відкривають нові можливості для автоматизації рефакторингу програмного коду; однак більшість існуючих досліджень зосереджена на виконанні окремих операцій рефакторингу, тоді як автоматизоване виконання послідовностей таких операцій залишається недостатньо дослідженим. **Метою статті** є дослідження ефективності використання великої мовної моделі для автоматизованого багатокрокового рефакторингу програмного коду мовою Java. **Завдання:** розробити сценарії багатокрокового рефакторингу, провести експериментальне оцінювання їх виконання великою мовною моделлю, проаналізувати вплив початкових характеристик програмного коду на успішність рефакторингу, порівняти ефективність різних сценаріїв та визначити типові помилки, що виникають під час виконання багатокрокових перетворень. **Методи:** ефективність оцінювалася за показниками Maintainability Index, Cyclomatic Complexity, Cognitive Complexity, Lines of Code, кількістю Code Smells, а також за успішністю компіляції та виконання модульних тестів. **Наукова новизна:** полягає в оцінюванні ефективності великої мовної моделі для виконання багатокрокового рефакторингу, що передбачає послідовність взаємопов'язаних операцій, а також у дослідженні впливу характеристик програмного коду та складності сценаріїв на успішність їх автоматизованого виконання. **Практична значимість:** отримані результати можуть бути використані під час розроблення інтелектуальних засобів автоматизованого рефакторингу, інтеграції великих мовних моделей у середовища розроблення програмного забезпечення та вдосконалення процесів забезпечення якості програмного коду. **Результати:** експериментальне дослідження показало, що велика мовна модель успішно виконувала сценарії багатокрокового рефакторингу, забезпечуючи покращення основних показників якості програмного коду. **Висновки:** встановлено, що ефективність автоматизованого рефакторингу залежить як від початкових характеристик програмного коду, так і від складності сценарію рефакторингу, а основними обмеженнями залишаються виконання довгих послідовностей взаємопов'язаних операцій та забезпечення повної функціональної еквівалентності програмного коду.

Ключові слова: багатокроковий рефакторинг; рефакторинг програмного коду; великі мовні моделі; Java; якість програмного коду; автоматизація програмування; штучний інтелект

ABOUT THE AUTHORS



Andrii I. Hryshchenko - Senior Software Engineer at Lowe's Companies, Inc.,1000 Lowe's Blvd., Mooresville, NC 28117, USA
ORCID: https://orcid.org/0009-0007-1191-7948; andrew.hryshchenko@gmail.com. Scopus Author ID: 60278347900
***Research field*:** web performance optimization; distributed systems; microservice architecture; edge computing; reliability engineering

Грищенко Андрій Ігорович - провідний інженер-програміст компанії Lowe's Companies, Inc. 1000 бульвар Лоуз, м. Мурсвілл, штат Північна Кароліна, 28117, США
Гвлузь досліджень: оптимізація веб-продуктивності; розподілені системи; мікросервісна архітектура; периферійні обчислення; інженерія надійності



Nataliia O. Komleva - PhD, Associate Prof., Head of Department of System Software. Odesa Polytechnic National University, Odesa, 65044, Ukraine
ORCID: https://orcid.org/0000-0001-9627-8530, komleva@op.edu.ua. Scopus Author ID: 57191858904
Scientific field: data analysis, software engineering, distributed computing

Комлева Наталія Олегівна - канд. техніч. наук, завідувач кафедри Інженерії програмного забезпечення. Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044, Україна
Гвлузь досліджень: аналіз даних, програмна інженерія, розподілені обчислення