www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 17(1), Jan-April 2026, 286-298. ISSN: 2007-1558.
https://doi.org/10.61467/2007.1558.2026.v17i1.1025

Ordered sequencing of Refactorings based on Preconditions and Postconditions

Nelida Barón Pérez, René Santaolaya Salgado, Blanca Dina Valenzuela Robles, Olivia Fragoso Diaz, Juan Carlos Rojas Pérez, Juan Gabriel González Serna, Adriana Mexicano Santoyo

Centro Nacional de Investigación y Desarrollo Tecnológico.

d18ce064@cenidet.tecnm.mx,rene.ss@cenidet.tecnm.mx,blanca.vr@cenidet.tecnm.mx, olivia.fd@cenidet.tecnm.mx, juan.rp@cenidet.tecnm.mx, ,gabriel.gs@cenidet.tecnm.mx, mexicanoa@gmail.com

Abstract. Refactoring is an essential practice in software development, aimed at improving the internal quality of code without modifying its external behaviour. This work introduces a modification to the Greedy algorithm through the integration of preconditions and postconditions to seek to optimise the ordering of refactoring operations. Unlike the traditional approach, which selects transformations without considering their cumulative impact, the proposed algorithm evaluates each step to help ensure progressive improvement in key metrics such as abstraction, modular protection, and implementation inheritance. To validate the effectiveness of the algorithm, it was applied to 30 applications selected from the GitHub repository, with the condition that each project remained executable both before and after refactoring. Metrics were evaluated at each stage of the process to assess the actual impact of the generated sequences.

Keywords: Software refactoring, Refactoring sequences, Sequencing code transformations, Quality assessment in refactoring.

Article Info
Received March 15, 2025
Accepted October 18, 2025

1 Introduction

In software development, refactoring has become a fundamental technique for improving the internal quality of code without altering its functionality. As systems evolve, issues such as technical debt and code smells often emerge, which affect the maintainability and scalability of applications (Fowler, 1999). If these problems are not addressed in a timely manner, they can increase maintenance costs and reduce system reliability. Therefore, refactoring is considered an essential element in software evolution, as it contributes to building more sustainable and efficient designs (Opdyke, 1992).

Although various techniques and automated tools exist to identify and apply refactoring sequences, many of these approaches do not analyze whether the quality obtained after applying one refactoring could be negatively affected by subsequent refactorings. This aspect is critical to ensuring cumulative and stable improvements during the refactoring process (Mens & Tourwé, 2004). For example, different heuristic search algorithms have shown effectiveness in identifying opportunities for improvement; however, they often omit verifying the accumulated impact that each step generates within a sequence, which may compromise the global quality of the system (Kessentini et al., 2013).

To address this limitation, the present work proposes an algorithm called the Improved Greedy Algorithm, which incorporates mechanisms based on preconditions and postconditions. Each refactoring requires a set of initial conditions that ensure its correct execution and prevent the loss of improvements obtained in previous refactorings. At the end of a refactoring, a set of output conditions is generated, which in turn becomes the input conditions for the next one. In a refactoring sequence, these conditions must be satisfied in an ordered manner to ensure the coherence of the process. Defining them allows evaluating, at each step, the compatibility with the conditions inherited from the previous refactoring, with the purpose of cumulatively

preserving the quality improvements of the system. For the reasons mentioned above, this approach considers three critical aspects of design: modular protection, lack of abstraction, and implementation inheritance.

To evaluate the impact of the proposed method, specialized metrics that analyze fundamental design attributes were employed. Specifically:

- modular protection: PMP, PMPr, PMF and PMT metrics were used, which evaluate different levels of access and encapsulation (Baron, 2023).
- implementation Inheritance and Flexibility: the Flexibility Factor (FF) and the Implementation Inheritance Factor of a class architecture (FHIAC) were applied (Ortiz, 2023).
- lack of abstraction: the Abstractness (A) metric, which measures the ratio of abstract classes to the total number of classes in a package, was used to evaluate the level of abstraction in the software design (Kaur & Sharma, 2015).

These metrics were used to compare the state of the software before and after applying the sequences generated by the Improved Greedy Algorithm, with the purpose of determining whether the executed refactorings preserved, improved, or avoided degradation in attributes such as modularity, abstraction, and implementation inheritance. The validation of the approach was carried out using 30 projects randomly selected from the GitHub repository, which allowed analyzing its effectiveness in different scenarios and evaluating its real impact on software maintainability.

The content of this article is organized as follows. Section 2 presents the related works. Section 3 describes the design and methodology used to develop the Improved Greedy Algorithm, detailing the preconditions and postconditions that distinguish it from traditional approaches. Section 4 introduces the mathematical formalization of the algorithm. Section 5 describes the validation process, followed by the results and discussion in Section 6. Finally, Section 7 presents the general conclusions and possible future research directions.

2 Related Works

In the field of software refactoring, several studies have addressed the identification and construction of refactoring sequences with the purpose of improving the internal quality of design. However, most of these works do not incorporate mechanisms to verify whether each refactoring preserves or deteriorates the improvements previously obtained, which constitutes the core of the approach presented in this study based on the Improved Greedy Algorithm.

Kessentini et al. (2013) proposed an approach based on genetic algorithms to optimize cohesion and coupling. Although their method efficiently explores large search spaces, it does not guarantee compatibility between consecutive refactorings, which may lead to unexpected degradations in quality attributes. In contrast, the proposed approach integrates preconditions and postconditions to ensure that each refactoring preserves the quality previously achieved.

Similarly, Ouni et al. (2017) developed machine learning techniques to predict refactoring sequences based on historical data. While their results are promising, these methods rely on large amounts of labeled information and do not explicitly evaluate the cumulative impact between consecutive steps. In contrast, the approach presented here evaluates each refactoring incrementally, preventing contradictions between transformations.

Kurbatova et al. (2020) proposed a machine learning–based approach to recommend *Move Method* refactorings using a semantic code representation (*code2vec*). Although their method outperforms tools such as JDeodorant and JMove, the evaluation focuses on individual refactorings and global quality improvements, without analyzing the cumulative impact of applying refactoring sequences. In contrast, the Improved Greedy Algorithm evaluates each step through preconditions and postconditions, ensuring progressive quality preservation.

Meananeatra (2012) analyzed the identification of refactoring sequences aimed at improving software maintainability, evaluating such sequences based on global criteria such as the number of removed bad smells and aggregated maintainability metrics. Although the proposed approach considers the ordering of refactorings, quality assessment is performed only at the level of the complete sequence, without quantitatively validating the impact of each intermediate refactoring step. The approach proposed in this work overcomes this limitation by verifying, metric by metric, the post-refactoring effect of each step through formal preconditions and postconditions, thus ensuring progressive improvements and preventing intermediate quality degradation.

Finally, Tarwani and Chug (2021) explored the use of heuristic algorithms such as Hill-Climbing and its variants to generate refactoring sequences. Although they evaluated maintainability metrics, their approach does not consider the risk that a subsequent refactoring may degrade previously obtained quality, which affects the reliability of the sequence in complex systems. In contrast, this work guarantees the preservation of design attributes through cumulative verification based on structural metrics.

In summary, although previous studies have significantly contributed to the identification and optimization of refactoring sequences, none integrates a systematic mechanism to validate that each refactoring preserves the quality already achieved. The approach based on the Improved Greedy Algorithm addresses this limitation through incremental and formal evaluation using preconditions and postconditions. Table 1 summarizes and compares the main characteristics, limitations, and differences between the most relevant approaches and the proposal presented in this work.

Table 1. Comparison of related works and contribution of the proposed approach

Author	Main technique	Validation between steps	Main limitation	Differential contribution of the proposed approach
Kessentini et al. (2013)	Genetic algorithms	No	May generate inconsistent sequences	Integrates pre/postconditions to ensure compatibility
Ouni et al. (2017)	Machine learning	No	Requires large datasets; no cumulative control	Evaluates each step using quantitative metrics
Kurbatova et al. (2020)	Machine learning (code2vec, SVM)	No	Evaluates only individual refactorings and global quality	Step level validation using preconditions and postconditions
Meananeatra (2012)	Refactoring sequence analysis	No	Quality assessment performed only at sequence level	Evaluation of each refactoring step using individual quality metrics
Tarwani & Chug (2021)	Heuristic algorithms	No	Later refactorings may degrade earlier improvements	Cumulative metric control across the sequence
This work	**Improved Greedy Algorithm**	**Yes**	—	**Preserves and verifies quality step by step**

From Table 1, it can be observed in the third column that only the work of Mahouachi and Kessentini (2014) partially evaluates the loss of quality in successive refactorings. The remaining related studies, by not considering this aspect, cannot ensure that cumulative degradation of the quality of legacy code does not occur. In contrast, the approach proposed in this work explicitly evaluates the initial conditions that must be satisfied before each refactoring to guarantee that such degradation does not take place.

3 Greedy algorithm modification design and methodology

The traditional Greedy algorithm makes decisions by selecting, at each iteration, the locally most promising option, relying solely on immediate criteria without considering the cumulative impact of previous decisions. Although this approach is efficient and simple, it may lead to refactoring sequences that, while appearing optimal at the end of the process, include intermediate steps that degrade the structural quality of the software before improving it again.

Figure 1 presents the pseudocode of the traditional Greedy algorithm, where each refactoring is evaluated independently and only the final combinations are considered, without analyzing whether quality decreases during the transformation process.

```
start
while there are pending tasks do
   select the best option according to the current criterion
   apply the selected option
end while
end
```

Fig. 1. Traditional Greedy Algorithm (Pseudocode).

To overcome these limitations, this work proposes a modified version of the Greedy algorithm that incorporates an explicit mechanism of preconditions and postconditions to control quality throughout the entire process. In contrast to the traditional approach, the Improved Greedy Algorithm does not generate or evaluate all possible permutations, nor does it select the final sequence beforehand. Instead, it constructs the sequence incrementally, applying only those refactorings that:

- Satisfy their applicability preconditions.
- Do not cause degradation in any of the quality metrics previously achieved.

This mechanism ensures that each transformation maintains or improves the accumulated quality, preventing structural regressions and guaranteeing a safer and more stable refactoring process.

Figure 2 shows the pseudocode of the improved algorithm, highlighting at each iteration the evaluation of preconditions and postconditions to determine whether a refactoring can be included in the final sequence.

```
Start
preconditions = define_preconditions()
postconditions = define_postconditions()
order_determination =
calculate_refactoring_order(preconditions, postconditions)
while there are pending tasks do
   for each option in the determined sequence do
      if it meets the preconditions then
         apply the provisional option
         if it meets the postconditions then
            confirm the application of the option
         else
            discard the provisional option
         end if
      end if
   end for
end while
End
```

Fig 2. Improved Greedy Algorithm with Preconditions and Postconditions (Pseudocode).

Key elements of the improved algorithm

- Preconditions: They establish the criteria that must be met before applying a refactoring, ensuring that only feasible and beneficial transformations are executed in the current state of the code.
- Postconditions: They evaluate the effects of each refactoring after its application, verifying that previous improvements are not compromised. If a refactoring decreases the quality of the code, it is only accepted if the resulting state is still superior to the initial one.

- Impact-based sequencing: Determines the optimal order of application of refactorings, ensuring that each step contributes to the cumulative improvement and avoiding any regression in software quality.

Improved Algorithm Methodology

1. Initialization: The system metrics are computed in their initial state, and the set of available refactorings is identified.
2. Evaluation of preconditions: At each iteration, the algorithm determines which refactorings can be applied based on the current state of the system.
3. Selection of the best available refactoring: Among the refactorings that satisfy the preconditions, the algorithm selects the one that produces the best local improvement without risk of degradation.
4. Application of the selected refactoring: The system is transformed and the metrics are updated accordingly.
5. Evaluation of postconditions: The algorithm verifies that none of the quality metrics decrease compared to the previous state. If any metric is degraded, the refactoring is discarded.
6. Iteration until no further options remain: The process continues as long as there exists at least one applicable refactoring that fulfills both preconditions and postconditions.

As part of the illustrative example demonstrating the operation of the Improved Greedy Algorithm, the code of the *Pago* class shown in Figure 3 was analyzed. The objective is to determine the correct order of application of two refactoring techniques: R_1 (modular protection) and R_2 (abstraction).

```
public class Pago {

    private void validar() { }

    public void procesar() {
        validar();
    }
}
```

Fig. 3. Pago class

Applying the preconditions and postconditions defined in Equations 8–17 for each technique, the algorithm determined that the optimal sequence for this case is:

$$S^* = [R2, R1]$$

The correctness is verified by manually applying the metrics.

Metrics used

Modular protection of private methods

$$\text{PMFP} = \frac{\sum_{ci=1}^{ci=n} \left(\frac{\sum_{fi=0}^{fi=m} \text{FP}}{\text{FTC}} \right)}{\text{NTC}} \quad \textbf{(1)}$$

Modular protection of protected methods

$$\mathrm{PMFPr} = \frac{\sum_{ji=1}^{ji=n}\left(\frac{\sum_{fi=0}^{fi=m}\mathrm{FPr}}{\mathrm{NTF}}\right)}{\mathrm{NTJ}} \tag{2}$$

Modular protection of default methods

$$\mathrm{PMFF} = \frac{\sum_{i=0}^{i=n}\mathrm{FF}}{\mathrm{NTF}} \tag{3}$$

Total modular protection

$$\mathrm{TPM} = \frac{\left(\left(\mathrm{PMFP}*1\right) + \left(\mathrm{PMFPr}*0.75\right) + \left(\mathrm{PMFF}*0.25\right)\right)}{NTF} \tag{4}$$

Abstraction

$$\mathrm{A} = \frac{Na}{NC} \tag{5}$$

From the example, for the class *Pago*:

$$PMP = 0,\ PMPr = 0,\ PMF = 0,\ NTM = 2,\ Na = 0,\ Nc = 0 \tag{6}$$

Therefore, the modular protection level (PMT) is:

$$\mathrm{A} = \frac{Na}{NC} \tag{7}$$

The initial state of the sequence is $S = \varnothing$

The preconditions and postconditions for each refactoring technique were as follows:

R_1 - Modular Protection

Precondition: At least one method must exist.
Postcondition: Modular protection must improve or remain unchanged.
Postcondition: None of the previously evaluated metrics may decrease.

R_2 - Abstraction

Precondition: At least one public method must exist.
Postcondition: Abstraction must improve or remain unchanged.
Postcondition: None of the previously evaluated metrics may decrease.

Applying Sequence 1: R1→R2

Application of $\boldsymbol{R}_1$: Applying the modular protection refactoring first, yields the following values: $PMT_1 = 0.5$ and $A_1 = 0$

We verify the postconditions of $\boldsymbol{R}_1$: They are satisfied.

We update: $PMT_{best} = 0.5$ and $A_{best} = 0$
Applying $\boldsymbol{R}_2$ to the refactored code yields: $PMT_2 = 0.33$ y $A_2 = 0.5$

We now check the postconditions of $\boldsymbol{R}_2$:
$A_2 \geq A_1 \rightarrow 0.5 \geq 0$ True
$PMT_2 \geq PMT_1 \rightarrow 0.33 \geq 0.5$ False

Based on the postconditions, it is concluded that the sequence is invalid, as it reduces the improvement previously achieved with $\boldsymbol{R}_1$.

Applying Sequence 2: R2→R1

Application of $\boldsymbol{R}_2$: when applying the abstraction technique, the following is obtained: $PMT_1 =$ 0 and $A_1 = 0.5$

We verify the postconditions of $\boldsymbol{R}_2$: They are satisfied.

We update: $PMT_{best} = 0$ and $A_{best} = 0.5$

Applying $\boldsymbol{R}_1$ to the refactored code yields: $PMT_2 = 0.33$ and $A_2 = 0.5$

We now check the postconditions of $\boldsymbol{R}_1$:
$A_2 \geq A_1 \rightarrow 0.5 \geq 0.5$ True
$PMT_2 \geq PMT_1 \rightarrow 0 \geq 0.33$ True

The results show that:

- The sequence $R_1 \rightarrow R_2$ is invalid because it reduces the modular protection metric, thus violating the postconditions of the algorithm.
- The sequence $R_2 \rightarrow R_1$ preserves and increases the metrics at each step.

Therefore, the sequence proposed by the Improved Greedy Algorithm: $S^* = [R2, R1]$, is correct and represents the optimal ordering.

4 Greedy Mathematical formalization

To formalize the process of selecting and ordering refactorings, a quality function and a set of preconditions and postconditions are defined. This formalization specifies how the Improved Greedy Algorithm evaluates, selects, and validates each refactoring to construct a sequence that preserves or improves the structural quality of the software.

In order to ensure precision and consistency, this section introduces the mathematical notation used throughout the formal model, followed by the fundamental equations that govern the algorithm. The table 2 summarizes all symbols, functions, and variables used in the mathematical formalization of the Improved Greedy Algorithm.

Table 2. Mathematical Notation Used in the Improved Greedy Algorithm

Symbol	Meaning
C	Set of classes in the system.
$R = \{R_1, R_2, \ldots, R_n\}$	Set of available refactoring techniques.
R_i	Specific refactoring technique.
S	Accumulated (partial) sequence of applied refactorings.
$F(S)$	Quality of the system after applying sequence S.
ΔF_{Ri}	Quality gain obtained after applying refactoring R_i.
$PMT(S)$	Modular protection metric after applying S.
$A(S)$	Abstraction metric after applying S.
W_{PMT}	Weight associated with the modular protection metric in the quality function.
W_A	Weight associated with the abstraction metric in the quality function.
$pre(R_i, S)$	Precondition verifying whether R_i is applicable at the current state.
$post(S \cup \{R_i\})$	Postcondition ensuring that applying R_i does not degrade previously achieved quality.
k	Iteration index of the Improved Greedy Algorithm.
a_k	Refactoring selected in iteration k.
S^*	Optimal sequence produced by the Improved Greedy Algorithm.

Quality Function

The quality of the system after applying the sequence *S* is defined as follows:

$$F(S) = W_{PMT} * PMT(S) + W_A * A(S) \quad (8)$$

Where the weights satisfy:

$$W_{PMT} + W_A = 1 \quad (9)$$

The quality gain obtained by applying a refactoring R_i is:

$$\Delta F_{Ri} = (F(S \cup \{R_i\}) - F(S) \quad (10)$$

Preconditions and Postconditions

General precondition

$$pre(R_i, S) \leftrightarrow R_i \quad (11)$$

is applicable in the current state of the code.

General postcondition

$$post\left(S \cup \{R_i\}\right) \leftrightarrow F\left(S \cup \{R_i\}\right) \geq F\left(S\right) \quad (12)$$

This condition ensures that no refactoring degrades the accumulated quality.

Selection rule of the Improved Greedy Algorithm

In each iteration, the algorithm selects the next refactoring:

$$a_k = \arg\max_{R_i \in R \setminus S} \{\Delta F_{R_i} | pre\left(R_i, S\right) \Delta post\left(S\{R_i\}\right)\} \quad (13)$$

The final sequence is:

$$S^* = \left[a_1, a_2, \ldots, a_n\right] \quad (14)$$

First-order logical constraint

Every applied refactoring must preserve or improve quality:

$$\forall R_i \in R\left(pre\left(R_i, S\right) \wedge post\left(S \cup \{R_i\}\right) \rightarrow \Delta F_{R_i} \geq 0\right) \quad (15)$$

General formula of the algorithm

$$S^* = \arg\max_{S' \subseteq R}\left(\sum_{R_i \in S'} \Delta F_{R_i}\right) \quad (16)$$

subject to:

$$\forall R_i \in S' : pre\left(R_i, S'_{<i}\right) \wedge post\left(S'_{\leq i}\right) \quad (17)$$

Where:

$S'_{<i}$ is the subsequence prior to applying R_i,

$S'_{\leq i}$ is the subsequence that includes R_i

5 Validation of the modification of the greedy algorithm

To evaluate the performance of the Improved Greedy Algorithm, the proposed approach was applied to a set of 30 applications extracted from the GitHub repository. These systems, developed in Java, exhibit substantial variability in size, structure, and implementation style, with the aim of analyzing the algorithm's behavior in heterogeneous scenarios.

To characterize the structural diversity of the dataset, descriptive statistics were computed considering three general metrics: number of .java files, total number of classes, and lines of code (LOC). Table 3 summarizes the values obtained.

Table 3. Descriptive statistics of the 30 analyzed projects

Metric	Minimum	Maximum	Mean	Median	Standard deviation
.java files	3	1,012	87.2	12	289.4
Total classes	3	1,962	108.4	12	360.8
Lines of code (LOC)	121	44,468	2,632.3	577	8,105.2

The results reveal a high degree of dispersion, particularly in large-scale projects such as *OnJava8-Examples-Maven*, which allows the algorithm to be tested against highly demanding codebases. The asymmetry between the mean and median indicates the presence of numerous small and medium-sized systems alongside a few very large projects, a characteristic that supports a robust evaluation of the algorithm across different levels of complexity. Table 4 below presents the list of applications used, along with links to their public repositories.

Table 4. Detail of Applications and Repository Links

No. Program	Name	URL
1	Blue-Block-master	https://github.com/abc013/Blue-Block
2	chess-system-java-master	https://github.com/acenelio/chess-system-java
3	CompanyManagementSystem-master	https://github.com/fdeniz07/CompanyManagementSystem
4	CustomerManagerApp-master	https://github.com/Sepobanz/CustomerManagerApp
5	FileOperations-master	https://github.com/berkaydursun/FileOperations
6	HospitalProject-master	https://github.com/fdeniz07/HospitalProject
7	InventoryManagementSystem-master	https://github.com/twilsonn/InventoryManagementSystem
8	Java_Console-Course-Management-System-main	https://github.com/DrNeonsy/Java_Console-Course-Management-System
9	Java_real_estate-master	https://github.com/Marcello-Goncalves/Java_Imobiliaria
10	Java-Console-Card-Game-master	https://github.com/dnavas77/Java-Console-Card-Game
11	Game-El-ahorcado-en-Java-master	https://github.com/davidmazparrote/Juego-El-ahorcado-en-Java
12	Library_Management_System_11-Jan-2022-master	https://github.com/hariprakash2113/Library_Management_System_11-Jan-2022
13	Movie-Service-Review-Java-main	https://github.com/lakshygupta/Movie-Service-Review-Java
14	online-quiz-application-in-java-with-jdbc-using-swing-login-registration-timer-master	https://github.com/srilekhadasari/online-quiz-application-in-java-with-jdbc-using-swing-login-registration-timer
15	Password-Generator	https://github.com/KZarzour/Password-Generator
16	PasswordManager-main	https://github.com/codershiyar/PasswordManager
17	Polynom	https://github.com/taijased/univers_cours2_Java/tree/master/Polinom
18	qurbani-animals-market-main	https://github.com/MSarmadQadeer/qurbani-animals-market
19	radio	https://github.com/KI7MT/java-app-examples/blob/master/ConsoleApps/src/beam/example/radio/station/RadioStation.java
20	real-estate-console-app-master	https://github.com/Leo-Chan01/real-estate-console-app
21	SchoolManagement-master	https://github.com/fdeniz07/SchoolManagement
22	shapp-master	https://github.com/vicianm/shapp
23	Simple-Java-Calculator-master	https://github.com/pH-7/Simple-Java-Calculator.git
24	TetrisForDesktopWithJava-master	https://github.com/kkikkodev/TetrisForDesktopWithJava
25	tic-tac-toe-console-master	https://github.com/rohandebsarkar/tic-tac-toe-console
26	tiny-compiler-master	https://github.com/almirfilho/tiny-compiler
27	treinaweb-java-oo-master	https://github.com/treinaweb/treinaweb-java-oo
28	UniversalCalculator-master	https://github.com/dima666Sik/UniversalCalculator
29	Keychains-master	https://github.com/JHORJE18/Llaveros
30	AI-Chat-Bot-master	https://github.com/gazalpatel/AI-Chat-Bot

During the evaluation process, the possible execution combinations of the three refactoring techniques were analyzed. Figure 3 presents the combination tree illustrating the application of modular protection (MR1), lack of abstraction (MR2), and implementation inheritance (MR3). Additionally, the order in which each algorithm executed the refactorings was examined, and the differences in the transformation sequence applied to each system were documented.

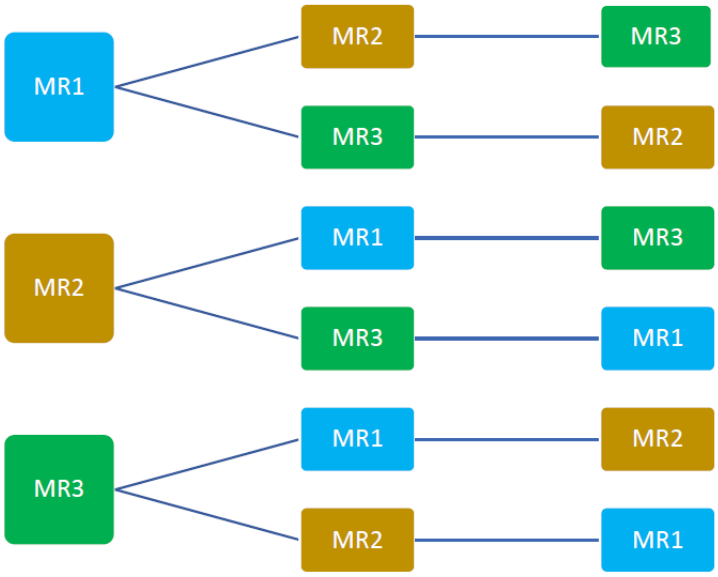

Fig. 4. Combination tree of refactoring methods for lack of abstraction, protection and inheritance of implementation

It is important to emphasize that the purpose of this section is not to determine which algorithm produces a higher accumulated quality, but rather to analyze how they differ in the way they construct their refactoring sequences. The traditional Greedy algorithm evaluates all possible permutations and selects the one with the highest final quality value, without considering whether intermediate degradations occur during the process. In contrast, the Improved Greedy Algorithm immediately discards any step that reduces the quality already achieved, thanks to the explicit enforcement of preconditions and postconditions that guarantee structural stability at each transition.

Under these criteria, the sequence selected by each approach was recorded for the 30 analyzed applications. Table 5 summarizes the resulting sequences and highlights the differences in decision-making between both methods.

Table 5. Sequence selection using the traditional Greedy algorithm and the Improved Greedy algorithm

Application	Sequence selected by the Traditional Greedy algorithm	Sequence selected by the Improved Greedy algorithm
Blue-Block-master	S1	S5
chess-system-java-master	S3	S2
CompanyManagementSystem-master	S1	S3
CustomerManagerApp-master	S2	S4
FileOperations-master	S3	S6
HospitalProject-master	S4	S5
InventoryManagementSystem-master	S2	S3
Java_Console-Course-Management-System-main	S5	S6
Java_real_estate-master	S1	S4
Java-Console-Card-Game-master	S3	S5
Game-El-ahorcado-en-Java-master	S2	S6
Library_Management_System_11-Jan-2022-master	S4	S3
Movie-Service-Review-Java-main	S5	S1
online-quiz-application-in-java-with-	S1	S6

jdbc-using-swing-login-registration-timer-master		
Password-Generator	S3	S2
PasswordManager-main	S2	S4
Polynom	S4	S6
qurbani-animals-market-main	S1	S3
radio	S5	S2
real-estate-console-app-master	S3	S6
SchoolManagement-master	S2	S5
shapp-master	S4	S1
Simple-Java-Calculator-master	S3	S6
TetrisForDesktopWithJava-master	S2	S3
tic-tac-toe-console-master	S5	S4
tiny-compiler-master	S1	S6
treinaweb-java-oo-master	S4	S5
UniversalCalculator-master	S3	S6
Keychains-master	S5	S2
AI-Chat-Bot-master	S2	S4

The results show that the Improved Greedy Algorithm generated different sequences in all cases. This divergence is explained by the nature of each approach: while the traditional Greedy algorithm optimizes only the final outcome, the improved version prioritizes a trajectory without intermediate degradations, which leads to different—and in many cases more stable—decisions for maintaining the structural quality of the software. Throughout the experiments, it was observed that the improved algorithm avoids refactorings that could compromise the achieved state, resulting in a safer and more consistent transformation process.

However, this strict preservation of quality limits the exploration of combinations that, although they may involve temporary degradations, could lead to globally optimal solutions. For this reason, opportunities for future improvements are identified, particularly in strategies that allow balancing stability and exploration.

6 Greedy Results and discussion

The validation results confirm that the incorporation of preconditions and postconditions significantly alters the way refactorings are organized. In every case analyzed, the sequence selected by the Improved Greedy Algorithm differed from the one proposed by the traditional Greedy approach, which validates that both methods pursue fundamentally different objectives. The Improved Greedy Algorithm fulfilled its core purpose:

- Avoiding any intermediate quality degradation,
- Ensuring a progressive and stable improvement, and
- Selecting only those sequences that satisfy the established criteria

This behavior is particularly valuable in software scenarios where structural stability is a priority, as it prevents transformations that could introduce temporary architectural inconsistencies. However, this same rigidity also opens interesting lines of research. Since the improved algorithm may discard potentially optimal sequences due to small temporary degradations, it becomes necessary to explore hybrid mechanisms that allow evaluating trade-offs between stability and global benefit. Overall, the validation demonstrates that the proposed approach provides finer control over the refactoring process, enabling the selection of safer and more coherent sequences, although with reduced exploration capacity compared to traditional methods.

7 Conclusions and future work

The findings of this research confirm that the modification to the Greedy algorithm enabled a more structured approach to the selection of refactoring sequences. The incorporation of preconditions and postconditions contributed to a more controlled ordering of the transformations applied to the code, although resulting in complete divergence from the sequences generated by the traditional Greedy algorithm. This outcome reinforces the hypothesis that adding additional constraints substantially changes the way refactorings are selected and applied.

Despite the progress achieved, it is necessary to continue exploring methods that improve the algorithm's precision in selecting optimal sequences. Based on the results of this study, the following directions for future work are proposed:

- Optimization of the algorithm through the integration of advanced techniques such as neural networks or graph-based search, with the goal of improving the accuracy of sequence selection.
- Extension of the study to other programming languages to evaluate whether the algorithm's effectiveness varies according to software architecture.
- Implementation within integrated development environments (IDEs) to assess its applicability in real-world scenarios and its impact on developer productivity.
- Refinement of the strategy for selecting sequences by incorporating additional techniques such as neural networks or graph-based search to improve the identification of the best sequence.
- Expansion of the evaluation to projects written in different programming languages to analyze how architectural characteristics influence the algorithm's performance.
- Deployment of the algorithm in IDE environments to evaluate its applicability in real time.

Overall, the results indicate that the Improved Greedy Algorithm provides a formal mechanism for controlling quality during the ordering of refactorings, preventing intermediate degradations and enabling more stable sequencing. The proposed formalization and its empirical validation constitute an initial foundation upon which more comprehensive approaches may be developed, approaches capable of balancing quality preservation with the exploration of alternative solutions. Future work should delve deeper into this balance and assess its impact on larger and architecturally diverse systems.

References

Barón, N., Salgado, R., Valenzuela, B., & Alcántara, J. (2023). *Technique to enhance modular protection in legacy software systems*. In *Proceedings of the 12th International Conference on Software Process Improvement (CIMPS)* (pp. 71–77). IEEE. https://doi.org/10.1109/CIMPS61323.2023.10528853

Eilertsen, A. M. (2020). *Refactoring operations grounded in manual code changes*. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings (ICSE '20 Companion)* (pp. 182–185). ACM. https://doi.org/10.1145/3377812.3381395

Fowler, M. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley.

Kaur, G., & Sharma, D. (2015). *A study on Robert C. Martin's metrics for packet categorization using fuzzy logic*. *International Journal of Hybrid Information Technology, 8*(12), 215–224. https://doi.org/10.14257/ijhit.2015.8.12.15

Kurbatova, Z., Veselov, I., Golubev, Y., & Bryksin, T. (2020). *Recommendation of move method refactoring using path-based representation of code*. In *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering Workshops (ICSEW '20)* (pp. 315–322). ACM. https://doi.org/10.1145/3387940.3392191

Mahouachi, R., Kessentini, M., & Ó Cinnéide, M. (2013). *Search-based refactoring detection*. In *GECCO '13 Companion: Genetic and Evolutionary Computation Conference* (pp. 205–206). ACM. https://doi.org/10.1145/2464576.2464680

Meananeatra, P. (2012). *Identifying refactoring sequences for improving software maintainability*. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)* (pp. 406–409). ACM. https://doi.org/10.1145/2351676.2351760

Mens, T., & Tourwé, T. (2004). *A survey of software refactoring*. *IEEE Transactions on Software Engineering, 30*(2), 126–139. https://doi.org/10.1109/TSE.2004.1265817

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks* (Doctoral dissertation). University of Illinois at Urbana–Champaign.

Ortiz-Gutiérrez, O., Santaolaya-Salgado, R., Fragoso-Díaz, O.-G., & Rojas-Pérez, J.-C. (2019). *Métricas para la medición del factor de flexibilidad y el factor de herencia de implementación de sistemas de software*. *RISTI – Revista Ibérica de Sistemas e Tecnologias de Informação*, (34), 97–111. https://doi.org/10.17013/risti.34.97-111

Ouni, A., Kessentini, M., & Sahraoui, H. (2014). *Multiobjective optimization for software refactoring and evolution*. In *Advances in Computers* (Vol. 94, pp. 103–167). Elsevier. https://doi.org/10.1016/B978-0-12-800161-5.00004-9

Tarwani, S., & Chug, A. (2020). *Assessment of optimum refactoring sequence to improve the software quality of object-oriented software*. *Journal of Information and Optimization Sciences*. https://doi.org/10.1080/02522667.2020.1809097

Tsimakis, A., Zarras, A., & Vassiliadis, P. (2019). *The three-step refactoring detector pattern*. In *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLop '19)* (pp. 1–9). ACM. https://doi.org/10.1145/3361149.3361168