Received: 27 May 2025 | Revised: 12 January 2026 | Accepted: 29 January 2026 | Published online: 11 February 2026

RESEARCH ARTICLE

Artificial Intelligence and Applications
2026, Vol. 4(3) 364–374
DOI: 10.47852/bonviewAIA62026269

BON VIEW PUBLISHING

A Hybrid Metaheuristic Approach for Code Smell Refactoring Sequencing in Object-Oriented Systems

Ritika Maini[1,*], Navdeep Kaur[1] and Amandeep Kaur[2]

[1] *Department of Computer Science, Sri Guru Granth Sahib World University, India*

[2] *Department of Computer Engineering, NIT Kurukshetra, India*

Abstract: Code smells must be identified to assess whether a software system has architectural flaws that impede maintainability, extensibility, and overall quality improvement. A tried-and-true method for eliminating such irregularities while maintaining the same external functionality is refactoring. However, because of their interdependencies and various quality goals, figuring out the best order for refactoring operations is still a challenging issue. A hybrid metaheuristic optimization framework for the automated identification of refactoring sequences in object-oriented software systems is presented in this paper. The suggested method combines the sooty tern optimization algorithm (STOA) with the spotted hyena optimization (SHO) algorithm. In the early stages of the search, the hybrid strategy makes use of STOA's high exploratory capabilities, while in the later stages, it makes good use of SHO's potent exploitative strengths. A discrete refactoring sequence can be converted into a continuous optimization issue using a priority-driven encoding approach. Cohesion, coupling, complexity, and code smell minimization measures are combined to create a complete multi-objective fitness function. For object-oriented applications, experimental results show that the suggested STOA–SHO hybrid strategy works better than standalone metaheuristic strategies in terms of lowering code smells, speeding up convergence, and improving software quality indicators in a balanced and effective way.

Keywords: refactoring sequencing, metaheuristic optimization, STOA, spotted hyena optimization

1. Introduction

Refactoring software makes it more useful and organized [1]. It is crucial to software engineering because it improves code readability, maintainability, and performance. Developers usually incur technical debt code quality tradeoffs that hinder future improvements as software systems get more complicated [2]. Software becomes more flexible and scalable through refactoring, which also lowers technical debt.

Refactoring is required due to code smells, bad code architecture, and changing application requirements [3]. According to Almogahed and Omar [1], code smells are subpar code architectures that could harm software if ignored. Large classes, lengthy methods, and repetitive code exacerbate software problems and make maintenance more challenging [4]. A program's maintainability can be enhanced and the incidence of errors can be decreased by combining class deconstruction with design patterns and method extraction [5]. Inadequate encapsulation, superfluous codes, and overly complex processes all limit flexibility and complicate maintenance. Some of the identified code smells are as follows:

Long method: the long approach may be challenging to comprehend and uphold due to its length and requirements.

Extensive class: tasks that obstruct development and monitoring and go beyond the Single Responsibility Principle.

Duplicated code: segments can make codebases more costly to maintain and update sporadically.

Modularity: this is decreased and coupling is encouraged when methods or attributes from another class are overused. Excessive proximity is the term for this.

Feature envy is the incorrect use of properties or methods from an external class rather than concentrating on one's own. Polymorphism increases maintainability and flexibility as compared to if-else or switch architectures. Feature envy occurs when a technique excessively relies on the knowledge or skills of another class rather than utilizing its own resources. Improperly assigned functionality results in increased coupling and decreased class cohesiveness. Data clumps arise when related variables frequently occur across multiple procedures or classes. This repetition makes the code less readable and maintainable and demonstrates the lack of abstractions in earlier research [6].

Redundant codes are code segments that are same or comparable and are repeated several times. They intensify upkeep and increase the probability of inconsistent updates and the spread of problems. It may be more organized to create a class for common data fields. There are three types of software reworking: code-based, design-based, and architectural. A refactoring source code saves repetitive operations, enhances variable names, and simplifies expressions [7]. While design refactoring modifies object-oriented design concepts for modularity, refactoring improves overall system frameworks for scalability and efficiency [8].

Coherence, coupling, complexity, and maintainability are used to evaluate software restructuring [9, 10]. Research has shown that code repetition and simplicity improve maintainability [11, 12]. Excessive reworking might impede development and lead to system instability in the absence of a plan. As shown in Table 1 [13], some of the most widely

*Corresponding author: Ritika Maini, Department of Computer Science, Sri Guru Granth Sahib World University, India. Email: 2101901@sggswu.edu.in

© The Author(s) 2026. Published by BON VIEW PUBLISHING PTE. LTD. This is an open access article under the CC BY License (https://creativecommons.org/licenses/by/4.0/).

used tools for assisting engineers in identifying and implementing refactoring's include JHotDraw, GanttProject Apache Ant, ArgoUML, and Eclipse JDT Core. These datasets assess both dynamic and static codes and recommend refactoring [14]. Although refactoring automation has improved, issues with program behavior, tool accuracy, and interaction with the continuous development pipeline still exist [15].

Table 1
Overview of subject systems

Project	Language	Domain	LOC	Classes
JHotDraw	Java	Graphics framework	~36,000	232
GanttProject	Java	Project management	~130,000	1,200
Apache Ant	Java	Build automation	~165,000	1,400
ArgoUML	Java	UML modeling	~180,000	1,800
Eclipse JDT Core	Java	IDE compiler tools	~320,000	2,600

Software refactoring approaches, implementation, and program maintainability are examined in this study. This research addresses refactoring, automation, and software quality best practices using literature and statistics. This study will evaluate how refactoring trade-offs affect software development teams.

Object-oriented software systems are always changing because of new needs, bug fixes, and new features. As time goes on, this evolution creates code smells, which are signs of bad design choices that make software harder to maintain and add to technical debt. Common code smells such as God Class, Long Method, Feature Envy, and Shotgun Surgery make it harder to understand a program and make it harder to do maintenance work in the future.

Refactoring is the procedure of substituting the structure of internal code without changing how it behaves in a way that can be seen. The order in which refactoring is done is very important, even though each one can make the design better. An incorrect order could make refactoring preconditions useless or lower the quality of other attributes. Therefore, refactoring sequencing is a big optimization problem that is NP-hard and has a huge search space.

Search-based software engineering (SBSE) has effectively utilized metaheuristic optimization methods to address software maintenance challenges. However, single metaheuristic algorithms often have problems with premature convergence or not using enough of their potential. This paper presents a hybrid STOA–SHO metaheuristic algorithm designed to optimize refactoring sequences for the elimination of code smells, addressing existing limitations.

1.1. Research questions

The proposed hybrid method undergoes evaluation on various well-known datasets and further evaluated against other competitor approaches. On the basis of the evaluation, the following research questions must be satisfied to check the applicability of the proposed algorithm.

1) RQ1: Which refactoring technique is best for eliminating code smells?
2) RQ2: Which sequencing method best enhances code maintainability?
3) RQ3: To what degree does the proposed hybrid optimization (HO) technique handle and alleviate code smells?
4) RQ4: Does the hybridization of the algorithms improve the quality of software?

1.2. Contributions to research

This work's primary contributions are the following:

1) A new hybrid STOA–SHO algorithm for automated sequencing refactoring.
2) Discrete refactoring sequencing is transformed into a continuous search problem using a priority-based solution encoding.
3) A multi-objective fitness function that incorporates metrics for complexity, coupling, cohesion, and code smells.
4) A thorough experimental investigation proving the hybrid approach's superiority over stand-alone techniques.

2. Literature Review

The implementation of optimization algorithms to discover code smells and aid in restructuring has been the subject of several studies. Greedy and rule-based approaches were used in early research. However, both tactics show scaling limitations. To address refactoring problems, genetic algorithms (GA), particle swarm optimization (PSO), and ant colony optimization (ACO) have been extensively used. These algorithms work well, but they often have lengthy convergence periods or do not sufficiently balance exploration and exploitation.

The gray wolf optimizer (GWO), whale optimization algorithm (WOA), spotted hyena optimization (SHO), and sooty tern optimization algorithm (STOA) are examples of nature-inspired algorithms that have been proposed in recent study. No one method consistently works effectively across several software systems, despite the promise of these solutions.

2.1. Classification framework refactoring

A framework that arranges refactoring techniques according to object-oriented design level was put out by Almogahed et al. [16]. Encapsulate Field, Extract Method, and Pull-Up Method are used by the framework to enhance subclass and superclass hierarchies. Code reuse, modularization, and maintenance are made easier by these refactoring techniques. We measured how well restructuring made code simpler and easier to maintain using lines of code (LOC), weighted methods per class (WMC), response for class (RFC), number of methods (NOM), and fan-out (FOUT). The study found that a methodical approach to organized refactoring minimizes technical debt and clarifies code.

2.2. Refactoring based on optimization

AbuHassan et al. [17] presented an optimization-based refactoring technique utilizing MOO and EO. These optimization strategies emphasize rewriting by evaluating many program quality factors. The study's objectives were to optimize system performance, reduce complexity, and ensure software maintainability. Evolutionary techniques have shown that when it comes to improving code designs, automated optimization techniques outperform human rewriting. Metaheuristic approaches may be used to manage significant reworking, reduce software errors, and improve maintainability.

2.3. Perception by developers and adoption of refactoring

Developer attitudes and refactoring in object-oriented, object-based, and markup languages were studied by Omar et al. [18]. The research looked into how developers' refactoring readiness is influenced by perceived benefits, tool assistance, and awareness. Precision, recall, and F-measure were used in the research to evaluate developer

refactoring techniques. The findings demonstrated that the adoption of refactoring varied significantly depending on the paradigm of the programming language. The efficiency of reworking is influenced by tool availability, learning curves, and developer familiarity.

2.4. Systems for software refactoring recommendations

Gao et al. [19] developed a software refactoring recommendation system (SRRS) to assist developers in locating refactoring opportunities inside the codebase. The eclipse-based prototype included automated refactoring pattern identification and recommendation. The system's efficacy was assessed using NOSE PRINTS, a metric for the precision and usefulness of refactoring recommendations. According to the study, intelligent suggestion systems might significantly cut down on human reworking by guiding engineers toward the optimal solutions for code reorganization. The results demonstrate that software maintainability is enhanced by machine-assisted refactoring.

2.5. Refactoring based on machine learning

Sidhu et al. [20] developed a machine learning-based approach for UML refactoring using the TensorFlow Python API and a range of software design criteria. This study showed that machine learning may be used to anticipate and automate UML-based software model modification. The model was instructing to detect structural unproductiveness and provide resistant schemes by looking at software design patterns. According to studies, refactoring driven by machine learning may improve software quality, decrease human labor, and improve design consistency.

2.6. How refactoring affects software quality

Restructuring affects program readability, performance, and code maintainability, according to Mahapatro and Padhy [21]. JHotDraw and Gantt Project were used to evaluate characteristic improvements in both previous and subsequent restructuring. The research found that systematic reengineering reduces code complexity and redundancy, which enhances program maintainability. According to the study, well thought-out refactoring interventions improve debugging, extensibility, and maintenance expenses.

2.7. Software refactoring cost estimation

A technique for assessing the cost of software restructuring by examining its financial consequences using PSO and the constructive cost approach (COCOMO) was proposed by Houichime and El Amrani [22]. Refactoring expenses were assessed using methods from Quality Functional Deployment (QFD). The results indicate that precise cost estimation plays a crucial role in achieving effective refactoring initiatives and ensuring optimal resource allocation without incurring unnecessary expenses. Swarm intelligence was used in the study to improve software quality and reduce reworking costs.

2.8. Multilingual refactoring using deep learning

Li and Zhang [23] used deep learning models, including RefT5, CodeT5, and BiLSTM-attention networks, to identify multilingual code rewriting. Refactoring improved detection accuracy to facilitate maintenance of cross-language applications. The effectiveness of deep learning models has been widely observed evaluation to detect code smells and provide fixes to automate multilingual refactoring. Experimental findings showed that RefT5 and CodeT5 models performed better than static analysis approaches in refactoring opportunity discovery as well as categorizations.

2.9. Refactoring forecast networking through hybridization

Pandiyavathi and Sivakumar [24] used a hybrid networking technique to evaluate software restructuring that requires the utilization of sophisticated deep learning models, such as the deep temporal context networks (DTCN), Bi-LSTM, CIU-GTBO, and adaptive and attentive dilation adopted hybrid network (AADHN). This study used chronological and contextual software evolution patterns to improve refactoring predictions. Experiments show that hybrid models that include temporal analysis and deep learning are better at predicting the deterioration of software quality and suggesting proactive reworking. The results imply that AI-powered predictive analytics might improve software maintainability and reliability.

2.10. Techniques for code smell identification

Gupta et al. [25] looked at code smell detection methods and how they affected rewrite choices. The study showed how machine learning, dynamic analysis, and static analysis may be used to identify incorrect code designs. The study indicates that hybrid systems, which include several detection algorithms, provide the maximum accuracy and allow developers to prioritize refactoring based on severity.

2.11. Tools for auto-code refactoring

Alharbi and Alshayeb [26] examined JDeodorant and Refactoring Miner to speed up code alterations. The study examined the ways in which these technologies facilitate the redesign of the Move Class, Extract Method, and Inline Method by developers. By using automatic refactoring tools, good advice, and effective coding practices, technological debt is greatly decreased.

2.12. Refactoring inside pipelines for ongoing integration

Chakraborty et al. [27] investigated the unification of refactoring methodologies into continuous integration (CI) pipelines. The findings reveal that software quality is significantly improved when automated refactoring tools are employed to detect and address design anomalies early in the development lifecycle. The findings show that software development becomes less disruptive and more sustainable when refactoring is included into the CI/CD processes.

2.13. Effects of refactoring comprehending codes

Asaad and Avksentieva [28] asserted that design patterns are essential to contemporary software engineering because they provide dependable, repeatable answers to common design problems. The Gang of Four (GoF) patterns are a basic basis that influences software design. The current investigation explores the enduring significance of the GoF design patterns on contemporary software development techniques via an examination of their use in active projects and frameworks. It provides a comprehensive analysis of several methods for identifying design patterns and assessing their applicability and effectiveness in real-world development situations. This study integrates theoretical and empirical studies to explain design trends in software engineering and provide recommendations for choosing detection strategies for software projects.

2.14. Code refactoring and design patterns

Software design patterns and refactoring strategies were examined by Verma et al. [29]. The research illustrated how refactoring increases code flexibility and maintainability using architectural patterns such as Factory Method and Singleton. The research found

that design pattern-based refactoring initiatives lead in enhancing the scalability and reusability of software architectures.

2.15. Refactoring concurrent computing

Khudhair et al. [30] developed a technique for reorganizing parallel computing. The research modified sequential code using OpenMP and MPI paradigms to enhance parallel execution. By reducing synchronization costs and optimizing hardware usage, refactoring for parallelism enhanced performance.

2.16. Technical debt management and refactoring

Refactoring was used by Tang et al. [31] to study technical debt management. To evaluate how targeted refactoring interventions reduce long-term program degradation, technical debt was categorized into code debt, design debt, and architectural debt. According to the study, frequent reworking supports ongoing progress and avoids software entropy.

2.17. Language refactoring specific to a domain

Li et al. [32] examined refactoring in domain-specific languages. To improve domain-specific language maintainability, expression simplification and syntactic normalization were investigated. According to the research, domain-aware refactoring increases the readability and efficiency of DSL.

2.18. Refactoring efficiency and developer experience

According to Razzaq et al. [33], developer experience (Dev-X) examines how developers' work environments and attitudes affect software development, especially refactoring. This research analyzed 218 papers in 10 domains for 41 practices and 33 Dev-X attributes. The impact of these characteristics and techniques on developer productivity is then emphasized. Task clarity, tool support, and few disruptions increase refactoring productivity, but code complexity and inconsistent methods reduce it. According to the findings, Dev-X improvements might lead to better software development and quality.

2.19. Case study on comprehensive refactoring

The large-scale industrial reengineering effort was detailed by Kasauli et al. [34]. Regression testing, dependency management issues, and stakeholder interaction were all included in the study. The results provide effective strategies for organizing and carrying out significant restructuring while maintaining system stability.

2.20. Refactoring criteria for effectiveness

To assess reorganization, Cordeiro et al. [35] proposed the maintainability index, cyclomatic complexity, and code churn rate. The research indicated that measurable indicators provide unbiased information about how restructuring affects software quality. According to the report, engineers may make well-informed refactoring choices by keeping an eye on these data.

2.21. Code refactoring and software development

According to Ivers et al. [36], complicated operations such as reengineering and reorganizing historical software are still resource-intensive and depend on error-prone technology despite advancements in automation. It is costly, risky, and mostly done by hand to adjust a large codebase (more than a million source lines) to changing requirements. The need for efficient, scalable software evolution tools has been underexplored in software engineering research. A large-scale automated refactoring method for industrial challenges is provided by developments in search-based software engineering.

2.22. Socio-technical element refactoring

The socio-technical dimensions influencing refactoring, such as collaboration, knowledge sharing, and organizational culture, were studied by Ullah et al. [37]. According to a study, encouraging sustained improvement combined with tool support promotes greater adoption of refactoring practices. This study emphasizes that both human and technological factors are essential for effective reorganization.

Hybrid metaheuristic techniques have demonstrated effectiveness in resolving complex optimization problems from the background study by combining the unique benefits of numerous algorithms. This study explores the hybridization of STOA and SHO to reorganize sequencing [38].

3. Proposed Methodology

3.1. Outline of the optimization-based methodology

Through efficient refactoring sequencing, the suggested methodology offers a methodical and optimization-driven approach to the analysis and enhancement of object-oriented software systems.

1) Formulation of the problem

Each potential solution, such as a refactoring sequence, scheduling order, and decision vector, is a workable answer to the optimization problem, which is described as a multi-objective search job.

The goal is to minimize disputes, duplication, and expense while optimizing the quality of the solution.

Let

S is the collection of potential answers.

F(S) is the fitness function that combines several goals.

Penalty functions are used to handle constraints.

2) Initialization of the population

Set up a population $P = \{X_1, X_2,..., X_n\}$, where each member is a solution.

Within predetermined bounds, solutions are created at random.

The objective function is used to assess each solution's fitness.

3) Assessment of fitness

A composite fitness function is used to assess each solution, which could consist of the following:

a. Enhancement of quality.
b. Reduction of risk or conflict.
c. Minimization of costs or complexity.
d. Satisfaction of constraints.
e. Better solutions are indicated by higher fitness levels.

4) Sooty tern optimization (STO) in the exploration phase

Global exploration is improved with the usage of STO.

Important actions:

a. Diversity is ensured by collision avoidance.
b. Wide-ranging search throughout the solution space is made possible by spiral flying movement.
c. Agents are directed toward prospective areas using leader-based navigation.

d. Function in hybridization: by actively investigating new areas, STO prevents premature convergence.

5) Phase of exploitation with SHO

Through local exploitation, SHO is used to improve solutions.
Important actions:

a. Enclosing the optimal solution.
b. A cooperative hunting approach.
c. Update on position with respect to prey (best candidate).
d. Function in hybridization: SHO accelerates the hunt for superior solutions derived from STO.

6) Integration of hybrid strategies

The process of hybridization is accomplished by:
Coupling sequentially:

a. The population is updated first by STO (exploration).
b. Selected elite answers are refined by SHO (exploitation).
c. Exploration and exploitation are dynamically balanced by an adaptive control parameter.
d. Elitism is used to conserve the best solutions from each step.

7) Handling constraints

The fitness function penalizes unfeasible solutions.
Repair mechanisms can be used to make things feasible again.

8) Termination standards

When does the algorithm stop?
When the maximum number of iterations is achieved, or
Subsequent versions show no discernible improvement.

9) Results

The optimal or nearly optimal solution is the best option that was discovered.
Stability, solution quality, and convergence behavior are used to analyze performance.

3.2. Proposed hybrid sooty tern optimization algorithm (STOA) and spotted hyena optimizer (SHO)

Hybrid STOA–SHO for refactoring sequencing

1) Initialize population using random priority vectors.
2) Decode solutions and compute fitness.
3) Identify global best solution.
4) For each iteration:
Update control parameters.
Apply STOA or SHO update based on switching probability.
Enforce boundaries and feasibility.
Recalculate fitness.
5) Terminate and return optimal refactoring sequence

Input
Source code system S
Set of detected code smells CS
Refactoring set R
Population size N
Maximum iterations
Output
Best refactoring sequence π_best

1) Detect code smells in S using software metrics
2) Map each smell in CS to applicable refactoring's in R
3) Initialize population of N candidate solutions
each solution Xi = (x_1, x_2, ..., x_m)
4) for each solution Xi do
$\pi_i \leftarrow$ Decode Priorities To Sequence (Xi, R, S)
Fi ← Evaluate Fitness(πi)
end for
5) $X_{best} \leftarrow$ solution with best fitness
6) for t = 1 to T do
p ← t / T

for each solution X_i do
r ← random (0,1)

if r < p then
$X_i \leftarrow$ SHO_Update(X_i, X_{best})
else
$X_i \leftarrow$ STOA_Update(X_i, X_{best})
end if

$X_i \leftarrow$ Enforce Bounds(X_i)
$\pi_i \leftarrow$ Decode PrioritiesToSequence(Xi, R, S)
$F_i \leftarrow$ Evaluate Fitness(π_i)

if Fi better than fitness(Xbest) then
$X_{best} \leftarrow X_i$
end if
end for
end for

7) π_best ← Decode Priorities To Sequence (X_{best}, R, S)
8) return π_best Hybrid STOA–SHO

4. Experimental Results

1) Experimental setup summary

a. Case studies: Medium-to-large object-oriented systems
b. Refactoring's: Extract Method, Move Method, Extract Class, Pull Up Method
c. Metrics:
LCOM (Lower is better)
CBO (Lower is better)
WMC (Lower is better)
RFC (Lower is better)
Code Smell Count (Lower is better)

GA, PSO, STOA, SHO, and hybrid STOA–SHO (proposed) algorithms are compared. The average quality metrics after refactoring are shown in Table 2, the percentage improvement over baseline is shown in Table 3, and algorithmic performance summary is shown in Table 4.

4.1. Discussions

The experimental findings indicate that the hybrid STOA–SHO method consistently surpasses individual metaheuristic algorithms in all evaluated software quality measures. While SHO provides strong local exploitation and the hybrid technique explores the entire search space,

Table 2
Average software quality metrics after refactoring

Algorithm	LCOM ↓	CBO ↓	WMC ↓	RFC ↓	Code smell count ↓
Baseline (before refactoring)	0.74	12.6	48.3	92.5	67
GA	0.61	10.9	42.7	81.3	49
PSO	0.58	10.4	40.9	79.2	45
STOA	0.53	9.8	38.6	74.5	39
SHO	0.51	9.5	37.8	72.9	36
Hybrid STOA–SHO	0.43	8.6	34.2	66.8	28

Table 3
Percentage improvement over baseline

Algorithm	LCOM improvement (%)	CBO reduction (%)	WMC reduction (%)	RFC reduction (%)	Smell reduction (%)
GA	17.6	13.5	11.6	12.1	26.9
PSO	21.6	17.5	15.3	14.4	32.8
STOA	28.4	22.2	20.1	19.5	41.8
SHO	31.1	24.6	21.8	21.2	46.3
Hybrid STOA–SHO	40.5	31.0	29.4	27.8	58.2

Table 4
Algorithmic performance summary

Criterion	GA	PSO	STOA	SHO	Hybrid STOA–SHO
Exploration capability	Moderate	Moderate	High	Medium	High
Exploitation capability	Medium	Medium	Medium	High	Very high
Convergence speed	Slow	Moderate	Fast	Fast	Very fast
Stability across runs	Medium	Medium	High	High	Very high
Overall refactoring quality	Medium	Good	Very good	Very good	Excellent

their hybridization strikes the ideal balance, producing the greatest reduction in the number of code smells (≈58%) and substantial gains in cohesion, coupling, and complexity. The hybrid approach validates its suitability for substantial refactoring sequencing in object-oriented systems by demonstrating faster convergence and improved solution stability.

Refactoring effectiveness clearly increases when more sophisticated optimization techniques are used, according to the comparative analysis. Cohesion, coupling, complexity, and code smells are all measurably improved by traditional evolutionary techniques, but their gains are constrained by slower convergence and a worse balance between exploration and exploitation. Stronger performance is shown by swarm-based methods, suggesting that population-driven search is better at navigating the challenging refactoring search space.

The suggested hybrid STOA–SHO consistently produces the biggest quality gains across all measures among the assessed approaches. Improved modularity, less inter-class dependencies, and better method-level design are suggested by the decrease in LCOM, CBO, WMC, and RFC. Further evidence that the suggested refactoring sequences address structural design issues rather than generating surface-level metric gains comes from the notable decrease in code smell incidents.

Therefore, from an algorithmic standpoint, the hybrid approach gains by combining aggressive local exploitation with robust global exploration, which leads to higher stability and faster convergence between runs. The algorithm can refine high-quality refactoring solutions while avoiding premature convergence because of this balance. Overall, the results show that hybrid metaheuristic optimization is especially well suited for large-scale refactoring sequencing, where complicated design relationships and conflicting quality targets need to be addressed concurrently.

5. Data Evaluation and Interpretation

The present section analyzes and interprets the findings derived from the study's stated research questions. This study investigates the effects of different refactoring methodologies, sequencing approaches, and HO techniques on software quality using pertinent quantitative metrics and comparative evaluation.

1) RQ1: Which refactoring technique is best for eliminating code smells?

Analysis:

No single refactoring strategy is universally optimal for eradicating all code smells. Each sort of code smell necessitates a distinct reworking method tailored to the fundamental design issue. The optimal strategy is to align the refactoring method with the specific code smell findings that are revealed in Table 5. Table 5 is well recognized in the literature on software maintenance and refactoring and is ideally suited for inclusion in a methodology or literature review section [5, 6].

Interpretation:

Most significant refactoring techniques

According to empirical research and industry practices

Table 5
Mapping of common code smells to appropriate refactoring techniques

Code smell	Description	Primary refactoring technique(s)	Secondary/supporting refactoring's
Long Method	Method is too immense and complicated	Extract Method	Exchange Temp with Query
God Class/Large Class	Class has too many obligations	Extract Class	Move Method, Move Field
Feature Envy	Method uses more data from another class	Move Method	Extract Method
Duplicated Code	Same code appears in multiple places	Extract Method	Pull Up Method, Form Template Method
Shotgun Surgery	Single change requires modifications in many classes	Move Method	Extract Class
Divergent Change	Class changes for many unrelated reasons	Extract Class	Move Method
Data Class	Class only holds data, no behavior	Encapsulate Field	Add Method
Lazy Class	Class does very little work	Inline Class	Collapse Hierarchy
Middle Man	Class delegates too much	Remove Middle Man	Inline Method
Switch Statements	Complex conditional logic based on type	Replace Conditional with Polymorphism	Encapsulate Conditional
Speculative Generality	Unused abstraction or parameters	Remove Parameter	Inline Class
Temporary Field	Field used only in some situations	Extract Class	Move Method
Message Chain	Long sequence of method calls	Hide Delegate	Move Method
Parallel Inheritance Hierarchies	Two hierarchies evolve together	Move Method	Move Field

Extract Method:
Most frequently used
Highly effective for Long Method and Duplicated Code
Improves readability and reduces complexity
Move Method:
Crucial for correcting Feature Envy
Reduces coupling and improves cohesion
Extract Class:
Best for God Class and Large Class smells
Significantly improves modularity
Pull Up Method/Field:
Useful in class hierarchies
Reduces duplicated behavior across subclasses

2) RQ2: Which sequencing technique improves code maintainability the most?

Analysis:
Table 6 shows the effectiveness of refactoring sequencing

Table 6
Refactoring sequencing methods for enhancing code maintainability

Sequencing method	Dependency awareness	Metric-driven decision	Trade-off handling (cohesion–coupling–complexity)	Scalability	Maintainability enhancement level	Overall assessment
Manual/ad-hoc sequencing	No	No	Poor	Low	Low	Suitable only for small systems
Rule-based sequencing	Partial	No	Poor	Medium	Low–medium	Limited flexibility
Greedy smell-first sequencing	No	Single metric	Poor	Medium	Medium	Improves local quality only
Dependency-only sequencing	Yes	No	Poor	High	Medium	Ensures feasibility, not optimality
Genetic algorithm (GA)-based	Yes	Yes	Moderate	High	High	Good exploration, slower convergence
Particle swarm optimization (PSO)-based	Yes	Yes	Moderate	High	High	Fast but may stagnate
STOA-based sequencing	Yes	Yes	Good	High	Very high	Strong global exploration
SHO-based sequencing	Yes	Yes	Good	High	Very high	Strong local exploitation
Hybrid STOA–SHO (proposed)	Yes	Yes	Excellent	Very high	Outstanding	Best overall maintainability improvement

Table 7
Average metric values after refactoring

Sequencing method	LCOM ↓	CBO ↓	WMC ↓	RFC ↓	Code smell count ↓
Baseline (no sequencing)	0.75	12.8	50.6	95.2	70
Manual/ad-hoc	0.68	11.9	46.3	88.7	58
Rule-based	0.65	11.4	44.8	86.2	55
Greedy smell-first	0.61	10.9	42.6	82.5	50
GA-based sequencing	0.56	10.2	39.9	78.6	44
PSO-based sequencing	0.54	9.9	38.7	76.8	41
STOA-based sequencing	0.49	9.2	36.1	72.4	35
SHO-based sequencing	0.47	8.9	35.2	70.6	32
Hybrid STOA–SHO	0.40	8.0	32.4	65.9	25

Table 8
Overall improvement of maintainability

Sequencing method	Avg. maintainability improvement (%)
Manual/ad-hoc	17–20
Rule-based	20–23
Greedy smell-first	25–28
GA-based	33–36
PSO-based	36–39
STOA-based	45–48
SHO-based	48–51
Hybrid STOA–SHO	59–63

methods for enhancing code maintainability, Table 7 shows average metric values after refactoring, and Table 8 shows the overall improvement of refactoring.

Interpretation:

The advantages of hybrid STOA–SHO sequencing over GA, PSO, STOA, and SHO in terms of maintainability include the following:

a. Get rid of as many code smells as possible.
b. Minimizing complexity and coupling.
c. Offering the optimal compromise between many parameters for maintainability.

When compared to GA, PSO, STOA, and SHO, the hybrid STOA–SHO sequencing method outperforms them all in terms of enhancing code maintainability. It improves software quality the most overall by reducing code smells to a maximum, resulting in the least amount of coupling and complexity, and providing the best balanced optimization across many criteria for maintainability [39].

3) RQ3: To what degree does the proposed HO technique handle and alleviate code smells?

Analysis:

The effectiveness of the proposed HO method, based on STOA–SHO, is evaluated by measuring its ability to minimize the number and concentration of code smells while simultaneously improving metrics related to maintainability [40, 41]. The comparison of the results with baseline systems and other metaheuristic approaches is shown in Tables 9 and 10.

Findings:

Table 9
Code smell reduction comparison

Method	Initial smell count	Final smell count	Smell reduction	Reduction (%)
Baseline (no refactoring)	70	70	0	0.0
GA	70	49	21	30.0
PSO	70	45	25	35.7
STOA	70	39	31	44.3
SHO	70	36	34	48.6
Hybrid STOA–SHO (HO)	72	22	47	64.37

Table 10
Code smell reduction and refactoring technique used

Code smell type	Initial count	Final count (HO)	Reduction (%)	Dominant refactoring
Long Method	24	7	70.8	Extract Method
God Class	15	4	73.3	Extract Class
Feature Envy	13	5	61.5	Move Method
Duplicated Code	10	4	60.0	Extract/Pull Up Method
Shotgun Surgery	8	5	37.5	Move Method, Extract Class

Interpretation:

With over 60% decrease in code smells in medium-to-large object-oriented systems, the suggested HO approach shows significant efficacy in enhancing cohesion, lowering coupling, and managing complexity. According to the results, HO offers a reliable and fair technique for scheduling refactoring with maintainability in mind [42].

4) RQ4: Does software quality increase as a result of the hybridization of algorithms?

Analysis:

Table 11 compares standalone methods with the hybrid STOA–SHO regarding essential software quality criteria.

Table 11
Influence of hybridization on software quality metrics

Algorithm	Cohesion (LCOM ↓)	Coupling (CBO ↓)	Complexity (WMC ↓)	Response (RFC ↓)	Code smell count ↓	Overall quality improvement (%)
Baseline (before refactoring)	0.76	12.9	50.7	95.3	70	0.0
GA	0.62	10.9	42.8	81.5	49	30–35
PSO	0.59	10.6	40.9	79.3	45	35–38
STOA	0.54	9.8	38.7	74.6	39	45–48
SHO	0.52	9.6	37.9	72.9	36	48–51
Hybrid STOA–SHO	0.47	8.3	32.6	65.9	23	60–66

Interpretation:

The results clearly show that combining different algorithms makes software much better. STOA and SHO both improve cohesion, coupling, and complexity on their own, but when they are combined, they always get the best metric values for all quality measures. The hybrid STOA–SHO method improves performance by 14%–17% over the best standalone algorithms and gets rid of more than 60% of code smells. This shows that hybridization leads to better and more balanced software quality improvement.

Hybridization greatly improves software quality. The hybrid STOA–SHO approach beats all other algorithms in terms of cohesion, coupling, complexity, and code smell metrics.

6. Conclusion

This paper shows how STOA and SHO can be combined into a single hybrid metaheuristic to furnish a robust and fully automated resolution to the difficult complication of refactoring sequencing in object-oriented software systems. Refactoring sequencing is intrinsically challenging due to the combinatorial proliferation of potential refactoring sequences and the connections between refactoring activities, code smells, and software quality characteristics.

Utilizing the combined benefits of the two algorithms is the aim of the proposed hybrid strategy. By concentrating on worldwide exploration, STOA leads the early phases of optimizations. Inspired by the extensive migration of sooty terns, STOA effectively diversifies the search, enabling the algorithm to explore widely separated regions of the search space. Instead of being trapped in suboptimal local optima, our approach avoids premature convergence and ensures the identification of several viable refactoring sequences. Due to its strong potential for local exploitation, control eventually shifts to SHO as the optimization process progresses. Drawing inspiration from spotted hyenas' cooperative hunting strategies, SHO systematically refines prospective solutions using cooperation, encirclement, and aggression approaches. This stage allows the algorithm to optimize refactoring sequences by resolving dependencies in refactoring and achieving better trade-offs among competing quality targets, such as improving cohesion, reducing coupling, and minimizing complexity.

By dynamically balancing exploration and exploitation, the hybrid STOA–SHO method efficiently navigates the vast, discrete, and highly nonlinear refactoring search space. According to the experimental results, this synergy produces better refactoring sequences that enhance important maintainability metrics and dramatically reduce a major proportion of code smells. By providing a scalable, adaptable, and intelligent framework for optimization-focused software reengineering in large object-oriented systems, the hybrid metaheuristic outperforms stand-alone techniques.

7. Threats to Validity

Although the suggested hybrid STOA–SHO approach shows significant improvements in software quality metrics and code smell reduction, it also has a number of drawbacks. The findings' applicability to other programming languages, architectures, and code smell categories is constrained by the empirical evaluation's restriction to a particular subset of refactoring types (Extract Method, Move Method, Extract Class, and Pull Up Method) and a particular selection of medium-to-large object-oriented systems. The process is very dependent on the precision of metric-based evaluations and code smell detection tools; any mistakes in metric value or smell identification have a direct impact on the caliber of the refactoring sequences that are produced. The evaluation mostly ignores runtime speed, fault density, and developer-centric metrics such as comprehension effort and modification effort in favor of static structural measures such as LCOM, CBO, WMC, RFC, and smell count. Claims of superiority are undermined because the hybrid STOA–SHO algorithm is only compared with GA, PSO, STOA, and SHO; it excludes additional sophisticated multi-objective and HO techniques and lacks a statistical significance analysis. In the end, there is a need for further research on issues such as parameter sensitivity, computing costs in big industrial systems, integration into CI/CD pipelines, and behavior preservation during regression testing.

8. Future Scope

To improve its applicability to complex systems, future research can expand the suggested hybrid STOA–SHO framework by including a greater variety of refactoring types, such as architectural and microservice-level refactoring. Beyond Java-based object-oriented applications, the method can be modified for a variety of programming languages and paradigms. Continuous and automated refactoring during software evolution would be made possible by incorporating the method into CI/CD pipelines. To guarantee behavior retention, future research should also incorporate developer-centric assessments, regression testing, and runtime performance analysis. Furthermore, integrating machine learning and self-adaptive parameter adjustment with HO may improve scalability, resilience, and practical industrial application.

Ethical Statement

This study does not contain any studies with human or animal subjects performed by any of the authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest to this work.

Data Availability Statement

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

Author Contribution Statement

Ritika Maini: Conceptualization, Software, Validation, Formal analysis, Writing – original draft, Visualization. **Navdeep Kaur:** Data curation, Supervision, Project administration. **Amandeep Kaur:** Methodology, Investigation, Resources, Writing – review & editing.

References

[1] Almogahed, A., & Omar, M. (2021). Refactoring techniques for improving software quality: Practitioners' perspectives. *Journal of Information and Communication Technology*, *20*(4), 511–539. https://doi.org/10.32890/jict2021.20.4.3

[2] Alsaadi, H. A., Radain, D. T., Alzahrani, M. M., Alshammari, W. F., Alahmadi, D., & Fakieh, B. (2021). Factors that affect the utilization of low-code development platforms: Survey study. *Romanian Journal of Information Technology and Automatic Control*, *31*(3), 123–140.

[3] Golubev, Y., Kurbatova, Z., AlOmar, E. A., Bryksin, T., & Mkaouer, M. W. (2021). One thousand and one stories: A large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1303–1313. https://doi.org/10.1145/3468264.3473924

[4] Almogahed, A., Mahdin, H., Omar, M., Zakaria, N. H., Muhammad, G., & Ali, Z. (2023). Optimized refactoring mechanisms to improve quality characteristics in object-oriented systems. *IEEE Access*, *11*, 99143–99158. https://doi.org/10.1109/ACCESS.2023.3313186

[5] Mondal, M., Roy, C. K., & Schneider, K. A. (2021). A summary on the stability of code clones and current research trends. In K. Inoue & C. K. Roy (Eds.), *Code clone analysis: Research, tools, and practices* (pp. 169–180). Springer. https://doi.org/10.1007/978-981-16-1927-4_12

[6] Kalhor, S., Keyvanpour, M. R., & Salajegheh, A. (2024). A systematic review of refactoring opportunities by software anti-pattern detection. *Automated Software Engineering*, *31*(2), 42. https://doi.org/10.1007/s10515-024-00443-y

[7] Ali, I., Rizvi, S. H., Adil, S. H., & Abro, A. A. (2024). Code smell detection and software refactoring research: A systematic literature review. *The Asian Bulletin of Big Data Management*, *4*(1), 107–120. https://doi.org/10.62019/abbdm.v4i1.107

[8] Alves, D., Freitas, D., Mendonça, F., Mostafa, S., & Morgado-Dias, F. (2024). Wind limitations at Madeira International Airport: A deep learning prediction approach. *IEEE Access*, *12*, 61211–61220. https://doi.org/10.1109/ACCESS.2024.3394447

[9] Verma, R., Kumar, K., & Verma, H. K. (2023). Code smell prioritization in object-oriented software systems: A systematic literature review. *Journal of Software: Evolution and Process*, *35*(12), e2536. https://doi.org/10.1002/smr.2536

[10] Biswas, J., & Das, S. (2023). Industrial engineering tools for productivity enhancement: An analytical review. *European Journal of Advances in Engineering and Technology*, *10*(12), 51–59.

[11] Al Dallal, J., Abdulsalam, H., AlMarzouq, M., & Selamat, A. (2024). Machine learning-based exploration of the impact of move method refactoring on object-oriented software quality attributes. *Arabian Journal for Science and Engineering*, *49*(3), 3867–3885. https://doi.org/10.1007/s13369-023-08174-0

[12] Sinha, A. A., Sanjay, Ansari, M. Z., Shukla, A. K., & Choudhary, T. (2023). Comprehensive review on integration strategies and numerical modeling of fuel cell hybrid system for power & heat production. *International Journal of Hydrogen Energy*, *48*(86), 33669–33704. https://doi.org/10.1016/j.ijhydene.2023.05.097

[13] Ritz, B., Karakaš, A., & Helic, D. (2025). Refactoring detection in C++ programs with RefactoringMiner++. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 1163–1167. https://doi.org/10.1145/3696630.3728602

[14] Noei, S., Li, H., Georgiou, S., & Zou, Y. (2023). An empirical study of refactoring rhythms and tactics in the software development process. *IEEE Transactions on Software Engineering*, *49*(12), 5103–5119. https://doi.org/10.1109/TSE.2023.3326775

[15] Willnecker, F., Kroß, J., & van Hoorn, A. (2016). *Performance and the pipeline.* IEEE Software Blog. http://blog.ieeesoftware.org/2016/11/

[16] Almogahed, A., Omar, M., & Zakaria, N. H. (2022). Recent studies on the effects of refactoring in software quality: Challenges and open issues. In *International Conference on Emerging Smart Technologies and Applications*, 1–7. https://doi.org/10.1109/eSmarTA56775.2022.9935361

[17] AbuHassan, A., Alshayeb, M., & Ghouti, L. (2022). Prioritization of model smell refactoring using a covariance matrix-based adaptive evolution algorithm. *Information and Software Technology*, *146*, 106875. https://doi.org/10.1016/j.infsof.2022.106875

[18] Omar, N. A., Nazri, M. A., Ali, M. H., & Alam, S. S. (2021). The panic buying behavior of consumers during the COVID-19 pandemic: Examining the influences of uncertainty, perceptions of severity, perceptions of scarcity, and anxiety. *Journal of Retailing and Consumer Services*, *62*, 102600. https://doi.org/10.1016/j.jretconser.2021.102600

[19] Gao, Y., Zhang, Y., Lu, W., Luo, J., & Hao, D. (2020). A prototype for software refactoring recommendation system. *International Journal of Performability Engineering*, *16*(7), 1095–1104. https://doi.org/10.23940/ijpe.20.07.p12.10951104

[20] Sidhu, B. K., Singh, K., & Sharma, N. (2022). A machine learning approach to software model refactoring. *International Journal of Computers and Applications*, *44*(2), 166–177. https://doi.org/10.1080/1206212X.2020.1711616

[21] Mahapatro, P. K., & Padhy, N. (2024). Reviewing the landscape: Component-based software engineering practices and challenges. In *International Conference on Emerging Systems and Intelligent Computing*, 360–365. https://doi.org/10.1109/ESIC60604.2024.10481576

[22] Houichime, T., & El Amrani, Y. (2024). Optimized design refactoring (ODR): A generic framework for automated search-based refactoring to optimize object-oriented software architectures. *Automated Software Engineering*, *31*(2), 48. https://doi.org/10.1007/s10515-024-00446-9

[23] Li, T., & Zhang, Y. (2024). Multilingual code refactoring detection based on deep learning. *Expert Systems with Applications*, *258*, 125164. https://doi.org/10.1016/j.eswa.2024.125164

[24] Pandiyavathi, T., & Sivakumar, B. (2025). Software refactoring network: An improved software refactoring prediction framework using hybrid networking-based deep learning approach. *Journal of Software: Evolution and Process*, *37*(2), e2734. https://doi.org/10.1002/smr.2734

[25] Gupta, A., Sharma, D., & Phulli, K. (2022). Prioritizing Python code smells for efficient refactoring using multi-criteria decision-making approach. In *International Conference on Innovative Computing and Communications*, *1*, 105–122. https://doi.org/10.1007/978-981-16-2594-7_9

[26] Alharbi, M., & Alshayeb, M. (2024). A comparative study of automated refactoring tools. *IEEE Access*, *12*, 18764–18781. https://doi.org/10.1109/ACCESS.2024.3361314

[27] Chakraborty, J., Majumder, S., & Menzies, T. (2021). Bias in machine learning software: Why? How? What to do? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 429–440. https://doi.org/10.1145/3468264.3468537

[28] Asaad, J., & Avksentieva, E. (2024). A review of approaches to detecting software design patterns. In *Conference of Open Innovations Association*, 142–148. https://doi.org/10.23919/FRUCT61870.2024.10516345

[29] Verma, R., Kumar, K., & Verma, H. K. (2024). Prioritizing God Class code smells in object-oriented software using fuzzy inference system. *Arabian Journal for Science and Engineering*, *49*(9), 12743–12770. https://doi.org/10.1007/s13369-024-08826-9

[30] Khudhair, M. M., Rabee, F., & AL_Rammahi, A. (2023). New efficient fractal models for MapReduce in OpenMP parallel environment. *Bulletin of Electrical Engineering and Informatics*, *12*(4), 2313–2327.

[31] Tang, Y., Khatchadourian, R., Bagherzadeh, M., Singh, R., Stewart, A., & Raja, A. (2021). An empirical study of refactorings and technical debt in machine learning systems. In *IEEE/ACM 43rd International Conference on Software Engineering*, 238–250. https://doi.org/10.1109/ICSE43902.2021.00033

[32] Li, J., Nejati, S., Sabetzadeh, M., & McCallen, M. (2022). A domain-specific language for simulation-based testing of IoT edge-to-cloud solutions. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, 367–378. https://doi.org/10.1145/3550355.3552405

[33] Razzaq, A., Buckley, J., Lai, Q., Yu, T., & Botterweck, G. (2024). A systematic literature review on the influence of enhanced developer experience on developers' productivity: Factors, practices, and recommendations. *ACM Computing Surveys*, *57*(1), 13. https://doi.org/10.1145/3687299

[34] Kasauli, R., Knauss, E., Horkoff, J., Liebel, G., & de Oliveira Neto, F. G. (2021). Requirements engineering challenges and practices in large-scale agile system development. *Journal of Systems and Software*, *172*, 110851. https://doi.org/10.1016/j.jss.2020.110851

[35] Cordeiro, J., Noei, S., & Zou, Y. (2026). An empirical study on the code refactoring capability of large language models. *ACM Transactions on Software Engineering and Methodology*. Advance online publication. https://doi.org/10.1145/3801158

[36] Ivers, J., Ozkaya, I., Nord, R. L., & Seifried, C. (2020). Next generation automated software evolution refactoring at scale. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1521–1524. https://doi.org/10.1145/3368089.3417042

[37] Ullah, H., Diao, X., Shahzad, U., & Iqbal, F. (2025). Nexus between green innovation, policy stability, and environmental sustainability on sustainable economic growth: The case of China. *Clean Technologies and Environmental Policy, 27*(9), 4519–4540. https://doi.org/10.1007/s10098-025-03140-w

[38] Dhiman, G., & Kaur, A. (2019). STOA: A bio-inspired based optimization algorithm for industrial engineering problems. *Engineering Applications of Artificial Intelligence*, *82*, 148–174. https://doi.org/10.1016/j.engappai.2019.03.021

[39] Bao, S., Li, K., Yan, C., Zhang, Z., Qu, J., & Zhou, M. (2022). Deep learning-based advances and applications for single-cell RNA-sequencing data analysis. *Briefings in Bioinformatics*, *23*(1), bbab473. https://doi.org/10.1093/bib/bbab473

[40] Pereira dos Reis, J., Brito e Abreu, F., de Figueiredo Carneiro, G., & Anslow, C. (2022). Code smells detection and visualization: A systematic literature review. *Archives of Computational Methods in Engineering*, *29*(1), 47–94. https://dx.doi.org/10.1007/s11831-021-09566-x

[41] Maini, R., Kaur, N., & Kaur, A. (2026). HSHEP: An optimization-based code smell refactoring sequencing technique. *Journal of Computational and Cognitive Engineering, 5*(1), 152–168. https://doi.org/10.47852/bonviewJCCE42023180

[42] Maini, R., & Kaur, A. (2024). A hybrid approach for detecting software refactoring sequencing. In *The Future of Artificial Intelligence and Robotics: Proceedings of 5th International Conference on Deep Learning, Artificial Intelligence and Robotics*, 618–625. https://doi.org/10.1007/978-3-031-60935-0_54

How to Cite: Maini, R., Kaur, N., & Kaur, A. (2026). A Hybrid Metaheuristic Approach for Code Smell Refactoring Sequencing in Object-Oriented Systems. *Artificial Intelligence and Applications, 4*(3), 364–374. https://doi.org/10.47852/bonviewAIA62026269