RESEARCH ARTICLE

BON VIEW PUBLISHING

HSHEP: An Optimization-Based Code Smell Refactoring Sequencing Technique

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: The process of refactoring enhances software quality by modifying its design composition while preserving its core framework. However, addressing code smells without appropriate prioritization can be ineffective. Code smells significantly increase maintenance costs and obstruct system evolution. Refactoring sequencing techniques mitigate these issues by improving a system's internal structure without altering its external behavior. In large-scale systems, the sheer number of code smells can be overwhelming, and not all can be automatically resolved. Hence, prioritizing code smells based on criteria such as risk and importance is essential. This paper introduces a novel hybrid approach utilizing the hybrid spotted hyena and emperor penguin (HSHEP) optimization-based algorithm. This approach aims to optimize the sequence of code smell bugs by incorporating maintainer opinions and requirements, thereby maximizing the resolution of critical code smells. Unlike existing technologies, the HSHEP algorithm combines the strengths of two optimization strategies, offering a unique and innovative solution to refactoring challenges. To validate the effectiveness of the proposed method, it was applied to various large-scale open-source systems, analyzing five different types of code smells. Results demonstrated a significant improvement in maintenance efficiency and system evolution, confirming the superior performance and practical applicability of the HSHEP-based approach.

Keywords: software refactoring, code smell, sequencing, optimization, Spotted Hyena Optimizer (SHO), Emperor Penguin Optimizer (EPO)

1. Introduction

During development life cycle of a software, maintenance of software holds a crucial position. However, maintaining software has become increasingly challenging due to evolving requirements. Programmers continuously modify the source code to adapt to these changes and enhance software features. Consequently, a significant portion, approximately eighty percent, of project costs is allocated to maintenance activities. Identifying code smells plays a vital role in reducing maintenance costs [1]. Smells of code seem natural flaws during programming describe bad software architecture and make it harder to maintain. They draw attention to problems in software application design. When code smells are present, the programming code is not easily understandable and may result in more changes and errors. A programmer's comprehension of the computer code is improved by identifying and eliminating code smells at the source of the code. Not all code smells can be automatically addressed, especially in large-scale systems. Determining the refactoring processes for the discovered code smells might be difficult. Maintenance and evolution of large-scale systems account for approximately 87% of the total software costs [2]. These costs are attributed to tasks like adding new functionalities, bug corrections, and code modifications to enhance quality. However, these continuous changes can lead the software far away from its distinctive design and

architecture, introducing code-smells, which are bad design effects [3]. Code-smells have an adverse effect on quality characteristics like adaptability and durability and are often unintentionally brought in by software developers during very first development as the outcome of poor design choices or maintenance decisions. Coding is to enhance an application's overall performance so that it runs, remains active for longer, and is visible. It provides no indication as to whether the application will run or not; it may still produce a result; it may take longer to process the code and raise the possibility of errors and defaults during the coding process. As the name suggests, code smells are observable or quickly apparent, but they also indicate a more serious issue [4]. The best part is that it gets easier to find, but it also presents an interesting challenge—classes with data but no behavior, for example. It is easy to collect coded scents with the use of equipment. Each character that indicates a more significant issue within the code base is called a code smell. Rather than being flaws or defects, code smells are a violation of application development standards that degrade the software. Code smells have the potential to slow down processing, raise the possibility of errors and failures, and make the program more prone to future issues, but they do not give precise details about how the software functions. In order to eliminate code smells in such systems, it becomes vital to order the refactoring processes according to the maintainer's preferences. Coding, as many of us know, takes a lot of time and requires several programmers. Throughout the coding process, the code will be reviewed, modified, and perhaps increased or lowered [5, 6]. This means that programmers work under a lot of pressure, and long or redundant code may occasionally result.

---

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

Code odors are caused by such laborious coding processes. Thus, this flaw in coding that makes it difficult to grasp and unreliable results in terrible programming that takes a lot of time and space. Refactoring is required to eliminate this programming degradation. Refactoring is the technique of changing a part of application structure rather than its nature. Refactoring is a methodical process of reconstructing a framework of existing program while maintaining inner organization and exterior behavior. The term refactoring is coined by Johnson and Opdyke in 1993 [7]. Even though the software industry has undergone many changes, the requirement for software refactoring remained. Refactoring strong impact on software quality causes it to become a hot topic of study. Software restructuring and software quality have been the subject of significant amounts of research. Refactoring frequently also makes it easier to spot issues and bad code. This has an advantageous result in software's parameters. Software applications include many codings and so the programmers and those highly skilled coders develop very big codes. As the programming run in different hands and through different techniques, there are more chances of errors and not an easy task. In case the issue has progressed a long way, code refactoring can be delayed. As refactoring is nothing but updating or changing the unnecessary coding. Code refactoring includes a number of methods to minimize code length and execution time, and an optimized code is received with the working of original classes, methods, and functions used in the starting of the program [8]. This refactoring is less costly than starting from new program coding. The major help is in iterations and increments where the software is divided into a number of parts and their coding can be easily changed. Refactoring cannot alter the class, functions, or classes of programme or software system. Undisrupted code, on either hand, might miss design code, as well as clears and maintains code quality, resulting in a programme that is easy to understand and comprehend, as well as error free. As a result, it reduces software bugs, but it is also about seeing the code base as just a living system that requires frequent maintaining of code to be strong. This article presents a hybrid approach to prioritize the refactoring techniques for the detected code smells. There are systems that have code smells, which leads to an increase in cost to maintain the product and there are more challenges to alter as well as emerge. Refactoring initiates the architecture of a program by modifying the underlying model irrespective of changing the extrinsic action, in order to eliminate code smells. Large-scale systems have a higher number of code smells to correct, and only some of them can be resolved. As a result, the list of code-smells needs to be sequenced depending on many factors, including the danger and significance of classes [9]. However, the majority of the refactoring techniques in use according to the same importance for fixing code smells. Refactoring sequencing, a widely used technique, is employed to address code-smells and improve its structure while retaining overall functionality and actions. This process involves two primary steps: (1) Detecting portions of code that require improvement, such as code-smells, and (2) identifying suitable refactoring sequencing techniques to achieve the desired improvements. The initial approach involves applying a novel optimization algorithm namely Hybrid Spotted Hyena and Emperor Penguin optimizer (HSHEP). This hybridized algorithm commonly aims to combine the strengths of both approaches. Once the results from the initial spotted hyena optimizer (SHO) algorithm are obtained, we run the hybridized algorithm to demonstrate the advantages of combining with EPO algorithm. This optimization-based code smell refactoring sequencing technique could help to ensure that code refactoring is completed efficiently and effectively,

reducing the risk of introducing new bugs or affecting the code's functionality.

> RQ1: Which optimal refactoring technique should be proposed on code smell?
> RQ2: Which refactoring sequence should be taken in order to increase code maintainability?
> RQ3: To what degree can the suggested approach (HSHEP) effectively address and correct code-smells?
> RQ4: Does the hybridization of two algorithms (HSHEP) lead to any improvement in quality of software?

The rest of this paper is structured as follows: Section 2 elaborates the study of previous work. Section 3 explains a proposed hybrid algorithm. In Section 4, the proposed method is applied to software refactoring technique for sequencing to remove the code smells. Computational complexity and results are given in Sections 5 and 6, respectively. Finally, conclusion is given in Section 7.

2. Literature Review

In the software engineering community, refactoring prioritization and analysis have not received much attention compared to smell refactoring processes. This is not due to the problem's lack of usefulness, but rather because of the inherent challenges linked to hybrid optimization challenges. In this section, the focus is on reviewing and highlighting alternatives for smell refactoring establishing a priority as shown in Table 1.

Kalhor et al. [10] proposed a high priority on antipattern identification, and several methods have been put forth to achieve this goal. There have been published articles of review thus far to categorize and contrast these methods. Nevertheless, a thorough investigation employing assessment criteria has not contrasted various antipattern identification techniques across all program abstraction levels. All the techniques that have been previously described are categorized in this article, followed by a discussion of their benefits and drawbacks. Ultimately, a comprehensive comparison of evaluation measures for every category is given.

Jain and Saha [11] propose two machine learning classifiers to find stinky methods and classes: k-nearest neighbor (kNN) and support vector machine (SVM). This explains its adaptability, SVM excels when used with datasets of modest to moderate size, data spaces of having higher dimensions, along with kernel methods for nonlinear data. The current study avoids this difficult problem and does away with manual intervention by using the power of meta-heuristic algorithms to determine the ideal values for machine learning classifier hyperparameters. It offers a unique method for code smell detection by combining machine learning classifiers with swarm-based methods. This cross-domain combination offers a fresh approach to the age-old problems of precise code smell detection and ideal hyperparameter setup.

Noei et al. [12] introduced MLRefScanner, a prototype tool created expressly to find refactoring commits in ML Python project histories. MLRefScanner uses machine learning techniques and algorithms to increase the coverage of refactoring operation identification in Python code and improve the capabilities of identifying refactorings unique to ML. A deeper understanding of the breadth and evolution of the ML codebase is made possible by MLRefScanner, which enables practitioners and researchers to track and evaluate refactoring actions in the software development history of ML Python libraries and frameworks with high precision and recall.

Table 1
Literature review

Sr. No.	Authors Name	Year	Paper Title	Tool/Technique	Purpose of the Study
1	Kalhor et al. [10]	2024	A systematic review of refactoring opportunities by software antipattern detection.	Clustering Techniques, Q-Modularity, Anti Pattern Detection Tools.	This article, followed by a discussion of the benefits and drawbacks. A thorough comparison of all the categories based on evaluation measures is offered. Three factors are taken into account in their suggested classification: the degree of abstraction, the reliance on the abilities of developers, and the methods employed. After then, an analysis is done on the evaluation metrics that have been reported on this topic, and the metrics' qualitative values for each category are shown. Researchers can use this data to compare, comprehend, and enhance already used approaches.
2	Jain and Saha [11]	2024	Improving and comparing performance of machine learning classifiers optimized by swarm intelligent algorithms for code smell detection.	Support vector machine (SVM) and k-Nearest Neighbor (kNN).	This work focuses on optimizing hyperparameters through twelve swarm-based methods to increase the performance metrics of machine learning algorithms. The best-performing swarm-intelligent algorithms include Salp Swarm Optimizer, Grey Wolf, and Artificial Bee Colony. With optimized classifiers, the odors that are easiest to identify are God and Data Class. Tests of statistical significance confirm the significant influence of using swarm-based methods to optimize machine learning classifiers. This smooth integration provides a solid answer to a recurring software engineering problem while also improving classifier performance and automating code smell detection.
3	Noei et al. [12]	2025	Detecting Refactoring Commits in Machine Learning Python Projects: A Machine Learning-Based Approach	Code conversion and AST pattern matching.	This comprehensive research encompasses 199 ML Python applications, and we validate MLRefScanner's functionality across several testing scenarios. MLRefScanner attains 94% precision, 82% recall, and 82% AUC. 89%, and a minimal feature set is offered to differentiate refactoring commits. With an average of 21% higher precision, 52% higher recall, 28% higher AUC, and 45% higher F1 scores in the mixed-projects scenario, our technique beats the state-of-the-art refactoring detection methods. PyRef, the most up-to-date and precise refactoring detection tool, may be used to resemble MLRefScanner and show a 60% increase in total recall. MLRefScanner effectively finds refactoring tasks that were missed by cutting-edge machine learning techniques in the past.
4	Almogahed et al. [13]	2023	A Refactoring Classification Framework for Efficient Software Maintenance	Quality Model for Object-Oriented Design (QMOOD) and the classification scheme for refactoring techniques.	A refactoring classification methodology is presented in this study with the goal of improving software systems internal quality attributes. An exploratory investigation, an experimental study, a multicase analysis, and the actual framework construction comprised the four primary stages of the framework development process. Five case studies of different sizes were also chosen for the experiments. In order to help software developers choose appropriate refactoring strategies to enhance the internal quality attributes of software systems, the framework is intended to act as a guide. Through the use of empirical data, the framework lessens the time and effort developers must spend assessing trade-offs and conflicts between refactoring strategies, which lowers the likelihood of incurring expenses and effort related to program maintenance.
5	Alkharabsheh et al. [14]	2022	Prioritization of God class design smell: A multicriteria-based approach.	Design smell prioritization, SpIRIT.	Method of Design Smell Prioritization. In particular, they assumed crucial and developers should be considered for God Class Design Smell based on a mixture and merging of three criteria (Historical Information, Design Smell Density, Developer Context). Subsequently, expert developers from the same software system teams conducted an empirical evaluation of the technique inside the framework of two distinct versions of twenty-four open-source software systems. In addition, the developers were questioned about the criteria they used to rate the God Class Design Smell. The rankings were examined using Spearman's correlation coefficient.

Table 1
(Continued)

Sr. No.	Authors Name	Year	Paper Title	Tool/Technique	Purpose of the Study
6	AbuHassan et al. [15]	2022	A probabilistic-based approach for automatic identification and refactoring of software code smells	Probabilistic graphical model (PGM), A Bayesian network.	This paper presents a probabilistic graphical approach for identifying object-oriented language structural antipatterns. Six distinct code smells—God class, data class, feature envy, complex class, spaghetti code, and speculative generality—are found in six distinct Java programs as part of the evaluation process. All things considered, the suggested model has located the antipatterns.
7	Saheb-Nassah et al. [16]	2022	Prioritization of model smell refactoring using a covariance matrix-based adaptive evolution algorithm	Multiobjective optimization (MOO) algorithm called the multiobjective (MO) covariance matrix adaptation evolution strategy (MOCMA-ES).	The published performance scores attest to the MO single bond CMA-ES algorithm's supremacy over NSGA-II. While it can correct all detected design smells, the former was able to identify the refactoring steps that result in the greatest gains in software quality and maintainability. The improvements are measured in terms of execution time, hypervolume, coverage, and spacing measures.
8	Kaur and Singh [17]	2022	How does Object-Oriented Code Refactoring Influence Software Quality? Research Landscape and Challenges	MOOSE, E MOOSE, NOM	In order to define the current state of the art and identify potential open challenges, this paper reports a thorough systematic mapping study that locates, evaluates, and presents the existing empirical literature regarding the impact of refactoring activities on software quality. First, they offered a succinct an overview of the software refactoring procedure. They later talked about the methods used to conduct this SMS mapping investigation.
9	Gao et al. [18]	2020	A Prototype for Software Refactoring Recommendation System	SRRS (Software Refactoring Recommendation System)	The numerous decision points involved in the software refactoring process mean that either all the proposed solutions are not adequate or all of the recommended methodologies and tools cannot fully cover all the decision points. In order to help programmers restructure decision-making, we create a software refactoring recommendation tool based on prior research. The tool can help users make better decisions about refactoring, which lowers the cost of refactoring, as demonstrated by experimental validation.
10	Sidhu et al. [19]	2022	A machine learning approach to software model refactoring	ML-based UML model	An adaptive supervised learning technique and a deep neural network model are used to achieve the suggested idea. A training dataset annotated with the metric values of a function that labels the diagrams with functional decomposition is inferred using UML class diagrams. After finding flaws, many refactoring processes that deal with functional decomposition are used. An empirical evaluation of the algorithm demonstrates its great accuracy.
11	AlOmar [20]	2021	State of Refactoring Adoption: Towards Better Understanding Developer Perception of Refactoring	Self-Affirmed Refactoring (SAR)	The SAR taxonomy and model can function as a strong foundation for different empirical research and, when combined with refactoring detectors, can indicate any early discrepancies between refactoring types and their documentation. Considering their discoveries regarding the industrial as part of their survey comments, they suggested a process to accurately record refactoring actions in the case study.

Almogahed et al. [13] rely on the strategy that helps reduce and streamline maintenance costs and processes is refactoring. Refactoring approaches' impact on quality criteria, however, shows uneven and contradictory results. As a result, software developers have challenges in successfully improving software quality. Furthermore, the lack of a complete framework makes it more difficult for developers to make decisions about which refactoring approaches are appropriate and in line with particular design goals. Given these factors, the purpose of this study is to present a novel paradigm for categorizing refactoring methods according to how much of an impact they have on internal quality criteria.

Alkharabsheh et al. [14] offered a multiple-criteria merged technique for prioritizing design smells, with a focus on a God Class Designs Smell. They empirically tweak and evaluate the method using a dataset of 24 open-source applications. The empirical judgment seeks to compare the highest-ranked God Class acquired using their proposed technique to the opinions of developers working on each project. The evaluation results indicate that the plan requires additional improvement. They recommend comparing initiatives where respondents' responses match the plan against projects where no link is seen. By doing so, they can refine and enhance the effectiveness of the prioritization approach.

AbuHassan et al. [15] present a novel framework that utilizes graphical models with probabilistic probabilities to detect and refactor antipatterns in software systems. The framework entails producing a graphical model by retrieving class attributes from its source code. In the final stage, a Bayesian network gets trained to detect antipatterns based on the properties of surrounding classes. To assess the effectiveness of our technique, they trained the idea on six distinct counterfeit patterns and applied it to six separate Java projects. The results demonstrate that the suggested model detects these antipatterns with an average precision of 85.16 percent and an average recall of 79%. Overall, the study highlights the importance of effectively prioritizing design smells and suggests avenues for future improvements based on empirical evaluation and feedback from developers.

Saheb-Nassagh et al. [16] used a multiobjective optimization (MOO) optimization method the multi-objective (MO) autocorrelation strategy for adapting and evaluating to prioritize model smell refactoring (MO-CMA-ES). To mitigate the negative consequences of design smells, its derivation is based on reworking unifying modeling language (UML) diagrams that represent classes. To balance refactoring, they utilize two competing goals: efficiency and the ability to be maintained. They begin by establishing a fresh solution depiction that ensures smell removal and removes the denial constraint. Additionally, they suggest a customized mapping strategy for accurately encoding actual values with different renderings.

The study by Kaur and Singh [17] aims to improve the quality of programming, and hence, the final product delivered by the programmer should yield optimal results. The article is introduced to address how object-oriented code modification affects software quality attributes. The plan is to choose 142 primary research that were published up until the end of 2017 through a series of stage inspection methods. The results reveal that studies conducted in academic settings tended to show a greater beneficial influence on software quality of refactoring compared to those carried out in industrial settings. Overall, refactoring activities had varying effects on different quality attributes, causing improvements or degradation in most cases, with the exception of cohesiveness, complexities, inheritance, fault-proneness, and consumption of energy.

Gao et al. [18] created and tested an experimental instrument for a technology refactoring recommendation system. Users interact with the tool to realize their refactoring intentions, and the tool provides them with an optimized software refactoring scheme. The tool has been proven to be effective, particularly for inexperienced and non-English speaking users.

Sidhu et al. [19] proposed design flaws technique would then prevent the vicious spiral of tiny refactoring processes as well as one's pipelined adverse effects. The notion of function decomposition is shown as an anomalous architectural goal and a primary source of object-oriented software design odors. They argue that significant design augmentation can be achieved within a brief quality assurance procedure by refactoring operations targeted at indicators of operational segmentation instead of atomic smells. Their idea was implemented by using a deep network that can recognize the current approach in object-oriented software UML models. Using big data techniques, the strategy described here first gains understanding of intricate and multidimensional application design aspects, and then it uses that understanding to generalize nuanced interactions between the architectural elements.

AlOmar [20] proposed the main findings that the vast majority of PSs experimentally test their approaches, and a few debugging some varieties were examined more frequently than others, also multiple approaches toward behavior preservation have been put forward in the literature, together with the ideas and tactics that guarantee program accuracy while handling refactoring tasks, the suggested automatic analyses, the stick shift assessment technique, and the great majority of PSs empirically access their methods. The current work evaluates the correctness of the transformation and determines whether these methods lead to a safe and dependable refactoring.

3. Contribution

In this paper, first we introduce a novel hybrid optimization algorithm, the HSHEP, specifically designed to address the prioritization of code smell refactoring. This innovative approach combines the strengths of two distinct optimization strategies, setting it apart from existing technologies and providing a unique solution to refactoring challenges. Second, the paper outlines a method to incorporate maintainer opinions and requirements into the optimization process, ensuring that the most critical code smells are prioritized effectively. Finally, the practical applicability and effectiveness of the HSHEP algorithm are validated through its application to various large-scale open-source systems, demonstrating significant improvements in maintenance efficiency and system evolution across five different types of code smells.

4. Hybridization of Algorithm

An algorithm that combines two or more different algorithms to solve the same problem is known as a hybrid algorithm. It may select one approach based on a feature of the data, or it may alternate between the algorithms during the algorithm. Usually, this is done to integrate each of the desired properties, making the whole algorithm superior than its individual parts. A "hybrid algorithm" is a mixture of algorithms that solve the same problem but differ in other aspects, most notably performance. It does not mean mixing various algorithms to tackle a separate problem, although many algorithms may be thought of as combinations of smaller parts.

HSHEP optimizer-based algorithm in software refactoring can improve the effectiveness of the optimization process, combining the advantages of both algorithms. The goal is to address code smells by iteratively applying refactorings to the code, seeking a better solution while avoiding getting trapped in local optima.

4.1. Spotted Hyena Optimizer

Spotted hyenas [21] can become familiar with the site of prey and encircle them. To mathematically depict the movements of spotted Hyenas, the finest search agent, representing the optimum, is the one aware of the prey's location. Other search firms build a network of reliable companions aligned with the pathway of the best search agent. They maintain the best ways obtained thus far to update their current position in pursuit of their objective.

There are four steps considering the working of SHO.

4.1.1. The encircling prey process

It involves the coordinated movement and positioning of multiple individuals around the target, aiming to surround it from all sides. Because of the initial unknown search space, the current leading contender solutions is viewed as the prey's or objective's goal, and it is quite near to the optimum. After determining the most efficient search contender solution, additional search agents are likely to change their positions depending on that optimal solution [22].

The encircling feature is mathematically defined as in Equation (1):

$$Sh\,(r+1) \ = \ Shj\,(u) - P.Pi \tag{1}$$

The space between the prey as well as the spotted hyena is formulated as in Equation (2):

$$Pi \ = \ |H.Shj\,(u) - Sh\,(u)| \tag{2}$$

The current iteration is denoted as 'u', where Shj indicates the vector's position of the prey, and Sh represents the vector's position of the spotted hyena. The operator '/' represents the true value, and the operator "$\cdot$" is implemented to denote vector multiplication. There two vectors are used naming P and H. The P and H are the coefficient vectors, and their mathematical equation can be calculated as in Equations (3) and (4):

$$P = 2.v1 \tag{3}$$

$$H = 2s.v2 - s \tag{4}$$

In this context, the variable "$smax$" denotes the highest recurrence count, and the iteration "s" ranges from 1 to $smax$. The vectors "$v1$" and "$v2$" are randomly generated vectors with values between 0 and 1. In order to strike a balance between exploration and exploitation during the iterations, the value of "s" gradually decreases linearly from 5 to 0 over the course of $smax$. This linear decrease is mathematically represented by the following Equation (5):

$$s \ = \ 5 - (s * (5/\,smax)) \tag{5}$$

4.1.2. Hunting process

In the mathematical description of spotted hyena behavior, it is assumed that the most effective search agent, representing the optimum, possesses knowledge about the prey's position.

The remaining search agents create a cluster that functions as an established circle of friends, and they orient themselves toward the most effective search agent. These agents store the ideal solutions they have found so far and update their positions based on this information.

The hunting process is represented by Equations (6)–(8):

$$Pi \ = \ |H.Shi - Shl| \tag{6}$$

$$Shl \ = \ Shi - V.Pi \tag{7}$$

$$Ui \ = \ Shl + Shl + 1 + \ldots + Shl + Y \tag{8}$$

Here, the best spotted hyena's position is represented as "Shi," while the positions of the other hyenas are denoted as "Shl." Additionally, the cluster containing "Y", in the present search space, the number of optimum solutions is referred to as "V."

Y represents the number of spotted hyenas and is computed in Equation (9).

$$Y \ = \ Zf\,(Shi,\ Shi + 1,\ Shi + 2,\ \ldots(Shi\ +\ A)) \tag{9}$$

A is the arbitrary vector inside [0.5, 1], f is the total amount of solutions, and Z is all possible solutions.

4.1.3. Attacking the prey process

We develop a parameter labeled "s" in order to mathematically represent the exploitation process, which entails attacking the target. During the iteration phase, this parameter is crucial in lowering the value of s from 5 to 0. Additionally, we adjust the V vector to reduce changes and let s steadily decline. The spotted hyena moves closer to the prey, signaling the start of the attacking phase in the equation, when the magnitude of vector V, indicated by |V|, falls below 1.

$$Sh\,(s+\ 1) \ = \ Ui\,Y \tag{10}$$

In Equation (10), $Sh(s + 1)$ refers to the process of updating the other search agent depending on the location of the best answer.

4.1.4. Searching the prey process

The location of the group of spotted hyenas in the vector Ui determines where they go in quest of prey. To search for and assault animals or food, they disperse wider. The search agents are under pressure to travel further away from the prey by altering the vector V with random values greater than 1 and less than −1. On the other hand, |V| > 1 then, the spotted hyenas are enticed to approach the prey. This process makes it easier to find global solutions.

4.2. Emperor Penguin Optimizer (EPO)

To begin with, Emperor penguins randomly establish the perimeter of their huddle. Simultaneously, the distance between emperor penguin is calculated, which aids in further exploration and exploitation. Subsequently, the optimal solution, known as the effective mover, is determined. This solution guides the search agents to better positions. Using the updated emperor penguin positions or search agents, the huddle boundary is then recomputed to enhance the overall effectiveness of the process.

There are four steps considering the working of EPO.

4.2.1. Achieve and gather complete chaos border

While huddling, emperor penguins typically arrange them along the boundary of a polygon-shaped grid. Each emperor penguin has at least two neighboring penguins, randomly selected. The flow of wind within the huddle is examined to determine the chaotic boundaries precisely. Notably, the wind speed is greater than that of an individual emperor penguin. To characterize the randomly produced chaotic border of emperor penguins, complex variables are used.

Let € represent the speed of the wind and £ the gradient of €.

$$€ \ = \ μΦ \tag{11}$$

The complex potential is generated by combining vectors ¥ and £.

$$G = \Phi + j\yen, \tag{12}$$

where *j* is the imaginary constant and G is an analytical function on the polygon plane.

4.2.2. Temperature around gathering

The major goal of the emperor penguin' pandemonium is to preserve energy and maximize the temperature within them. To mathematically depict this situation, we assume the following assumptions: When the radius of the polygon, represented as *R*, exceeds 1, the temperature U is set to 0. However, when the radius decreases and becomes less than 1, the temperature Te is assigned a value of 1. This temperature profile plays a crucial role in driving the exploration and exploitation processes for the emperor penguins across various locations within the chaos.

$$Te' = Te - Max\,Iterations \tag{13}$$

$$y - Max\,Iterations$$

$$Te = \begin{cases} 0 & \text{if } R > 1 \\ \text{Or} & \\ 1 & \text{if } R < 1 \end{cases}$$

where *y* explains the current iteration, Max Iterations represent the maximum number of iterations, *R* is the radius, and *Te* is the moment to identify the best possible outcome in a search space.

4.2.3. The separation between emperor penguins

After producing the cluster border, the distance among the emperor penguin and the best-found optimum solution is determined. The current best ideal solution is the one with the fitness value closest to the optimum. The remaining search agents, or emperor penguins, will modify their placements based on the information from the current optimal solution.

$$Dep = Abs\,(S.A\,P. - C.Pep\,x) \tag{14}$$

The variable "*Dep*" reflects the difference between the emperor penguin and the fittest best search agent (i.e., the most fit emperor's penguins with the lowest fit value) in iteration "*Y*". The variables "*A*" and "*C*" are utilized to prevent collisions between neighboring emperor penguins. "*P*" denotes the best optimal solution, which corresponds to the fittest emperor penguin, and "*Pep*" represents the emperor penguin's position vector. "*S*" represents the social influences that influence emperor penguins, pushing them toward the most ideal search agent.

4.2.4. Relocating the mover

The locations of emperor penguins have been modified using the best-found optimum solution, known as the "mover." This movement is in charge of changing the locations of other search agents in the search space, and it will leave its present position unoccupied during this update. To determine an emperor penguin's future position, the following equation is suggested:

$$P\,x\,(x + 1) = Px\,A.\,Dep \tag{15}$$

where *Px* (*x* + 1) indicates the emperor penguin's most recent position update. The clustered habits of emperor penguins are recalculated throughout the iteration phase once the moving object has been repositioned.

4.3. Proposed hybrid HSHEP algorithm

The hybridization of SHO and EPO two algorithms involves combining their principles, mechanisms, or strategies to create a new hybrid optimization approach. The aim is to leverage the strengths of both algorithms, leading to potentially improved exploration and exploitation capabilities, and achieving better performance in solving challenging optimization problems [23, 24]. Refactoring is a software development technique that modifies current code to enhance its structure, readability, and maintainability while retaining its exterior behavior. Let us create an algorithm for refactoring the fictional "hybrid spotted hyena and emperor penguin" algorithm we previously defined. This refactoring aims to enhance the code's structure, clarity, and modularity. Since the previous algorithm was also fictional, we will assume that it is written in a high-level language like Python for simplicity.

Algorithm for refactoring sequencing the HSHEP.

4.3.1. Initialization phase

Initialize the population using random solutions. Set settings such as population size, maximum iterations, crossover rate, mutation, and local search rate. Determine the initial best solution in the population using the fitness function.

4.3.2. Main optimization loop

Repeat for a predefined number of iterations:

1) **Hunt and capture phase** (spotted hyena behavior)

Create offspring individuals by performing crossover and mutation operations on the population. Add the offspring to the population.

2) **Information sharing phase** (spotted hyena behavior)

Share information between individuals to enhance global search capabilities.

3) **Huddling phase** (emperor penguin behavior)

Sort the population based on fitness and perform local search around the top individuals. Add the local search results to the population.

4) **Individual movement phase** (emperor penguin behavior)

Exploring new areas of the search space can lead to the development of new solutions. Introduce the new solutions to the population.

1) **Selection**

Select the best individuals from the population based on fitness, keeping the population size constant.

2) **Update the best solution**

If a better solution is discovered during the iteration, update the best solution accordingly.
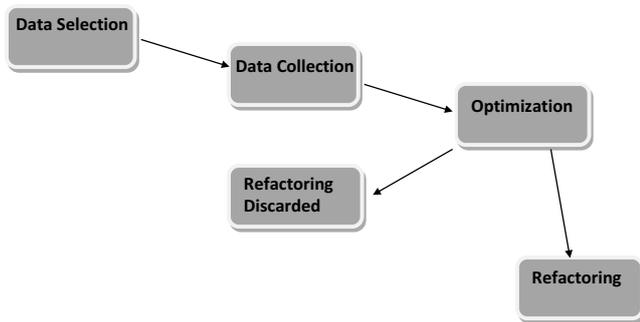
4.3.3. Termination

When the maximum number of iterations is achieved or another stopping requirement is fulfilled, the main loop ends.

4.3.4. Bring back the best solution

Return the best solution discovered throughout the optimization process as the ideal answer to the problem.

Figure 1
Proposed methodology of code smell refactoring sequencing

In this theoretical description, each step outlines the main actions and phases performed in the Hybrid Spotted Hyena and Emperor Penguin Optimization (HSHEPO) algorithm [25–27]. The specific details and implementation of functions may vary based on the problem domain and the algorithm's requirements. The algorithm combines emperor penguin social dynamics and spotted hyena hunting behavior to improve optimization outcomes.

4.4. Research methodology

Prior to elaborating on the suggested framework for refactoring prioritization, we will now discuss the proposed research methodology, outlined in figure given below [28]. The primary objective of this paper is to explore the potential analogies between the behaviors and characteristics of emperor penguins with spotted hyenas and the process of refactoring code sequences. The methodology comprises four phases, each of which is introduced in Figure 1:

1) Phase 1

Data selection: The data selection for performance evaluation and analysis involves choosing different open-source and publicly available Java datasets. These different datasets will be used to assess and analyze the proposed approach.

2) Phase 2

Data collection: The data and code smells are being gathered. A software tool is built to extract metrics such as the number of classes coupled with the main class (CBO), the relationship between various methods and parameters of the class (LCOM), the complexity of the class (RFC), the maintainability value, and the complexity of the methods. Five code smells, including a lengthy argument list, long function, blob, feature envy, and huge class, have been recognized as needing to be refactored. Table 2 provides detailed measurements and descriptions.

Table 2
Dataset metrics

System	#Classes	#Bad Smells
JHotDraw	753	42
GanttProject	245	67
JEdit	184	51
JFreeChart	521	82
Xerces	991	106

3) Phase 3

Optimization: The hybrid spotted HSHEP serves as the foundation for identifying the most effective refactoring sequences to completely eliminate the identified design smells. By drawing inspiration from the resourcefulness of spotted hyenas and the collective intelligence of emperor penguins, the algorithm proposes creative strategies to tackle code smells and optimize the refactoring process as shown in Table 3. These metaphorical mappings guide developers in selecting the optimal refactoring actions that result in the successful eradication of design smells from the code.

Table 3
Open-source program parameters

Dataset Name	Dataset Type	Total Number of Classes	Total Number of Smells
JHotDraw	Open Source	398	1215
GanttProject	Open Source	776	63
JEdit	Open Source	5421	1365
JFreeChart	Open Source	3257	793
Xerces	Open Source	2551	5989

4) Phase 4

Refactoring: During this phase, refactoring tasks are performed based on the sequences generated in the previous phase. To enhance user involvement, our method provides software engineers with three alternatives for dealing with side effects. They can choose to address all newly discovered odors, rearrange the sequence of smells, or ignore the new smells entirely [29, 30]. These choices empower engineers to make informed decisions while maintaining control over the refactoring process. The programme statics of different smells are shown in Table 4.

Table 4
Programme statistics

System	#Classes	#Bad Smells
JHotDraw	753	42
GanttProject	245	67
JEdit	184	51
JFreeChart	521	82
Xerces	991	106

5. Computational Complexity

5.1. Time complexity

The time complexity of the HSHEP optimization-based algorithm is primarily influenced by the individual complexities of the Spotted Hyena Optimization (SHO) and Emperor Penguin Optimization (EPO) algorithms, as well as the operations required to combine them effectively.

The SHO algorithm typically exhibits a time complexity of $O(n2)$, where n is the number of hyenas or the population size. This complexity arises from the need to evaluate the fitness of each individual and update their positions iteratively based on the best solutions found.

Similarly, the EPO algorithm has a time complexity of $O(n * m)$, where n is the population size and m is the number of

iterations. This results from the evaluation of fitness and position updates based on the emperor penguin behavior model.

When these two algorithms are hybridized in the HSHEP approach, the combined time complexity can be approximated as $O(n2 + n * m)$. The exact complexity depends on how the two algorithms are integrated. If the processes are sequential, the complexity would be the sum of the individual complexities. If they run in parallel or have overlapping operations, the effective complexity could be closer to the higher of the two.

Additionally, the integration mechanism in HSHEP, which involves hybridizing the search and optimization strategies, adds an overhead. Assuming a linear combination, the complexity remains dominated by the higher-order term, which is $O(n2)$ in this context, considering $m <= n$. Therefore, the overall time complexity of the HSHEP algorithm can be approximated as $O(n2)$, ensuring efficient performance for large-scale optimization problems with a manageable population size.

6. Experimental Results and Discussions

The initial step involved a comparison of the approach with two alternative algorithms that do not incorporate prioritization—specifically, SHO without prioritization and the method proposed by EPO. In this comparison, a fitness function is used to quantify the number of rectified code-smells. This was done to assess the efficacy of using prioritizing in the schema.

The extended assessment compares the outcomes of the SHO approach to three well-known metaheuristics: PSO, GA, and the suggested algorithm. During this stage of the study, the same fitness function is used to validate the utility of the SHO method.

Given the intrinsic randomized behavior of the algorithms and techniques being investigated, it is crucial to remember that the findings may differ somewhat with each run. To account for this unpredictability and back up our deductive statistical assertions, the experimental research includes 31 unique runs of simulation for each method and approach examined.

To determine the significance of the findings, they used the Wilcoxon ranking summation test, as described in PSO and GA. This test was used to evaluate the SHO-based strategy against all the other strategies investigated in the study. Another important point to note is that the bulk of unresolved code smells detected in both SHO without prioritization and EPO are of the "large class" type [31, 32]. This type of code smell often needs numerous refactoring operations and is extremely difficult to resolve without the introduction of a specialized strategy, such as priority.

Conversely, the computed Code-Smell Correction Ratio (CCR) score associated with the "blob" code-smell is acceptable, averaging at 79% across all systems. However, it is worth noting that this score falls short of the scores achieved by both of the other approaches. The deficiency in the correction ratio for data classes is considerably compensated by the substantial improvements observed in terms of importance, risk, and severity scores.

This lower score is primarily attributed to the fact that, in the experiments, data classes are not prioritized; they are assigned the least priority score, set to 1, in contrast to the treatment of blob code-smells. This score allocation is in accordance with developer preferences. Moreover, data classes, in general, undergo infrequent alterations during development and maintenance, primarily containing data and predominantly composed of setters and getters functions, which involve minimal processing of data.

As a result, the significance score associated with this code smell is fairly low. In contrast, as shown in Table 5, all discovered "large" code-smells are properly handled, resulting in a perfection

rate of more than 90%. As a result, their relevance score is significantly higher, making them more deserving of prioritizing.

Furthermore, to ensure the effectiveness and applicability of our technique, we manually verified the likelihood of the different recommended refactoring sequencing for every system.

They're encountered certain semantic errors in the program's behavior. When identifying such errors manually, the classified operations associated with these changes as suboptimal recommendations [33]. To assess the success of an approach, compute an accuracy precision outcome, which is the ratio of potential refactoring actions. On average, 90% of the refactorings are deemed feasible. This score is in line with those achieved by both of the other approaches as shown in Table 6.

However, it is worth mentioning that our results for data classes are somewhat less favorable than those obtained by other methodologies. Generally, this type of code smell is less critical and poses a lower risk than some other code smells, requiring less extensive corrective action by software engineers, especially when compared to code blobs. To fix difficulties with data classes, program maintainers can simply use refactorings such as inlining classes, shifting methods/fields to offer new behaviors/functionality, or combining data classes with existing classes in the system. Although the technique does not explicitly select data classes, we were able to obtain an acceptable rectification score. This is partly attributed to the fact that code blobs are often associated with data classes [34–36]. Consequently, addressing code blobs can indirectly resolve issues related to their associated data classes. Experimental results on different datasets are shown in Figures 2–6.

Furthermore, there were good findings in terms of importance, risk, and severity correction ratings. The majority of the important, high-risk, and severe code smells were successfully handled, and over 90% of the recommended refactoring sequences were semantically consistent.

6.1. Fitness function

The pursuit of optimal solutions is driven by a single fitness. The primary objective is to maximize the system's overall quality. By doing so, the approach becomes cost-effective; as it effectively minimizes cost of the rework that may arise in the future. The primary objective of any software system, from a software engineering standpoint, is to achieve exceptional software quality. This focus on software quality has been a prominent area of interest in the software engineering field for many decades [34, 35]. Among the various activities in the software development lifecycle, software maintenance and evolution stand out as the costliest, accounting for more than 75% of the development expenses [36]. Consequently, the presence of software smells poses a potential risk for future software maintenance [37]. However, by detecting and correcting these software smells through refactoring, the system's overall quality can be significantly enhanced proactively, before these undesirable traits spread to other phases and incur additional maintenance costs [38].

$$Fitness(r) = \sum_{i=0}^{n-1} \left(yi * (\delta * Severity(ci) + \gamma * Priority(ci) \right. \\ \left. + \beta * Risk(ci) + \alpha * Importance(ci)) \right) \quad (16)$$

Equation (1) enables the computation of the efficacy of the refactoring solution denoted as "w." In this formula, "yi" is assigned a value of 0 if the current class is identified as a code smell using the code smell detection criteria, and 1 if no code smell is found. The parameters $\delta, \gamma, \beta,$ and α collectively sum up to 1, reflecting the confidence or significance (weight) attributed to each individual

Table 5
Refactoring results: code smell

| System | Approach | Code Smell Correction Ratio | | | | |
		Long Method (%)	Feature Envy (%)	Blob (%)	Long Parameter List (%)	Large Class (%)
JHotDraw	SHO	91(5/9)	89(15/4)	90(3/4)	89(4/9)	92(9/11)
	EPO	91(6/9)	91(2/4)	89(2/4)	91(5/9)	90(8/11)
	PSO	93(5/9)	92(3/4)	90(3/4)	92(7/9)	91(8/12)
	GA	94(6/9)	91(2/4)	90(3/4)	90(4/9)	89(6/11)
	Proposed Algorithm	92(7/9)	92(47/4)	93(4/4)	93(8/9)	92(10/11)
GanttProject	SHO	91(4/7)	93(7/9)	93(6/7)	87(11/19)	90(14/18)
	EPO	92(5/7)	92(6/9)	92(5/7)	91(16/19)	91(15/18)
	PSO	94(6/7)	90(5/9)	92(5/7)	92(17/19)	93(15/18)
	GA	94(4/7)	92(6/9)	90(6/7)	92(17/19)	90(14/18)
	Proposed Algorithm	93(6/7)	93(8/9)	93(7/7)	94(19/19)	93(16/18)
JEdit	SHO	90(197/27)	90(16/20)	90(20/27)	89(17/22)	91(20/27)
	EPO	89(17/27)	91(16/20)	91(21/27)	91(18/22)	90(16/27)
	PSO	91(19/27)	92(17/20)	91(20/27)	92(20/22)	90(20/27)
	GA	90(19/27)	89(15/20)	90(16/27)	93(21/22)	91(21/27)
	Proposed Algorithm	93(20/27)	92(18/20)	92(22/27)	93(21/22)	92(22/27)
JFreeChart	SHO	92(15/17)	87(20/27)	92(21/26)	87(7/14)	91(14/16)
	EPO	91(14/17)	91(24/27)	89(16/26)	90(11/14)	92(15/16)
	PSO	92(15/17)	92(25/27)	91(19/26)	91(12/14)	92(15/16)
	GAProposed	89(9/17)	93(25/27)	90(19/26)	90(119/14)	89(9/16)
	Algorithm	92(15/17)	94(27/27)	93(24/26)	92(612/14)	92(15/16)
Xerces	SHO	88(17/29)	91(63/72)	89(27/31)	87(17/29)	91(26/31)
	EPO	92(21/29)	92(64/72)	91(26/31)	91(23/29)	91(25/31)
	PSO	91(20/29)	89(57/72)	91(25/31)	91(24/29)	92(26/31)
	GA	91(21/29)	91(62/72)	92(26/31)	90(21/29)	89(27/31)
	Proposed Algorithm	93(23/729)	92(65/72)	93(29/31)	93(25/29)	93(29/31)
Average		92	93	93	93	92

Table 6
Refactoring results: importance, risk, severity, and refactoring precision scores

Systems	Approach	Importance	Risk	Severity	Precision Score
JHotDraw	SHO	89	90	87	89
	EPO	76	76	76	75
	PSO	72	72	73	74
	GA	61	61	61	67
	Proposed algorithm	57	53	59	56
GanttProject	SHO	89	92	87	93
	EPO	77	77	76	76
	PSO	71	72	73	76
	GA	61	63	67	64
	Proposed algorithm	59	66	87	59
JEdit	SHO	88	94	90	91
	EPO	77	76	77	78
	PSO	72	72	73	74
	GA	65	61	69	65
	Proposed algorithm	56	59	57	59

Table 6
(*Continued*)

Systems	Approach	Importance	Risk	Severity	Precision Score
JFreeChart	SHO	88	92	87	89
	EPO	77	76	76	78
	PSO	73	72	72	74
	GA	61	61	65	65
	Proposed algorithm	55	57	58	57
Xerces	SHO	89	93	89	90
	EPO	77	76	76	87
	PSO	75	74	72	75
	GA	68	65	61	64
	Proposed algorithm	59	55	54	57

Figure 2
Experimental results on Gantt Project using proposed and competitor approaches
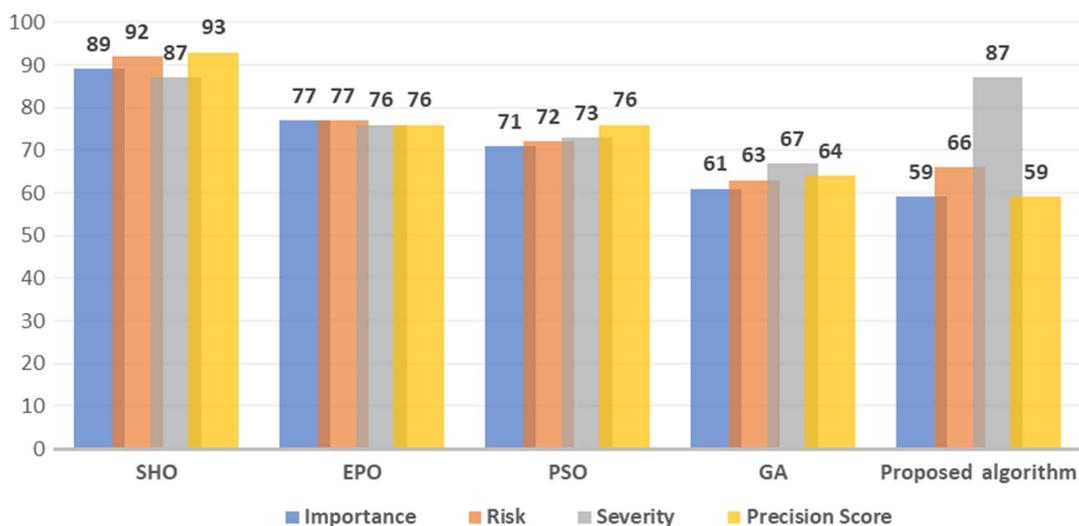
Figure 3
Experimental results on JEdit using proposed and competitor approaches
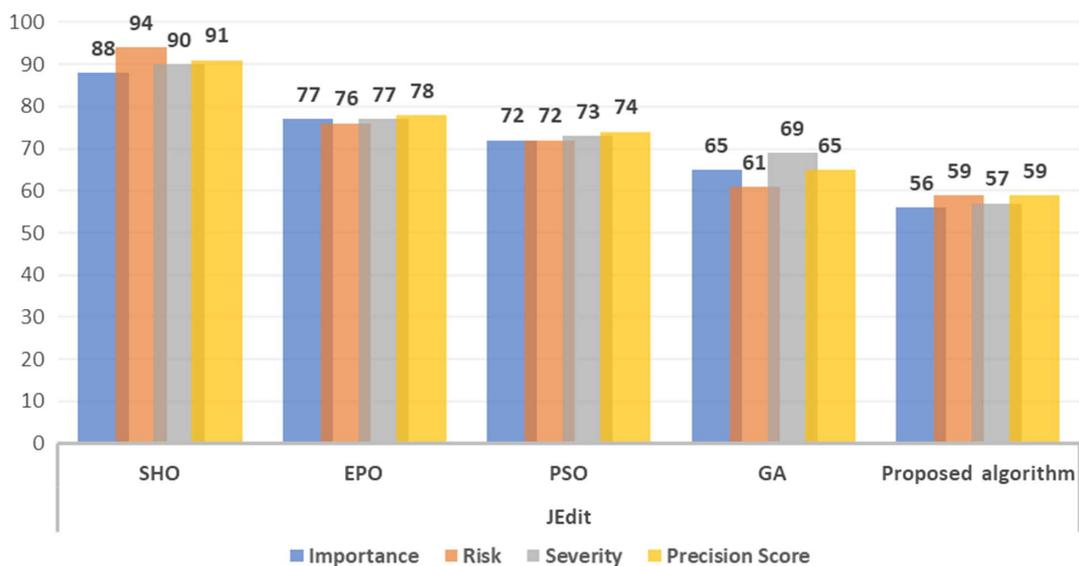
Figure 4
Experimental results on JHotDraw using proposed and competitor approaches
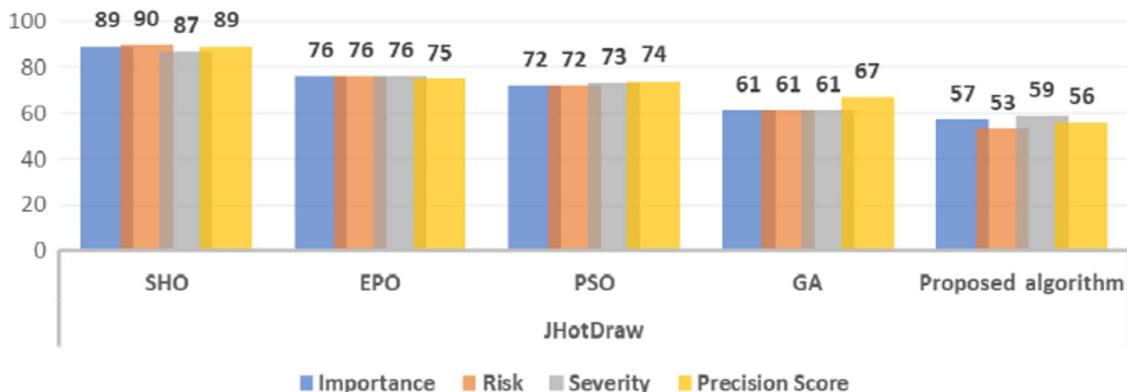
Figure 5
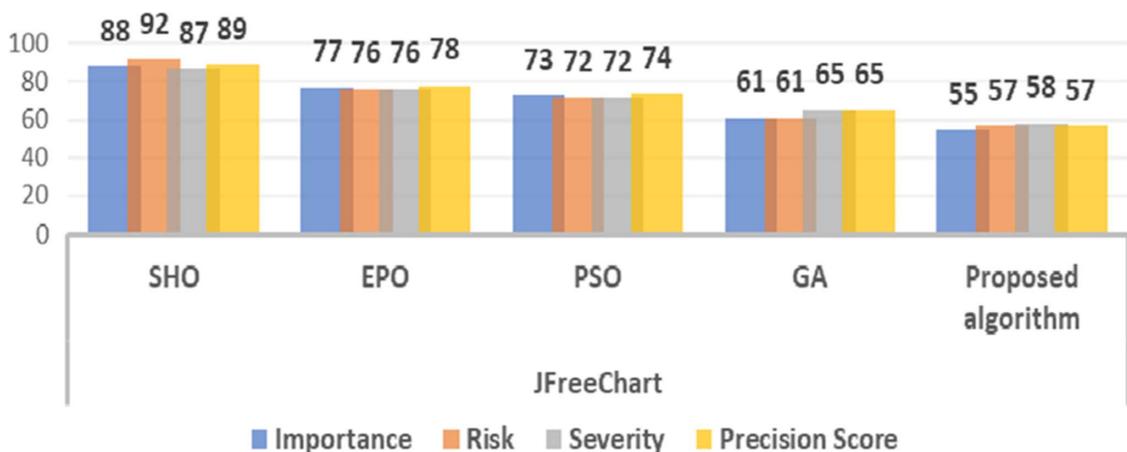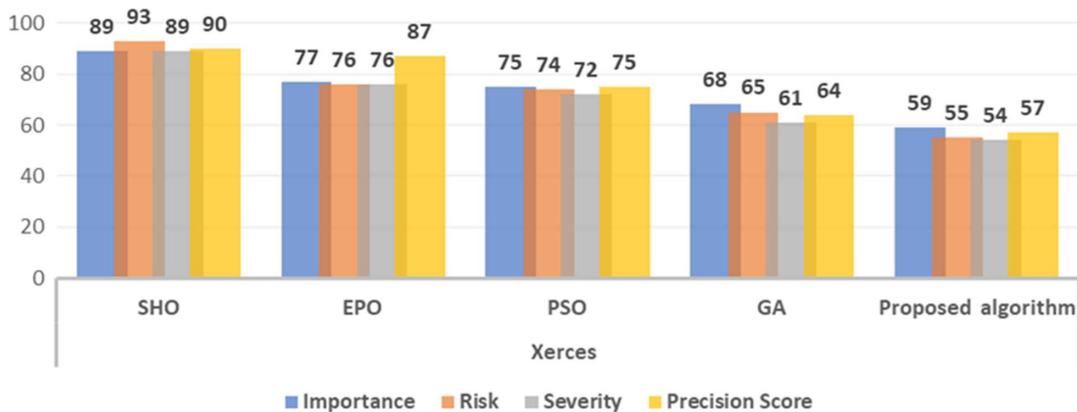**Experimental results on JFreeChart using proposed and competitor approaches**

Figure 6
Experimental results on Xerces using proposed and competitor approaches

measure. These weights can be adjusted based on developer preferences. A series of comprehensive experiments were conducted, involving diverse combinations of weights for each prioritization measure. In the specific case of equal weights, each measure was assigned a weight of 0.25.

Subsequently, here we present a set of 4 prioritizing criteria (severity, priority, risk, and importance) that is included into the methodology for resolving code smells during refactoring.

Severity: In practical scenarios, the influence or significance of code smells is not uniform across all instances. Each specific occurrence is associated with a severity rating that enables designers to promptly identify and address the most crucial occurrences of each code smell. More precisely, identical types of code smells can manifest in various code segments, each carrying distinct impact scores on the system's architecture. These impact scores gauge the relative magnitude of the code smell, encompassing both its comparative severity and the absolute detrimental effect on the overall system quality.

Priority: Developers commonly assign varying degrees of significance to distinct types of code smells, which can have diverse implications for the overall quality of the system. By prioritizing detected code smells based on their individual preferences, developers can arrange these types in a ranked order. This prioritization approach empowers designers to optimize time utilization and enhance the effectiveness of resource allocation for maintenance tasks within their software projects.

Risk: An essential factor to take into account is the vulnerability score. Therefore, this was posited that as code strays further from established best practices, its susceptibility to risk increases. Consequently, during the correction phase, the code smells posing the highest level of risk should be given precedence. Each identified code smell is accompanied by a vulnerability score, reflecting the degree of departure from well-structured code design.

Importance: Typically, developers require an understanding of the key code segments (such as classes and packages) within the entire software system to effectively direct their efforts toward enhancing their excellence. Within a standard software package system, pivotal code segments are those that undergo frequent modifications throughout the development and maintenance stages, facilitating the addition of new features, adaptation to changes, and overall enhancement of the software's structural integrity [39, 40].

The graphical representations of refactoring results on different types of the open-source software JHotDraw, JFreeChart, JEdit, Gantt Project, and Xerces are given below in Figure 7.

6.2. Objective of maintainability

Maintainability of programs is the ease with which software systems may be changed. It is a critical software quality factor that directly influences the cost of software development. Therefore, to effectively manage software development costs, it becomes imperative to evaluate the maintainability of software systems [41]. To achieve this, a metrics-based approach will be employed, which is commonly utilized for predicting and estimating quality attributes, specifically to assess the effect of fixing code smells. In our scenario, the system's maintainability will be assessed following the implementation of a full refactoring process.

Software maintainability can be separated into five key features:

1) Modularity: The software is organized into separate and cohesive modules, allowing for easier understanding and changes to be made to individual parts without affecting the whole system.
2) Readability: The code is well documented and written in a clear, understandable manner, enabling developers to comprehend its functionality and make modifications efficiently.
3) Testability: The software is designed to be easily testable, with well-defined test cases and test environments, facilitating the identification and resolution of defects.
4) Extensibility: The software is constructed in a way that allows for straightforward addition of new features or functionalities without major code changes or disruptions to the existing system.
5) Reusability: The code is structured in a manner that promotes the reuse of modules or components in other parts of the software or in different projects, leading to increased efficiency and reduced development effort.

By adhering to these characteristics, software developers can create maintainable systems that are more adaptable to changes, have fewer defects, and are more cost-effective to manage over their lifecycle [42].

Figure 7
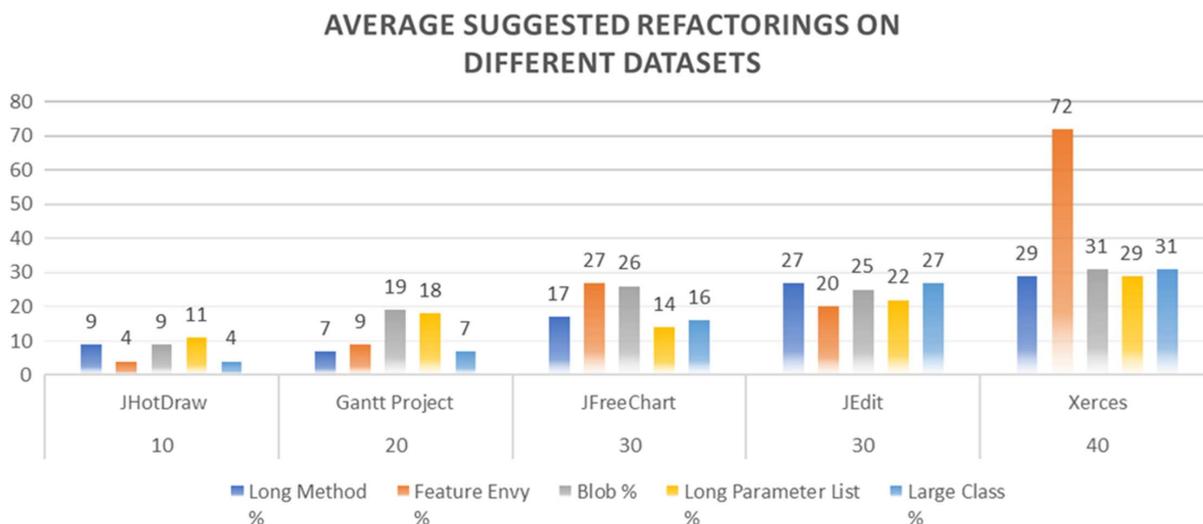**Experimental results on all datasets using proposed and competitor approaches**

Figure 8
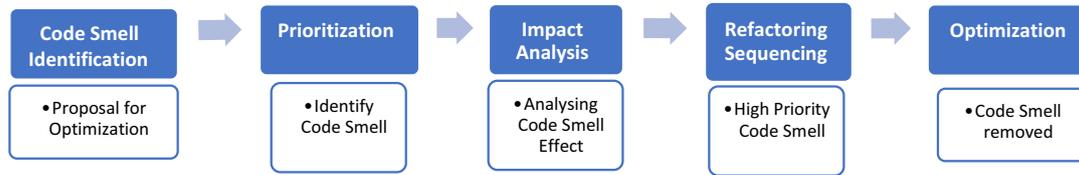**Code smell refactoring sequencing technique**

Table 7
The five different code smell are used

Code Smell	Illustration	Refactoring Technique
1. Long Method	A method containing more than 10 lines of code (LOC) is classified as a long method.	Extract Method
2. Feature Envy	A method shows more attention or dependencies toward other classes rather than the class in which it is declared.	Move Method
3. Blob	A class or method becomes excessively large and takes on multiple responsibilities.	Extract Class or Extract Method
4. Long Parameter List	A method has a large number of parameters, which can make the method's signature cumbersome and hard to manage	Replace Parameter
5. Large Class	A class becomes overly large, containing a significant number of attributes and methods	Extract Class or Sub Class

6.3. Refactoring techniques

Refactoring is the practice of reorganizing software code to reduce unpleasant odors while maintaining its exterior functionality. Code smell leads to errorless coding in the programmes, hence these code smells should be removed by refactoring. The refactoring sequencing technique takes place in different steps as shown in Figure 8. "The process of refactoring involves modifying the internal attributes of the software while preserving its external behavior [43]. The primary objective is to eliminate code inefficiencies and bad smells. Below are some of the techniques employed in this study" as shown in Table 7.

In this study, the focus is basically on five code smells and HSHEP technique is employed to choose the top optimized solution for refactoring sequencing in order to remove these smells:

1) Long method: A long method is a code smell. When a method is implemented in a long method of code, it smells. Excessively long, containing numerous lines of code and performing multiple tasks. This can make the method challenging to comprehend, test, and modify.
2) Feature envy: The code smell known as "feature envy" takes place when a procedure is more focused on another object's data than on its own. In other words, a method is excessively using the data and behavior of another class, indicating a potential design issue.
3) Blob: It occurs when a class or module becomes excessively large, complex, or has too many responsibilities, resulting in a monolithic and difficult-to-maintain structure.
4) Long parameter list: It is a code smell that occurs when a function or method has an excessive number of parameters. This code smell can make the code harder to read, understand, and maintain. It is typically an indication that the function may be taking on too many responsibilities or that its interface could be improved.

5) Large class: It is a code smell that occurs when a class becomes excessively large, typically due to having too many methods, attributes, or responsibilities. Large classes can be challenging to understand, maintain, and extend, and they often violate the principles of encapsulation and single responsibility.

6.4. Potential facts

This section addresses the threats to the validity of the prioritization detection technique, discussing various factors that could potentially impact the results.

Constructional accuracy: This hazard refers to probable mistakes in the model's scent detection rules. Although we employed highly accurate, precise, and recall-oriented rules for detection, there remains a possibility that some model smells were not identified or that certain detected smells were not genuine. The used model odors detection approach may generate false positives, influencing the results of our research.

Inner authenticity: This risk assesses the possibility that the quality estimate and maintenance calculator used will really achieve their intended aims [44, 45]. We utilized the number of refactoring possibilities as a measure of the effort required to convey model quality and improve the system, which is consistent with past research. For sustainability, we follow the ISO/IEC 25,010 standard. However, it should be noted that these indicators were chosen based on their utilization in relevant research, which may have an impact on internal validity [46].

External validity: This danger examines the generalizability of the findings. In our studies, we used data from seven open-source apps covering multiple areas and sizes. Nonetheless, further research may be conducted to improve the generalizability of the findings [47].

Addressing these validity threats allows for a more comprehensive understanding of the prioritization detection technique and

the robustness of its results [48]. By acknowledging and considering these potential limitations, we can gain greater confidence in the outcomes of the study.

7. Conclusion

This article proposes a unique hybrid strategy based on the HSHEP algorithm that uses single strategies for objective optimization. The refactoring of design smells is handled by the solutions, in contrast to existing refactoring methodologies. A five-point evaluation of the HSHEP priority algorithm. The optimum refactoring order, which results in the highest quality with the least amount of maintainability, may be made using a variety of model smells. A sizable bespoke dataset was used to evaluate the effectiveness of the chosen refactoring strategies are created along dataset which includes more 29,000 class records, using five well-known open-source software initiatives. This article also provided the performance data for benchmarking reasons represents an enhanced variation of the current emperor penguin and spotted hyena. The proposed approach can fix all reported pattern smells while also identifying the restructuring sequencing that result in the greatest improvements in software reliability and maintainability. These gains are measured in terms of execution time, hyper volume, coverage, and spacing parameters. According to the given results, the effective prioritization process successfully rectified all detected design smells. The HSHEPI algorithms revealed 2871 and 2856 refactoring possibilities utilizing 100-length sequences after 1000 iterations, in contrast to current single objective solutions. The method is based on a unique relative accuracy coverage statistic, which compares all solution points on average. To address additional model and code smells, the HSHEP applied an objective optimization strategy to achieve the highest average maximum quality score of 1149. In addition, the HSHEP technique will be used to address challenges in other domains, which include design pattern prioritizing or defect repair priority, as well as the proposed mapping scheme between actual numbers and solution representations. Finally, we want to construct a stand-alone tool that includes the proposed refactoring solutions. Future aims involve turning the proposed refactoring techniques into a stand-alone tool for practical use. This study will also look at how the HSHEP method may be used to solve issues in a variety of fields, exploiting its benefits in terms of software quality and maintenance.

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

References

[1] Acharya, B., & Sahu, P. K. (2020). Software development life cycle models: A review paper. *International Journal of Advanced Research in Engineering and Technology, 11*(12), 169–176.

[2] 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

[3] Jerzyk, M., & Madeyski, L. (2023). Code smells: A comprehensive online catalog and taxonomy. In N. Kryvinska, M. Greguš, & S. Fedushko (Eds.), *Developments in information and knowledge management systems for business applications* (Vol. 7, pp. 543–576). Springer. https://doi.org/10.1007/978-3-031-25695-0_24

[4] Kaur, M., & Singh, D. (2022). An intelligent code smell detection technique using optimized rule-based architecture for object-oriented programmings. In *International Conference on Artificial Intelligence and Sustainable Engineering, 1,* 349–363. https://doi.org/10.1007/978-981-16-8542-2_27

[5] Traini, L., di Pompeo, D., Tucci, M., Lin, B., Scalabrino, S., Bavota, G., ..., & Cortellessa, V. (2021). How software refactoring impacts execution time. *ACM Transactions on Software Engineering and Methodology, 31*(2), 25. https://doi.org/10.1145/3485136

[6] Agnihotri, M., & Chug, A. (2020). A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems, 16*(4), 915–934. https://doi.org/10.3745/JIPS.04.0184

[7] Johnson, R. E., & Opdyke, W. F. (1993). Refactoring and aggregation. In *International Symposium on Object Technologies for Advanced Software: First JSSST International Symposium,* 264–278. https://doi.org/10.1007/3-540-57342-9_78

[8] Mu, L., Sugumaran, V., & Wang, F. (2020). A hybrid genetic algorithm for software architecture re-modularization. *Information Systems Frontiers, 22*(5), 1133–1161. https://doi.org/10.1007/s10796-019-09906-0

[9] Alotaibi, M. (2018). *Advances and challenges in software refactoring: A tertiary systematic literature review.* Master's Thesis, Rochester Institute of Technology.

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

[11] Jain, S., & Saha, A. (2024). Improving and comparing performance of machine learning classifiers optimized by swarm intelligent algorithms for code smell detection. *Science of Computer Programming, 237,* 103140. https://doi.org/10.1016/j.scico.2024.103140

[12] Noei, S., Li, H., & Zou, Y. (2025). Detecting refactoring commits in machine learning python projects: A machine learning-based approach. *ACM Transactions on Software Engineering and Methodology, 34*(3), 1–25. https://doi.org/10.1145/3705309

[13] Almogahed, A., Mahdin, H., Omar, M., Zakaria, N. H., Mostafa, S. A., AlQahtani, S. A., ..., & Hidayat, R. (2023). A refactoring classification framework for efficient software maintenance. *IEEE Access, 11,* 78904–78917. https://doi.org/10.1109/ACCESS.2023.3298678

[14] Alkharabsheh, K., Alawadi, S., Ignaim, K., Zanoon, N., Crespo, Y., Manso, E., & Taboada, J. A. (2022). Prioritization of god class design smell: A multi-criteria-based approach. *Journal of King Saud University-Computer and Information Sciences, 34*(10), 9332–9342. https://doi.org/10.1016/j.jksuci.2022.09.011

[15] 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

[16] Saheb-Nassagh, R., Ashtiani, M., & Minaei-Bidgoli, B. (2022). A probabilistic-based approach for automatic identification and refactoring of software code smells. *Applied Soft Computing, 130,* 109658. https://doi.org/10.1016/j.asoc.2022.109658

[17] Kaur, S., & Singh, P. (2019). How does object-oriented code refactoring influence software quality? Research landscape and challenges. *Journal of Systems and Software, 157,* 110394. https://doi.org/10.1016/j.jss.2019.110394

[18] 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

[19] 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

[20] AlOmar, E. A. (2021). *State of refactoring adoption: Towards better understanding developer perception of refactoring*. PhD Thesis, Rochester Institute of Technology.

[21] Vissia, S., Wadhwa, R., & van Langevelde, F. (2021). Co-occurrence of high densities of brown hyena and spotted hyena in central Tuli, Botswana. *Journal of Zoology, 314*(2), 143–150. https://doi.org/10.1111/jzo.12873

[22] McCormick, S. K., Holekamp, K. E., Smale, L., Weldele, M. L., Glickman, S. E., & Place, N. J. (2022). Sex differences in spotted hyenas. *Cold Spring Harbor Perspectives in Biology, 14*(6), a039180. https://doi.org/10.1101/cshperspect.a039180

[23] Lacerda, G., Petrillo, F., Pimenta, M., & Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software, 167,* 110610. https://doi.org/10.1016/j.jss.2020.110610

[24] Krishna, M. M., Panda, N., & Majhi, S. K. (2021). Solving traveling salesman problem using hybridization of rider optimization and spotted hyena optimization algorithm. *Expert Systems with Applications, 183,* 115353. https://doi.org/10.1016/j.eswa.2021.115353

[25] Dhiman, G., Oliva, D., Kaur, A., Singh, K. K., Vimal, S., Sharma, A., & Cengiz, K. (2021). BEPO: A novel binary emperor penguin optimizer for automatic feature selection. *Knowledge-Based Systems, 211,* 106560. https://doi.org/10.1016/j.knosys.2020.106560

[26] Harifi, S., Mohammadzadeh, J., Khalilian, M., & Ebrahimnejad, S. (2021). Hybrid-EPC: An Emperor Penguins Colony algorithm with crossover and mutation operators and its application in community detection. *Progress in Artificial Intelligence, 10*(2), 181–193. https://doi.org/10.1007/s13748-021-00231-9

[27] Subha, R., & Haldorai, A. (2022). Improved EPOA clustering protocol for lifetime longevity in wireless sensor network. *Sensors International, 3,* 100199. https://doi.org/10.1016/j.sintl.2022.100199

[28] Dhiman, G. (2021). SSC: A hybrid nature-inspired metaheuristic optimization algorithm for engineering applications. *Knowledge-Based Systems, 222,* 106926. https://doi.org/10.1016/j.knosys.2021.106926

[29] Muhammad, A. H., Siddique, A., Youssef, A. E., Saleem, K., Shahzad, B., Akram, A., & Al-Thnian, A. B. S. (2020). A hierarchical model to evaluate the quality of web-based e-learning systems. *Sustainability, 12*(10), 4071. https://doi.org/10.3390/su12104071

[30] Tarwani, S., & Chug, A. (2020). Investigating optimum refactoring sequence using hill-climbing algorithm. *Journal of Information and Optimization Sciences, 41*(2), 499–508. https://doi.org/10.1080/02522667.2020.1724614

[31] AbuHassan, A., Alshayeb, M., & Ghouti, L. (2021). Software smell detection techniques: A systematic literature review. *Journal of Software: Evolution and Process, 33*(3), e2320. https://doi.org/10.1002/smr.2320

[32] AlOmar, E. A., Wang, T., Raut, V., Mkaouer, M. W., Newman, C., & Ouni, A. (2022). Refactoring for reuse: An empirical study. *Innovations in Systems and Software Engineering, 18*(1), 105–135. https://doi.org/10.1007/s11334-021-00422-6

[33] Alizadeh, V., Fehri, H., & Kessentini, M. (2019). Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction. In *19th International Working Conference on Source Code Analysis and Manipulation,* 181–192. https://doi.org/10.1109/SCAM.2019.00029

[34] Ghafori, S., & Gharehchopogh, F. S. (2022). Advances in spotted hyena optimizer: A comprehensive survey. *Archives of Computational Methods in Engineering, 29*(3), 1569–1590. https://doi.org/10.1007/s11831-021-09624-4

[35] Panda, N., Majhi, S. K., & Pradhan, R. (2022). A hybrid approach of spotted hyena optimization integrated with quadratic approximation for training wavelet neural network. *Arabian Journal for Science and Engineering, 47*(8), 10347–10363. https://doi.org/10.1007/s13369-022-06564-4

[36] Jia, H., Li, J., Song, W., Peng, X., Lang, C., & Li, Y. (2019). Spotted hyena optimization algorithm with simulated annealing for feature selection. *IEEE Access, 7,* 71943–71962. https://doi.org/10.1109/ACCESS.2019.2919991

[37] Khan, M. R., & Das, B. (2021). Multiuser detection for MIMO-OFDM system in underwater communication using a hybrid bionic binary spotted hyena optimizer. *Journal of Bionic Engineering, 18*(2), 462–472. https://doi.org/10.1007/s42235-021-0018-y

[38] Das, S. R., Sahoo, A. K., Dhiman, G., Singh, K. K., & Singh, A. (2021). Photo voltaic integrated multilevel inverter-based hybrid filter using spotted hyena optimizer. *Computers & Electrical Engineering, 96,* 107510. https://doi.org/10.1016/j.compeleceng.2021.107510

[39] Abid, C., Alizadeh, V., Kessentini, M., Dhaouadi, M., & Kazman, R. (2021). Prioritizing refactorings for security-critical code. *Automated Software Engineering, 28*(2), 4. https://doi.org/10.1007/s10515-021-00281-2

[40] AbuHassan, A., Alshayeb, M., & Ghouti, L. (2024). Software refactoring side effects. *Journal of Software: Evolution and Process, 36*(1), e2401. https://doi.org/10.1002/smr.2401

[41] Kataoka, Y., Imai, T., Andou, H., & Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance,* 576–585. https://doi.org/10.1109/ICSM.2002.1167822

[42] Alsarraj, R., & Altaie, A. (2021). Refactoring for software maintenance: A review of the literature. *Journal of Education and Science, 30*(1), 89–102. http://dx.doi.org/10.33899/edusj.2020.127426.1085

[43] Wholin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering: An introduction.* USA: Springer.

[44] Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., & Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology, 126,* 106347. https://doi.org/10.1016/j.infsof.2020.106347

[45] Moghadam, I. H., Ó Cinnéide, M., Sardarian, A., & Zarepour, F. (2024). Model-based source code refactoring with interaction and visual cues. *Journal of Software: Evolution and Process, 36*(5), e2596. https://doi.org/10.1002/smr.2596

[46] Kuo, J. Y., Hsieh, T. F., Lin, Y. D., & Lin, H. C. (2024). The study on software architecture smell refactoring. *International Journal of Software Innovation, 12*(1), 1–17. https://doi.org/10.4018/IJSI.339884

[47] Nandini, A., Singh, R., & Rathee, A. (2024). Code smells and refactoring: A tertiary systematic literature review. *International Journal of System of Systems Engineering, 14*(1), 83–143. https://doi.org/10.1504/IJSSE.2024.135914

[48] Aljohani, A., & Do, H. (2024). From fine-tuning to output: An empirical investigation of test smells in transformer-based test code generation. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing,* 1282–1291. https://doi.org/10.1145/3605098.3636058