Recommending Refactoring Solutions for Code Smells using Sandpiper Optimization Algorithm

D. Juliet Thessalonica
Ramanujan Computing Centre, Anna University Chennai 600025, Tamil Nadu, India
Email: juliet.jsamuel@gmail.com

H. Khanna Nehemiah
Ramanujan Computing Centre, Anna University Chennai 600025, Tamil Nadu, India
Email: nehemiah@annauniv.edu

ABSTRACT

Code smells are indicators of software defects that increase the cost of software maintenance and difficult to update the software. A framework for optimizing the refactoring sequence using Sandpiper Optimization Algorithm (SOA) is designed and implemented. The objective is to detect code smells in the software and fix them effectively. The goals are met by using metric-based detection rules from earlier research and refactoring the code with sequences chosen from Fowler's refactoring catalog using SOA. The fitness function, which is determined, based on priority, severity, risk, and importance of classes, is used to evaluate the optimized refactoring solutions. The priority score assigned to code smell is focused on the expert's preferences. The severity score is derived from the infusion tool that shows the degree of severity of code smells. A risk score is generated from the SonarQube tool based on continuous observation of code quality. An importance score gives the history of changes in the code extracted from the Github. Open-source software used for experimentation includes GanttProject and Xerces-J. The proposed work achieved refactoring precision values of 78 percent and 68 percent for GanttProject and Xerces-J respectively.

Keywords—Software maintenance, Code Smells, Refactoring, Sandpiper Optimization Algorithm.

I. INTRODUCTION

Software maintenance and evolution of systems is the vital activity that requires 90% of the total software production costs [1]. However, the fact remains true that a large percentage of software costs is spent on software maintenance. Software maintenance is more than simply fixing bugs; it also includes making code changes by adding functionalities and repairing the code to improve its performance. Constantly evolving systems increases the complexity by introducing design flaws and code smells [2]-[4].Code smells can negatively affect quality characteristics like flexibility and maintainability because they are signs of a more serious problem with the system [5]. Refactoring must be used to restructure the code and address these code smells. Fowler [4] presents 22 code smells and 72 refactoring that have been extensively outlined in the literature [6]-[8].

Refactoring software without altering its observable behaviour can make it simpler to understand and less expensive to modify [4]. The software continues to have the same functionality as before. It is a systematic approach of code cleaning that reduces the possibility of introducing bugs. In essence, refactoring enhances the structure of code after it has been created. Kent Beck [4] described two types of qualities in the software program, “the value of today and the value of tomorrow”. Programmers frequently concentrate on immediate goals. They increase the value of today’s software program by fixing a bug or adding a feature, but they are not quite sure about tomorrow. A choice based on yesterday's assumption can be

changed today. Similarly, the understanding that the programmers have today may seem foolish tomorrow that too may be modified. Refactoring can be used to address code smells caused by all these frequent changes.

Refactoring is accomplished in two primary steps namely, the detection of code smells and suggested refactoring operations to restructure the code. The first step is achieved through a variety of techniques and methods described in the literature [9]-[13]. Code smells differ in their effects and importance; they must be classified based on their severity and risk [14]-[15]. The second step includes manual and semi-automated refactoring techniques to fix specific code smells without considering their impact and risk [16].

This paper proposes a framework by suggesting refactoring solutions to eliminate code smells in which the code smells with the higher risk are given importance. The objective is to find the optimal sequence of refactoring solutions from the list of Fowler's refactoring catalog (http://www.refactoring.com/catalog/). The SOA [17] is a nature-inspired meta-heuristic algorithm that competes with GA [18], PSO [19], CRO [20] and SPEA [21] in suggesting refactoring solutions. Proposed by Kaur et al., SOA mimics the two natural behaviors of sandpipers namely migrating and attacking, and attempts to travel in the direction of the best-fittest sandpiper. The SOA is used to select the appropriate refactoring solutions by maximizing the quantity of corrected code smells based on risk, severity and importance of code fragments.

The remainder of the paper is organized as follows: Section 2 presents the related works in the literature. Section 3 presents an outline of the background materials and methods. Section 4 provides description of the proposed work. Section 5 presents the empirical design and definition. Section 6 presents the results and evaluation. The challenges to the work's validity are addressed in Section 7. Conclusion and scope for future work are provided in Section 8. Appendix 1 presents the abbreviations used in this work in alphabetical order.

II. RELATED WORKS

Ouni et al. [13] have presented an approach that employs two algorithms GP and NSGA-II to detect defects and correct them. The detection step generates detection rules from the defects examples and software metrics using GP. The correction step applies the detection rules and recommends a set of refactoring solutions using NSGA-II. The approach has been experimented on QuickUML, Xerces-J, GanttProject, ArgoUML, Log4J and Azureus. The effectiveness of the approach shows that 75% of the detected defects have been fixed using the suggested refactoring.

Saranya et al. [21] have presented an approach using the SPEA that prioritizes the list of refactoring solutions, while maintaining consistency with the previous refactoring in the processing of fixing the code smells. Six code smells namely Blob, Functional Decomposition, Data Class, Schizopheric Class, Shotgun Surgery and Swiss Army Knife have been detected from JHotdraw and Xerces-J software. Following the detection of code smells, the changes in each class are tracked using Git Hub repositories, and the change frequency has been calculated. The context similarity has been tracked using structural and semantic similarity. The SPEA generates an optimal set of refactoring suggestions using the aforementioned inputs. SPEA fixes 94% of code smells, compared to 90% for CRO and 75% for NSGA II, respectively.

Ouni et al. [22] have presented a multi-objective optimization approach for determining the best refactoring solutions by reducing the quantity of code smells while maintaining the construct semantics. The approach has considered the source code, a list of refactoring, code smell detection rules, semantic measures, a history of code changes as input and generated an optimal set of refactoring operations. The approach has been experimented on Xerces-J,

JFreechart, GanttProject, Artofillusion and JHotdraw. The effectiveness of the approach produces 86% of suggested refactoring which is consistent with the change history and fixes 85% of code smells are fixed.

Dea et al. [23] have presented D-EA that uses two GA with different fitness function, solution representation, and operators to fix code smells by producing best refactoring solutions. The first GA algorithm evaluates the refactoring solutions based on the amount of detected code smells and the second GA algorithm evaluates solutions by reducing the similarity between the reference code and code containing code smells. The similarity is based on the use of global and local alignment techniques. Local alignment identifies similarities between parts of the reference code and faulty code. Global alignment is the end-to-end matching between the reference code and the faulty code. The best solutions from both algorithms are then selected to fix the code smells. The work has been experimented on Xerces-J, GanttProject, ArgoUML, Ant-Apache and Azureus. The effectiveness of the approach shows that 86% of the code smells were corrected using suggested refactoring.

Malhotra et al. [24] have presented a framework to find the probable classes that need quick refactoring based on code smells and design attributes. The work computes QDIR for each class to identify classes that are severely affected and require immediate refactoring operation. The work has been experimented on Java Project ORDrumbox. Based on the QDIR value, priority has been assigned to 10% of the total classes, and the appropriate refactoring method has been suggested. The results show that by providing refactoring operations to the most critically affected classes which constitute 10% of the total classes, an overall 47% improvement in software quality has been achieved.

Kebir et al. [25] have presented detection rules and refactoring catalog for architectural bad smells. These bad smells are software designs that emerge from the reverse engineering and re-engineering process. Automated refactoring of design smells has been implemented using the genetic algorithm. Architectural smells namely, connector envy, ambiguous interface, component concern overload, scattered parasitic functionality and overused interface have been detected based on detection rules comprising software metrics. To correct the above-mentioned bad smells, refactoring solutions namely, pull interface, extract component, push component and extract interface have been used. Experiments have been conducted with open-source Eclipse MAT. The result shows that the design smells have been fixed with an acceptable efficiency of 53% (4.41/8.27).

Vidal et al. [26] have presented a tool to rank code smells based on changes in the component, modifications of software scenarios and importance of code smell. Moose framework for software analysis has been used to build the tool named spIRIT. The system has been loaded with MSE, a generic file format similar to XML to evaluate it in spIRIT. Ten code smells have been detected by spIRIT. Each smell has been represented as a rule comprising of metrics and predetermined thresholds. The smells have been ranked according to their importance. The ranking has been calculated by aggregating SRC, RCS and RMS values. Experiments have been carried out on beef-cattle farm simulator and subscriberDB application to rank code smells. The ranking by the spirit tool has been compared with an expert. The outcome reveals a strong correlation between the ranking suggested by the expert and the one suggested by spIRIT with a probability between 0 and 0.5.

Ghannem et al. [27] have presented a multi-objective approach for correcting defects within a model by producing a meaningful sequence of refactoring. The first objective is to increase structural and textual similarities between a particular model and a group of badly designed models and the second objective is to reduce the structural similarity between the given model and well-designed models. The candidate solution with the highest structural and textual similarities between a given model and well-designed models provides the best sequence of refactoring. The experimental results on eight Java projects namely Ant,

GanttProject, JabRef, JHotDraw, JRDF, JGraphx, Xerces and Xom yield an average correctness of 80% respectively.

The following conclusions are drawn from a review of numerous studies in the literature. To begin, semantic and structural similarity between original code and the infected code are used as the criteria for refactoring operations. Second code smells are ranked using SRC, RCS and RMS values. This work uses priority, severity, risk, and importance of classes as the fitness function for SOA to select the best refactoring sequences.

III. MATERIALS AND METHODS

This section outlines code smell detection rules, prioritization measures and the algorithm used for selecting refactoring operations.

A. Code Smell Detection Rules

Software metrics are measurable features used to assess software performance through the detection of code smells. Code smells exhibit few symptoms that can be identified using detection rules. These detection rules are extracted from previous research work [28]-[29]. Detection rules for detecting code smells considered in this work namely, blob, functional decomposition, feature envy, data class and spaghetti code are given in Table 1.

Table 1: Detection Rules for Code Smells

Code Smell	Detection Rule
Blob	(NLOCC>=1500 and NLOCM>=129) or NM>=100
Functional Decomposition	NOA > 2 and NIM ≤ 30 and CBO > 4
Feature Envy	ATFD(method)>4 and NOA ≤ 16
Data Class	NOAM > 2 and WMC ≤ 21 and NIM ≤ 30
Spaghetti Code	LOC(method)≥80 and CYCLO ≥ 8 and CBO > 8 and RFC>245

B. Refactoring Catalog

Refactoring is a method of improving the design of an existing code. It consists essentially of a series of minor behavioral changes. These changes must be carried out in small steps to reduce the risk of errors. Performing refactoring for a long period in small steps can help to maintain the system avoiding breakdown. The refactoring catalog at http://www.refactoring.com/catalog/ contains a large number of potential refactoring from which the proposed approach can select the best sequence.

A software having one blob, two DC, and two FD code smells are considered for experimentation. Assume that there are two solutions S1 and S2 to fix these code smells. The solution S1 fixes both DC and FD with the correction score CCR (S1) = 4/5 = 0.8, while the solution S2 fixes the blob, 1 DC, and 2 FD with the correction score CCR (S2) = 4/5 = 0.8. Although S1 and S2 both produce the same correction score, the blob class is more severe and it may have a significant impact on the overall system design. From this perspective, solution S2 is considered better than S1.

Consider a Student Registration System as shown in Figure 1 in which student gets enrolled in a semester. The semester class is made of one or more courses. Each course has a title and course Code. The student can register, drop, withdraw the course and can view the results in the semester. Also a semester may be freezed or attended. There are two kinds of

students namely Undergraduate Students and Postgraduate Students. Each type of students enrolls in different ways.

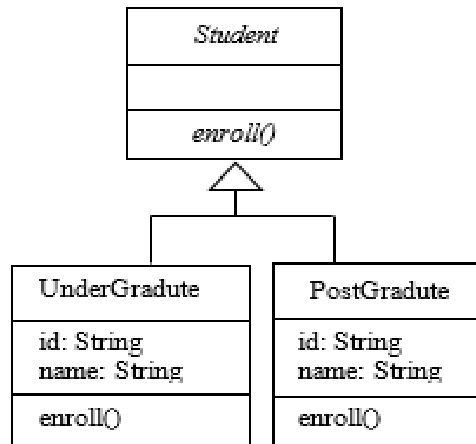

Figure 1: Student Registration System

When a defect occurs, classes must be restructured using refactoring operations. When two classes have association relationship, a unique field in one class should be referenced by another class. If the referencing of the class is missing, then there occurs a defect. As a result, classes need to be restructured. When restructuring classes, new classes may be added and functionality may be moved between classes. Move field refactoring allows moving a field from one class to another. The simplest way is to copy the field from the source class to the destination class. As presented in Figure 2, the Student class is associated to the Semester class, so the Student class should have a foreign key reference to the Semester Class. The MoveField(Semester, Student, id) operation is used to correct the defect by moving the id field from the source class Semester to the target class Student. Here semid is the foreign key in the Student class that refers to the primary key id in the Semester class.

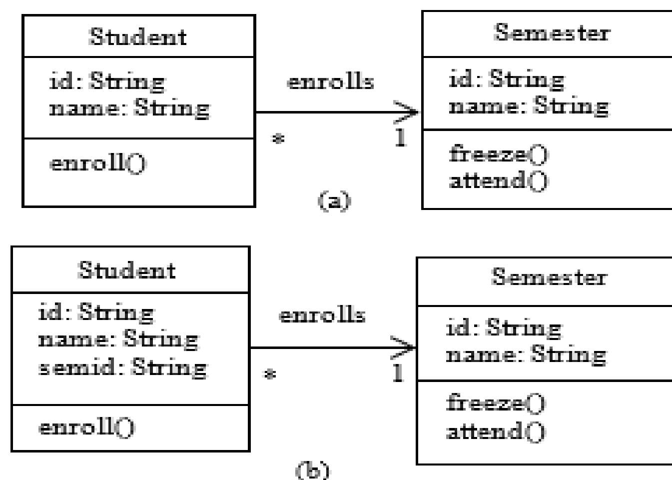

Figure 2: (a) Before refactoring (b) After move field refactoring operation

Class inheritance, hierarchy, and extraction are all aspects of abstraction. Abstraction is to reduce duplication in software programming. The pull-up/push-down refactoring is an illustration of abstraction. The pull up removes redundancy by pulling up code into a super

class. Methods and fields from a super class are pushed down into subclasses using push down. Pull up field refactoring allows moving the same field from subclasses to the superclass. The simplest way is to copy the field from two subclasses to the superclass.

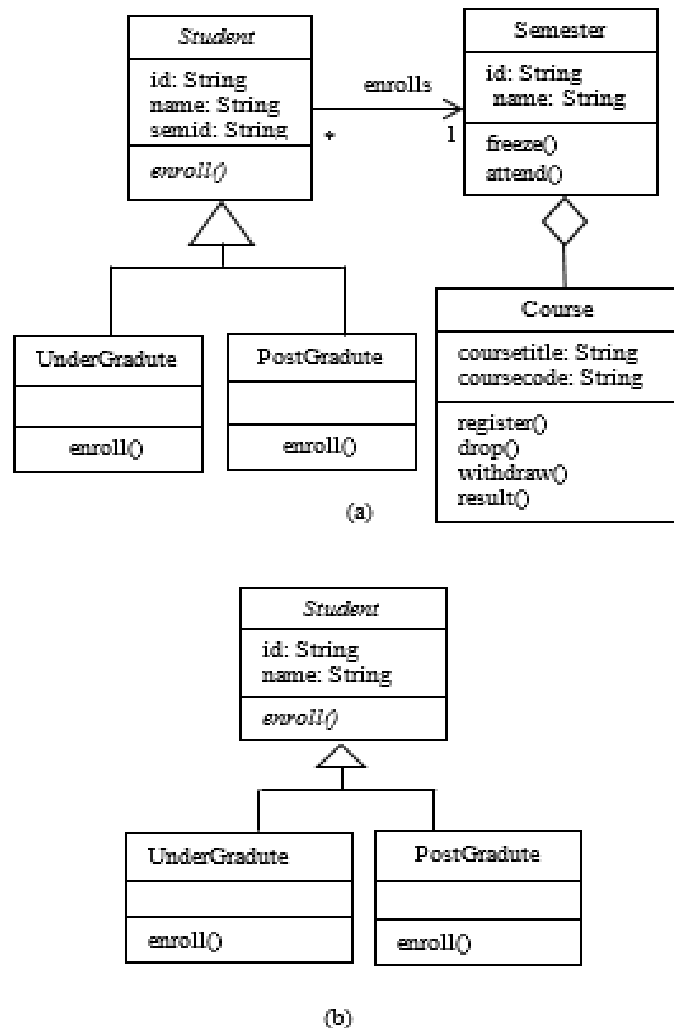

Figure 3: (a) Before refactoring (b) After pull up field refactoring operation

As presented in Figure 3, the class Under Graduate and Post Graduate are all subclasses of Student which are disjoint and complete. The source class Under Graduate and Post Graduate has a spaghetti code smell that must be corrected by using PullupField(). The PullupField(UnderGraduate, PostGraduate , Student, id, name) is used to correct the spaghetti code smell by moving the id and name fields from the source classes UnderGraduate and PostGraduate to the target class Student.

C. Prioritization Measures

This section provides an outline of four prioritization measures namely priority, severity, risk and importance score that are used as the fitness function to select the suggested refactoring sequences.

- Severity Score

Not all code smells have the same effect [16]. Each code smell has a severity score that allows developers to identify and fix the most critical instances. Concretely, multiple code fragments may contain the same type of code smell, but they may have varying impact scores. This score reflects the overall negative impact and associated code-smell severity. A blob code smell may be detected in two instances with 27 and 36 methods, but each has a different impact score on the system quality. The severity of code smells is measured using the infusion tool based on design characteristics, namely encapsulation, complexity, coupling, hierarchy and cohesion. Moreover, these design characteristics are effective in existing code smell detection methods [10]-[11],[13].

- Risk Score

The risk score is an important score that measures how risky the code smell is [12]. So, during the correction phase, it is important to give priority to the riskiest code smells. An open-source platform called SonarQube is used for continuous code quality checks. A risk score is given to each detected code smell that deviates from well-designed code.

- Priority Score

Some code smell types are usually prioritized by developers because they can have varying effects on the system's quality. Developers can prioritize different types of detected code smell on their preferences. Based on the literature on software refactoring and code smell correction [16], [30] , a priority score of 5 has been assigned to the blob code smell, 4 to functional decomposition, 3 to spaghetti code, 2 to feature envy, and 1 to the data class.

- Importance Score

Developers must understand which classes and packages in the code fragments are critical to the overall software system. Critical code fragments are those that change frequently during the maintenance process to add new functionalities and to accommodate new changes. Moreover, as reported in the literature [15], [31]-[32], classes having code smells are more likely to be changed. As a result, code smells associated with frequently changing classes should be given importance during the correction process.

D. Sandpiper Optimization Algorithm(SOA)

SOA is a bio-inspired meta-heuristic algorithm introduced by Kaur et al. that attempts to simulate the migration and attacking behavior of sandpipers. Sandpipers are sea birds that can be found all over the world. They eat insects, fish, reptiles, earthworms and are omnivorous. Sandpipers are intelligent birds that flock together. They can make rain-like sounds with their feet while searching for earthworms. Moreover, sandpipers have a unique pair of glands located just over their eyes that helps to remove excess salt from their body. The representation of a sandpiper depends on the problem and it can express a feasible solution. A sandpiper has two kinds of behavior namely, migrating and attacking. The migration behavior must meet three conditions namely, collision avoidance, convergence in the direction of best neighbor and updating position considering the best fittest. The attacking behavior generates spiral shape movement in the air which hits the prey. The exploration phase is a global search that investigates various promising solutions in the search space, whereas the exploitation phase is a local search that searches for optimal solutions among those promising solutions.

- Mathematical model for SOA

To apply SOA, the following elements have to be defined, how to encode solutions (sandpipers) in the search process, how to create a population of solutions (a group of sandpipers), how to measure the competency of candidate solutions using an evaluation

function, how to select solutions and how to create/modify new solutions using behavioral operations namely collision avoidance, convergence and updating position. The parameter setting of the sandpiper optimization algorithm was presented in Table 2. The steps of SOA are given in Algorithm 1.

Table 2: Parameter setting of SOA

Parameter	Value
Initial number of sandpiper	50
Parameter C_A	[2,0]
Parameter C_B	0.05
Max. no. of iterations	100

Algorithm 1: Sandpiper Optimization Algorithm

Input: Initial population P_{sp}

Process:

Step 1: Initialize the population of N sandpipers (solutions) at random. The length of each solution is n, where n is the number of refactoring operations. Figure 4 shows the initial population.

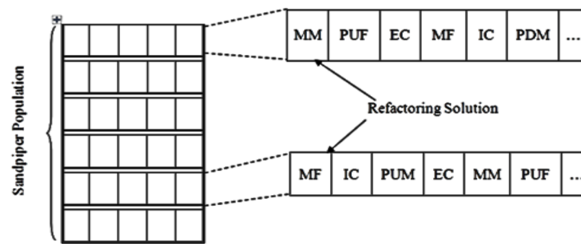

Figure 4: Population Generation and Solution Representation

Step 2: Calculate the fitness value of each sandpiper using fitness function for the solution 'w' as given in Eq.(1).

$$\text{Fitness } (w)=\sum_{i=0}^{n-1}(x_i*(\alpha*\text{severity}(c_i)+\beta*\text{priority}(c_i)+\gamma \quad (1)$$

where

$$x_i=\begin{cases}0 & \textit{if the actual class is detected as code smell by detection rules}\\ 1 & \textit{otherwise}\end{cases}$$

α+β+γ+δ = 1 ; Each having a weighted value = 0.25

Step 3: Sort the sandpipers in the descending order according to the fitness value and find the best fittest sandpiper P_{bst}

Step 4: Each sandpiper updates its position and generates a new set of sandpipers (solutions) computed using Eq. (2).

$$P'_{sp} = D_{sp} \times (x' + y' + z') \times P_{bst} \quad (2)$$

where P'_{sp} denotes updated position of sandpiper, D_{sp} denotes the distance between the sandpiper and the best fittest sandpiper, P_{bst} denotes the best fittest sandpiper.

$$x' = Radius \times Sin(i)$$
$$y' = Radius \times Cos(i)$$
$$z' = Radius \times i$$
$$Radius = u \times e^{kv}$$

where u and v are constants to define spiral shape and assigned a value of 1, e is the base of the natural logarithm, i lies in the range $0 \le k \le 2\pi$, $Radius$ is the radius of each spiral turn.

Step 5: Migration behavior directs the sandpiper to move from one position to another and can also change their speed and the angle of attack continuously. This movement triggers a discrete change in the position of sandpiper, which promotes higher exploration rather than the sandpiper becoming stuck around the same position. An additional variable C_A is employed for computing new position by avoiding collision between their neighbouring sandpipers computed using Eq.(3). After collision avoidance the sandpiper converge towards the direction of the best neighbor and is computed using Eq.(4).

$$C_{sp} = C_A \times P_{sp} \quad (3)$$

where C_{sp} denotes the new position which does not collide, P_{sp} denotes current position of the sandpiper, C_A is responsible for movement of sandpiper in a search space and assigned a value of 2.

$$M_{sp} = C_B \times (P_{bst} - P_{sp}) \quad (4)$$

where M_{sp} denotes new position after converge, $C_B = 0.5 \times Rand$,where $Rand = [0,1]$.

Step 6: Find the fitness value of newly generated sandpipers.

Step 7: Update P_{bst} , if there is a better solution than the previous optimal solution.

Step 8: Repeat steps 2 to 7 for a maximum of 100 iterations. The number of iterations is constrained to 100 because this will mean that the saturation will be reached. The solution with the maximum fitness value is considered the optimal refactoring subset, which is stored as the suggested refactoring.

Output: Suggested Refactoring Sequence

IV. PROPOSED WORK

The framework is composed of code smell detection, mapping code smells with refactoring and refactoring sequence selection subsystems as shown in Figure 5.

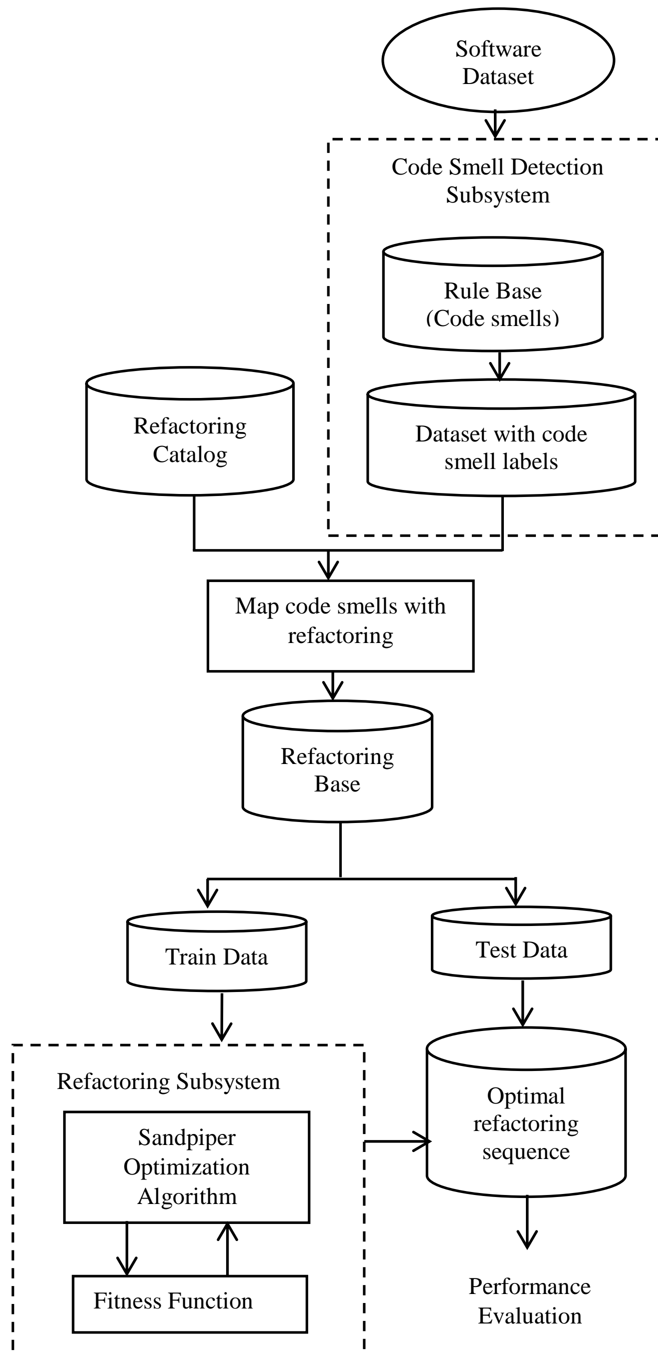

Figure 5: Proposed framework

A. Code Smell Detection Subsytem

Code smells identify the structural problems of a system. Each smell affects several components namely packages, classes and methods. This work considers code smells namely Blob, functional decomposition, feature envy, data class and spaghetti code. To help developers to identify code smells in the software, several detection rules are available in the literature. A

total of 65 and 161 instances of the five code smells have been found by examining Xerces-J and GanttProject, as shown in Table 3.

Table 3: Code smells in GanttProject and Xerces-J

Software	Code smells	Instances
GanttProject	Blob	7
	Functional Decomposition	18
	Feature Envy	11
	Data Class	16
	Spaghetti Code	13
	Total	65
Xerces-J	Blob	31
	Functional Decomposition	16
	Feature Envy	72
	Data Class	29
	Spaghetti Code	13
	Total	161

B. Mapping Code Smells with Refactoring

After detecting the five code smells they need to be mapped with the corresponding refactoring operation. Fowler proposed a catalog of 22 code smells and group of 70 refactoring for all the code smells [5]. The refactoring operations for the five code smells are extracted from the refactoring catalog and mapped with the corresponding code smells using the JDeodorant an eclipse plugin. Table 4 presents an outline of the five code smells and few of their associated refactoring operations.

Table 4: Code smells and their associated refactoring

Code Smell	Refactoring Operation
Blob	Extract class
	Extract subclass
	Extract interface
	Move method
	Move field
Functional Decomposition	Move method
	Move field
Feature Envy	Move method
	Extract method
	Inline class
	Push down method
	Push down field
Spaghetti Code	Extract subclass
	Extract super class
	Extract class

	Move method
	Pull up method
	Pull up field
Data Class	Move method
	Extract method

C. Refactoring Sequence Selection Subsystem

Refactoring would be time consuming by examining every code smells. It may be less important and given low priority to refactor code smells in a class that hasn't changed since its first implementation than a class that has seen more changes. By capturing priority, severity, risk and importance score the code smells can be prioritized. The rank of the code smells is calculated based on various factors namely priority, severity, risk and importance score. Priority scores are obtained from the developers who can prioritize different types of code smell based on their preferences. Severity scores are measured using the infusion tool based on design characteristics namely size, complexity, encapsulation, coupling, cohesion and hierarchy. Risk scores are measured using SonarQube tool based on continuous code quality inspection. Importance score are computed based on the code change history from the github repository. The fitness function is based on these four scores to evaluate the solution.

The refactoring subsystem employs the SOA algorithm to determine the optimal sequence of refactoring. It accepts inputs from a refactoring base containing code smells with a set of refactoring operations. Each solution is a set of refactoring operations along with their control parameters. Each refactoring operation is evaluated using the fitness function. The solutions are then refined using a predetermined number of iterations. The sandpipers are sorted based on their best fitness function. Each sandpiper updates its position and generates a new set of sandpipers (solutions). Migration behavior directs the sandpiper to move from one position to another and can also change their speed and the angle of attack continuously. The subsystem generates the optimal refactoring sequence, from a list of possible refactoring, with the goal of improving software quality by reducing the number of detected code smells.

- Solution representation

Vector-based solution coding has been implemented in SOA design. A refactoring operation is represented by vector dimension. These refactoring operations are applied in the order indicated by their vector positions. A set of control parameters is selected for each refactoring. Table 5 provides an illustration of a solution. Initially, an empty vector is created that represents the current refactoring solution. Then, a refactoring operation is selected from the list of possible refactoring operations along with its controlling parameters. The refactoring operation is applied to an intermediate model that represents the source code. After applying each refactoring operation, the model is updated and the process is repeated n times until the maximal solution length (n) is reached.

Table 5: Solution representation

Rop1	Move method(a1,a2,p)
Rop2	Pull-up field(a1,a2,q)
Rop3	Extract class(a1,a2)
Rop4	Move field(a1,a2,q)
Rop5	Inline class(a1,a2)
Rop6	Pushdown method(a1,a2,p)

- Refactoring feasibility

It is critical to ensure that the proposed framework is feasible and that the suggested refactoring can be implemented. Opdyke [33] is the first to introduce a method of formalizing the pre-conditions that must be imposed before refactoring to preserve the system's behavior. Opdyke [33] developed functions that could be used to formulate constraints in predicate expressions. These are similar to the analysis functions that Cinneide [34] and Roberts [35] used to reduce program analysis through automatic refactoring tool. The proposed work uses a system inspired by Cinneide to check a set of simple conditions that employs pre and post-conditions expressed in terms of conditions on the software code. The pre and post-conditions for the few refactoring operation is outlined in Table 6.

Table 6: Outline of pre and post-conditions for refactoring operation

Refactoring	Pre and post-conditions
Move method(a1,a2,p)	**Pre:** Classes a1, a2 must exist & p must be method of a1. **Post:** Classes a1, a2 must exist & p must be method of a2.
Move field(a1,a2,q)	**Pre:** Classes a1, a2 must exist & q must be field of a1. **Post:** Classes a1, a2 must exist & q must be field of a2.
Pull-up field(a1,a2,q)	**Pre:** Classes a1, a2 must exist and a2 is superclass of a1 & q must be field of a1. **Post:** Classes a1, a2 must exist and a2 is superclass of a1 & q must be field of a2.
Pull-up method(a1,a2,p)	**Pre:** Classes a1, a2 must exist and a2 is superclass of a1 & p must be method of a1. **Post:** Classes a1, a2 must exist and a2 is superclass of a1 & p must be method of a2.
Pushdown field(a1,a2,q)	**Pre:** Classes a1, a2 must exist and a2 is subclass of a1 & q must be field of a1. **Post:** Classes a1, a2 must exist and a2 is subclass of a1 & q must be field of a2.
Pushdown method(a1,a2,p)	**Pre:** Classes a1, a2 must exist and a2 is subclass of a1 & p must be method of a1. **Post:** Classes a1, a2 must exist and a2 is subclass of a1 & p must be method of a2.

V. EMPIRICAL STUDY DEFINITION AND DESIGN

Experiments are conducted on open-source systems, namely GanttProject and Xerces-J, to evaluate the framework's feasibility and efficiency in producing the best refactoring ideas. This section presents research questions, details of experiment design and discusses the results.

A. Research Questions and Objectives

The performance of the work is evaluated by determining whether it could lead to effective refactoring strategies that addresses code smells. The three research questions are outlined below, and the study explains how the experiments are set up to investigate them. The following are the three research questions:

RQ1: How many code smells can be fixed by the proposed approach?

RQ2: To what extent can the most severe, riskiest and important code smells be corrected by the proposed approach?
RQ3: How effective is the proposed SOA-based approach in comparison to other algorithms GA, PSO, CRO and SPEA?

B. Systems Studied

The work has been experimented on the well-known Java projects namely Xerces-J and GanttProject. Xerces-J is a software packages for parsing XML. GanttProject is a software tool for project management. These systems are selected for validation because they have been actively developed over the past ten years, and their design has not induced a slowdown in their development. An outline of descriptive statistics for the aforementioned software is shown in Table 7. Typically, a github repository is used to access open-source software and its change history. Code changes may be characterized in terms of recorded refactoring that are applied to earlier versions for software.

Table 7: Software statistics

Software	Release	Class	LOC	Complexity	Code smells	Code changes
GanttProject	V1.10.2	245	47k	1.697	83	91
Xerces-J	V2.7.0	991	238k	1.4	171	7493

C. Analysis Method

RQ1 is addressed using the CCR and RP metrics.

CCR measures the proportion of corrected code smells to the total number of detected code smells prior to applying the suggested refactoring sequence as shown in Eq.(5).

$$CCR = \frac{Numberof\ Corrected\ Code\ smells}{Total\ Number\ of\ Code\ smells\ before\ applying\ refactoring} \in [0,1] \qquad (5)$$

The feasibility of the proposed refactoring sequences for each system has been manually inspected for RP. The software ECLIPSE is used to implement the recommended refactoring, and the modified code fragments are examined for the semantic consistency. There have been some conceptual errors discovered through software behavior. When a conceptual error is discovered, the operations related to this change are considered as a bad recommendation. The correctness precision rate is expressed as the ratio of possible refactoring operations to the total number of proposed refactoring as shown in Eq.(6).

$$RP = \frac{Number\ of\ feasible\ refactorings}{Number\ of\ proposed\ refactorings} \in [0,1] \qquad (6)$$

RQ2 is addressed using the ICR, RCR and SCR metrics.

The ICR is expressed as the total sum of importance scores of detected code smells using a particular refactoring *w* to the one prior to refactoring as shown in Eq.(7). ICR measures the efficiency of a refactoring solution in fixing important code smells. The higher the ICR, better is the refactoring solution.

$$ICR = 1 - \frac{\sum_{i=0}^{r-1}(x_i \times importance(a_i))}{\sum_{j=0}^{s-1}(x_j \times importance(a_j))} \in [0,1] \qquad (7)$$

where s and r are the number of classes in the system before and after applying the refactoring solution *w*, the function $importance(a_i)$ gives the importance score of the class a_i, and x_i equals 0 if the actual class a_i is detected as a code smell using detection rules, and 1 otherwise.

The RCR is expressed as the sum of risk scores of detected code smells after applying a given refactoring solution w compared to the one prior to refactoring as shown in Eq.(8). RCR measures the efficiency of a refactoring solution in fixing the riskiest code smells. The higher the RCR, better is the refactoring solution.

$$RCR = 1 - \frac{\sum_{i=0}^{r-1}(x_i \times risk(a_i))}{\sum_{j=0}^{s-1}(x_j \times risk(a_j))} \in [0,1] \quad (8)$$

where s and r are the number of classes in the system before and after applying the refactoring solution w, the function $risk(a_i)$ gives the risk score of the class a_i, and x_i equals 0 if the actual class a_i is detected as a code smell using code smell detection rules, and 1 otherwise.
The SCR is expressed as the sum of severity scores of detected code smells after applying a given refactoring solution w compared to the one prior to refactoring as shown in Eq.(9). SCR reflects the efficiency of a refactoring solution in correcting severest code smells. The higher the SCR, better is the refactoring solution.

$$SCR = 1 - \frac{\sum_{i=0}^{r-1}(x_i \times severity(a_i))}{\sum_{j=0}^{s-1}(x_j \times severity(a_j))} \in [0,1] \quad (9)$$

where s and r are the number of classes in the system before and after applying the refactoring solution w, the function $severity(a_i)$ gives the severity score of the class a_i, and x_i equals 0 if the actual class a_i is detected as a code smell using code smell detection rules, and 1 otherwise.

RQ3 is addressed by evaluating the performance of the SOA algorithm compared to the three algorithms GA, PSO, CRO and SPEA. Additionally, recent surveys show that these three meta-heuristics are the most frequently used in solving various software engineering problems.

VI. RESULTS AND EVALUATION

The performance of SOA is compared with four popular meta-heuristics (GA, PSO, CRO and SPEA) using the same fitness function. Because the algorithms are stochastic, a slightly new sets of results are generated every time during execution. This difficulty is solved by providing experimental study based on 31 independent simulation runs for each algorithm. The average CCR, ICR, RCR, and SCR scores for identifying the refactoring solution with the suitable prioritization over 31 independent simulations upon GanttProject is 90, 80, 92, and 85%, as shown in Figure 6. The Wilcoxon rank sum test [36] is used to compare SOA-based approach and each of the other algorithms in terms of CCR, ICR, RCR, and SCR with a 99 % confidence level($\alpha = 1\%$). The tests show that the results are not merely coincidental but statistically significant ($p\ value < 0.01$).

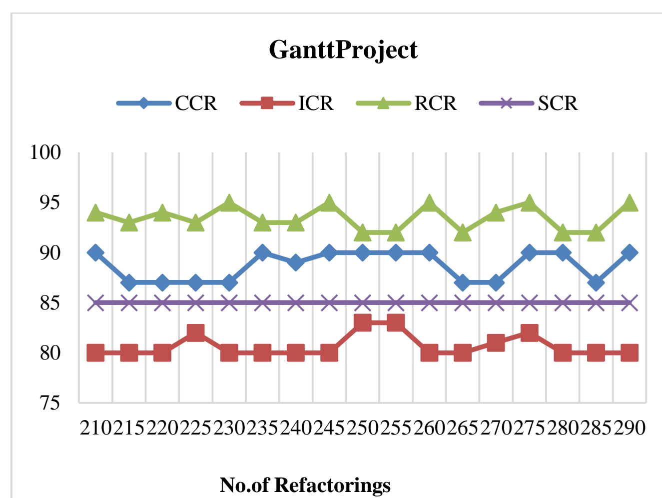

Figure 6: Multiple simulation runs on GanttProject

To better evaluate this work and answer RQ4, the results of SOA-based framework are compared with four different population-based algorithms (GA, PSO, CRO and SPEA), which have been shown to have competing performance in solving different software engineering problems. All algorithms employ the same formulation. According to the comparison results outlined in Table 8, SOA competes with the four algorithms in terms of CCR, ICR, and RCR with the refactoring precision of 70%. The average number of code smells fixed by SOA is 94%, compared to 84, 84, 90, and 93% for GA, PSO, CRO, and SPEA, respectively. In addition, SOA fixed 89% of significant code smells according to ICR, compared to less than 87% for other techniques. Based on these results, SOA competes with GA, PSO, CRO and SPEA.

Table 8: Comparison of Proposed work with GA, PSO,CRO and SPEA

Systems	Algorithm	CCR		ICR		RCR		SCR		RP	
		Score	*p* value	Score	*p* value	Score	*p* value	Score	*p* value	Score	*p* value
GanttProject	GA	87	<0.01	84	<0.01	93	<0.01	84	<0.01	67	0.387
	PSO	87	<0.01	84	<0.01	93	<0.01	86	<0.01	67	0.235
	CRO	91	<0.01	87	<0.01	91	<0.01	87	<0.01	67	0.586
	SPEA	92	<0.01	-	-	-	-	-	-	68	0.651
	Proposed	94	<0.01	88		92	<0.01	87	<0.01	68	0.235
Xerces-J	GA	84	<0.01	86	<0.01	87	<0.01	88	<0.01	76	0.689
	PSO	84	<0.01	85	<0.01	88	<0.01	87	<0.01	76	0.556
	CRO	91	<0.01	89	<0.01	90	<0.01	89	<0.01	76	0.695
	SPEA	94	<0.01	-	-	-	-	-	-	78	0.659
	Proposed	95	<0.01	90	<0.01	91	<0.01	90	<0.01	78	0.521

The suggested refactoring operations improve code quality significantly, with good code smell correction scores as outlined in Table 9. For Xerces-J, 100 % of blobs (31over 31), 94 % of Functional decomposition (15 over 16), 90 % of Feature envy (65 over 72), 90 % of Data Class (26 over 29) and 100 % (13 over 13) of spaghetti code are fixed. For GanttProject, 100 % of blobs (7 over 7), 100% of Functional decomposition (18 over 18), 91% of Feature envy (10 over 11), 88 % of Data Class (14 over 16) and 92% (12 over 13) of spaghetti code are fixed.

Table 9: Code smell Correction Ratio (CCR)

Software	Code Smells	CCR
GanttProject	Blob	100% (7/7)
	Functional Decomposition	100% (18/18)

	Feature Envy	91% (10/11)
	Data Class	88% (14/16)
	Spaghetti Code	92% (12/13)
Xerces-J	Blob	100% (31/31)
	Functional Decomposition	94% (15/16)
	Feature Envy	90% (65/72)
	Data Class	90% (26/29)
	Spaghetti Code	100% (13/13)

Another important factor is the distribution of refactoring operations. For both systems, the majority of the suggested refactoring is related to moving methods, moving fields, and adding parameters. Figure 7 illustrates how refactoring types are distributed differently in Xerces-J. Moving fields and moving methods are the two most frequently recommended refactorings. The majority of code smells are blob, functional decomposition, and spaghetti code that require refactoring.

To fix a blob code smell, attributes and methods of blob class are moved to other classes by reducing the functionalities of blob class and adding functionalities to other classes. As a result, refactoring operations namely the move field and move method are more likely useful to correct the blob code smell. Furthermore, code smells correction scores for data classes will be very low and most of them are corrected with a good score of 89 %. This is mainly because data classes are relatively simple to fix and do not require extensive refactoring. There is a structural relationship between data classes and blobs, so fixing blobs can implicitly fix data classes. Improving the correction of blobs can implicitly increase the correction of data classes.

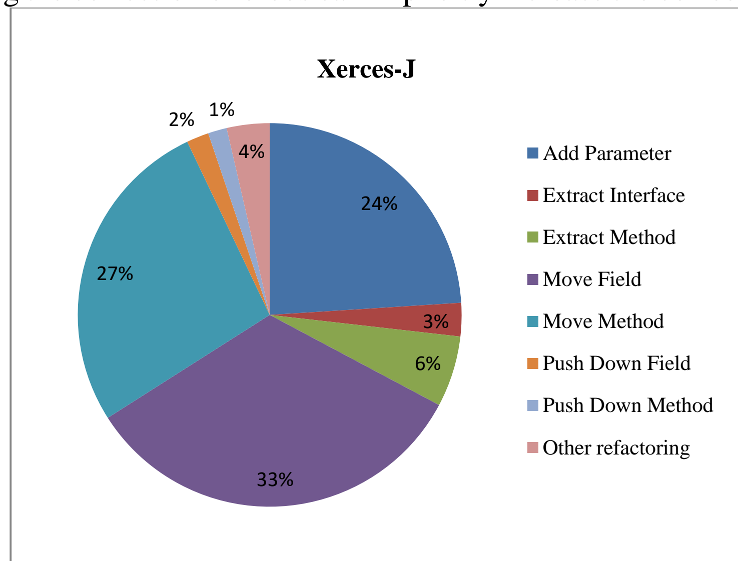

Figure 7: Suggested refactoring distribution for Xerces-J

The performance of the proposed method is also compared with two existing techniques; that is, SPEA, and the CRO. The improved performance of the proposed framework can be attributed to the ability of its efficiecy,effectiveness and the aplicability. Efficiency is determined by how well the suggested solution can correct code smells. The extent to which a framework is able to provide the necessary functionality serves as an index

of its effectiveness. The degree to which the framework can be adapted for various software is its applicability. The results of the comparison are shown in Figure 8 through Figure 10.

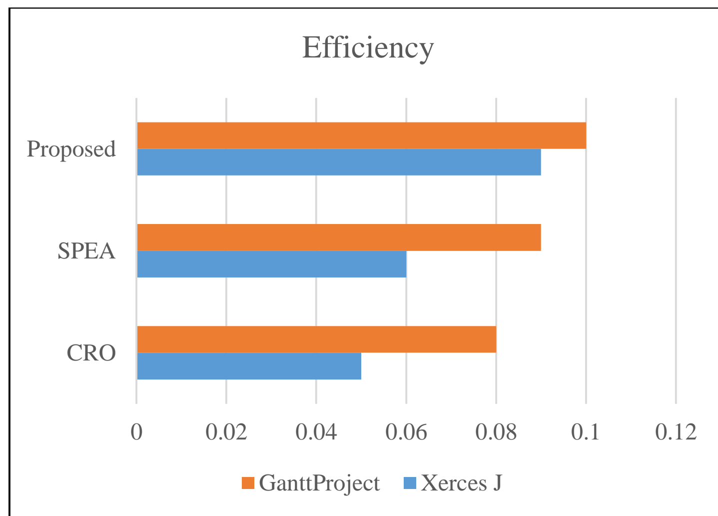

Figure 8: Efficiency – Proposed vs Exisiting work

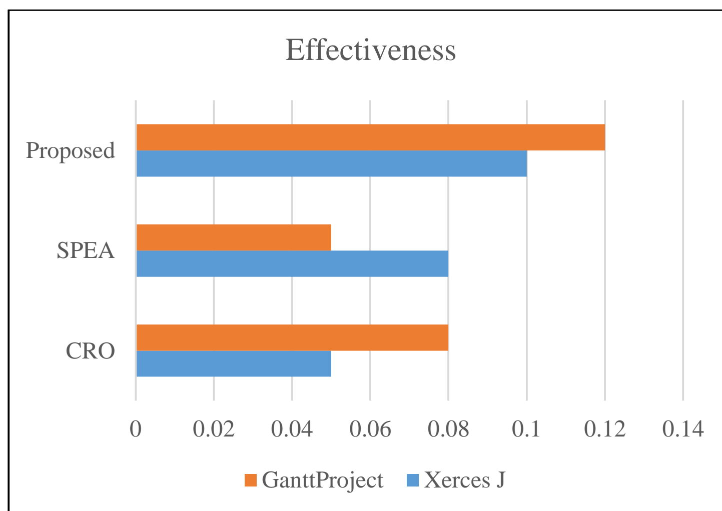

Figure 9: Effectiveness – Proposed vs Exisiting work

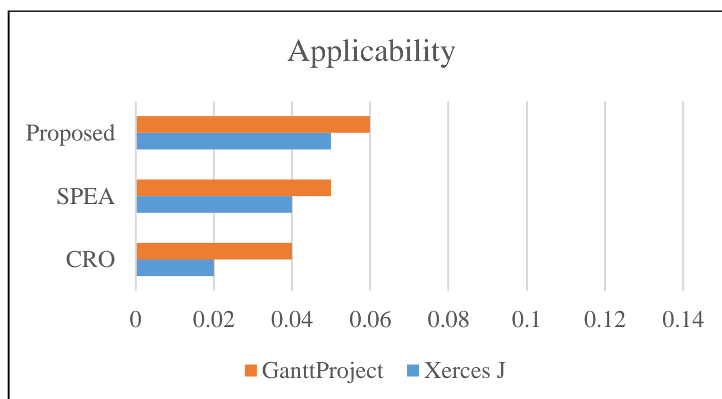

Figure 10: Applicability – Proposed vs Exisiting work

VII. THREATS TO VALIDITY

Construct validity refers to the relationship between the theory and the observation. The measurement of CCR that depends on the code smell detection rules is often questionable. This threat can be eliminated by inspecting and validating each code smell manually.

Internal validity addresses the bias in the results that were obtained. The results produced by the stochastic algorithms on 31 independent simulation runs for each instance are statistically analysed using the Wilcoxon rank sum test [36] with a 99 % confidence level($\alpha = 1\ \%$). The parameter tuning of the various optimization algorithms however creates another internal threat despite the same stopping criteria, which needs evaluation in further work.

External validity is the generalization of observed results outside the sample instances utilized in the experiment. In this work, experiments are performed on five different code smell types and two widely used open-source systems with different sizes, as outlined in Table 7. However, the findings are not applicable to industrial applications or to other programming languages. Further replications of this study are required to confirm the generalizability of our findings.

VIII. CONCLUSION AND SCOPEW FOR FUTURE

The Sandpiper optimization algorithm is used in this paper to recommend the best refactoring solutions for fixing code smells considering software maintainers' preferences. Initially metric-based detection rules are employed to detect code smells and the relevant refactoring operations are extracted from refactoring catalog. Refactoring operations are optimized using SOA to produce best refactoring sequences. The recommended refactoring sequence fixes the majority of critical, riskiest and severe code smells. Experiments on two software systems having five code smell types show that the proposed framework is competitive with four different popular meta-heuristic algorithms. The key benefit is to select the best refactoring sequence to fix the code smells. Future research would explore the connection between different code smells, as well as relationship between code smells and refactoring approaches. Injecting code smells into software and implicitly fixing them is considered as an additional objective.

ACKNOWLEDGEMENT

We thank Visvesvaraya, Ph.D, Scheme for Electronics and IT for the financial support of the research work.

APPENDIX

Appendix.1. Abbreviations Used

Abbreviation	Phrase
QDIR	Quality Depreciation Index Rule
SOA	Sandpiper Optimization Algorithm
GA	Genetic Algorithm

PSO	Particle Swarm Optimization
CRO	Chemical Reaction Optimization
SPEA	Strength Pareto Evolutionary Algorithm
GP	Genetic Programming
NSGA-II	Nondominated Sorting Genetic Algorithm
D-EA	Distributed evolutionary algorithm
MAT	Memory Analyzer Tool
spIRIT	Smart Identification of Refactoring Opportunities
MSE	Encrypted Maxscript
SRC	Stability of Related Component
RSS	Relevance of a Code Smell
RMS	Related Modifiability Scenarios
DC	Data class
FD	Functional Decompositions
CCR	Code Smells Correction Ratio
RP	Refactoring Precision
ICR	Importance Correction Ratio
RCR	Risk Correction Ratio
SCR	Severity Correction Ratio
XML	Extensible Markup Language

REFERENCES

[1] L. Erlikh, “Leveraging legacy system dollars for e-business”. IT professional, 2000, 2(3), pp.17-23.

[2] W.J Brown, R.C. Malveau, H.W McCormick III, and T.J Mowbray, “Refactoring software, architectures, and projects in crisis”. Google Scholar Google Scholar Digital Library Digital Library, 1998..

[3] N.Fenton, and J. Bieman, “Software metrics: a rigorous and practical approach”. CRC press, 2014.

[4] M. Fowler, “Refactoring”. Addison-Wesley Professional, 2018.

[5] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant’Anna, “On the evaluation of code smells and detection tools”. Journal of Software Engineering Research and Development, 5(1), pp.1-28, 2017.

[6] M. Misbhauddin, and M. Alshayeb, “UML model refactoring: a systematic literature review”. Empirical Software Engineering, 20, pp.206-251, 2015.

[7] E.V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, “A systematic literature review on bad smells–5 w's: which, when, what, who, where”. IEEE Transactions on Software Engineering, 47(1), pp.17-66, 2018.

[8] B.L Sousa, M.A Bigonha, and K.A Ferreira, “A systematic literature mapping on the relationship between design patterns and bad smells”. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (pp. 1528-1535), 2018.

[9] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws”. In 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. (pp. 350-359). IEEE, 2004.

[10] N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F. Le Meur "Decor: A method for the specification and detection of code and design smells". IEEE Transactions on Software Engineering, 36(1), pp.20-36, 2009.

[11] M. Kessentini, S. Vaucher and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code". In Proceedings of the IEEE/ACM international conference on Automated software engineering (pp. 113-122), 2010, September.

[12] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum and A. Ouni, "Design defects detection and correction by example". In 2011 IEEE 19th International Conference on Program Comprehension (pp. 81-90). IEEE, 2011, June.

[13] A. Ouni, M. Kessentini, H. Sahraoui and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach". Automated Software Engineering, 20, pp.47-79, 2013.

[14] S. Olbrich, D.S. Cruzes, V. Basili and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems". In 2009 3rd international symposium on empirical software engineering and measurement (pp. 390-400). IEEE, 2009, October.

[15] S.M. Olbrich, D.S Cruzes and D.I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems". In 2010 IEEE international conference on software maintenance (pp. 1-10). IEEE, 2010, September.

[16] M. Harman and L.Tratt, "Pareto optimal search based refactoring at the design level". In Proceedings of the 9th annual conference on Genetic and evolutionary computation (pp. 1106-1113), 2007, July.

[17] A. Kaur, S. Jain, and S. Goel, "Sandpiper optimization algorithm: a novel approach for solving real-life engineering problems". Applied Intelligence, 50, pp.582-619, 2020..

[18] D.E. Golberg, "Genetic algorithms in search, optimization, and machine learning". Addion wesley, 1989(102), p.36, 1989.

[19] J. Kennedy, and R. Eberhart, "November. Particle swarm optimization". In Proceedings of ICNN'95-international conference on neural networks (Vol. 4, pp. 1942-1948). IEEE, 1995.

[20] A. Ouni, M. Kessentini, S. Bechikh and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization". Software Quality Journal, 2015, 23, pp.323-361.

[21] G. Saranya, H.K. Nehemiah, A. Kannan, and V. Pavithra, "Prioritizing Code Smell Correction Task using Strength Pareto Evolutionary Algorithm". Indian Journal of Science and Technology, 11(20), pp.1-12, 2018..

[22] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue and M.S Hamdi, "Improving multi-objective code-smells correction using development history". Journal of Systems and Software, 105, pp.18-39, 2015..

[23] T.J. Dea, M. Kessentini, W.I Grosky, and K. Deb, "Software Refactoring Using Cooperative Parallel Evolutionary Algorithms".

[24] R. Malhotra, A. Chug, and P.Khosla, "Prioritization of classes for refactoring: A step towards improvement in software quality". In Proceedings of the Third International Symposium on Women in Computing and Informatics (pp. 228-234), 2015, August.

[25] S. Kebir, I. Borne, and D. Meslati, "A Genetic Algorithm for Automated Refactoring of Component-Based Software". EAI Endorsed Transactions on Creative Technologies, 3(9), pp. e2-e2, 2016.

[26] S.A Vidal, C. Marcos, and J.A Díaz-Pace, “An approach to prioritize code smells for refactoring. Automated Software Engineering, 23, pp.501-532, 2016.

[27] A. Ghannem, M. Kessentini, M.S. Hamdi, and G. El Boussaidi, “Model refactoring by example: A multi-objective search based software engineering approach”. Journal of Software: Evolution and Process, 30(4), p.e1916, 2018.

[28] D.J Thessalonica, H.K. Nehemiah, S. Sreejith and A. Kannan, “Metric-based rule optimizing system for code smell detection using Salp Swarm and Cockroach Swarm algorithm”. Journal of Intelligent & Fuzzy Systems, (Preprint), pp.1-18, 2022..

[29] D. Juliet Thessalonica, H. Khanna Nehemiah, S. Sreejith, and A. Kannan, “Intelligent Mining of Association Rules Based on Nanopatterns for Code Smells Detection”. Scientific Programming, 2023.

[30] A. Ouni, M. Kessentini, H. Sahraoui, and M.S Hamdi, “Search-based refactoring: Towards semantics preservation”. In 2012 28th IEEE International Conference on Software Maintenance (ICSM) (pp. 347-356). IEEE, 2012.

[31] A. Ouni, M. Kessentini, and H. Sahraoui, “Search-based refactoring using recorded code changes”. In 2013 17th European Conference on Software Maintenance and Reengineering (pp. 221-230). IEEE, 2013, March.

[32] F. Khomh, M. Di Penta, and Y.G. Gueheneuc, “An exploratory study of the impact of code smells on software change-proneness”. In 2009 16th Working Conference on Reverse Engineering (pp. 75-84). IEEE, 2009, October.

[33] W.F. Opdyke, “Refactoring object-oriented frameworks”. University of Illinois at Urbana-Champaign, 1992.

[34] M.O. Cinnéide, “Automated application of design patterns: a refactoring approach”. Dublin: Trinity College, 2001.

[35] D.B Roberts, “Practical analysis for refactoring”. University of Illinois at Urbana-Champaign, 1999.

[36] A. Arcuri, and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering”. In Proceedings of the 33rd international conference on software engineering (pp. 1-10), 2011, May.