Computers, Materials & Continua
DOI:10.32604/cmc.2021.015586
Article

Tech Science Press

Code Smell Detection Using Whale Optimization Algorithm

Moatasem M. Draz[1], Marwa S. Farhan[2,3,*], Sarah N. Abdulkader[4,5] and M. G. Gafar[6,7]

[1]Department of Software Engineering, Faculty of Computers and Information, Kafrelsheikh University, Kafr Elsheikh, Egypt
[2]Faculty of Informatics and Computer Science, British University in Egypt, Cairo, Egypt
[3]Department of Information Systems, Faculty of Computers and Artificial Intelligence, Helwan University, Cairo, Egypt
[4]Department of Computer Science, Faculty of Computers and Artificial Intelligence, Helwan University, Cairo, Egypt
[5]Faculty of Computer Studies, Arab Open University, Cairo, Egypt
[6]Department of Computer Science, College of Science and Humanities in Al-Sulail, Prince Sattam bin Abdulaziz University, Kharj, Saudi Arabia
[7]Department of Machine Learning and Information Retrieval, Faculty of Artificial Intelligence, Kafrelsheikh University, Kafr Elsheikh, Egypt
[*]Corresponding Author: Marwa S. Farhan. Email: Marwa.salah@bue.edu.eg
Received: 29 November 2020; Accepted: 14 February 2021

Abstract: Software systems have been employed in many fields as a means to reduce human efforts; consequently, stakeholders are interested in more updates of their capabilities. Code smells arise as one of the obstacles in the software industry. They are characteristics of software source code that indicate a deeper problem in design. These smells appear not only in the design but also in software implementation. Code smells introduce bugs, affect software maintainability, and lead to higher maintenance costs. Uncovering code smells can be formulated as an optimization problem of finding the best detection rules. Although researchers have recommended different techniques to improve the accuracy of code smell detection, these methods are still unstable and need to be improved. Previous research has sought only to discover a few at a time (three or five types) and did not set rules for detecting their types. Our research improves code smell detection by applying a search-based technique; we use the Whale Optimization Algorithm as a classifier to find ideal detection rules. Applying this algorithm, the Fisher criterion is utilized as a fitness function to maximize the between-class distance over the within-class variance. The proposed framework adopts if-then detection rules during the software development life cycle. Those rules identify the types for both medium and large projects. Experiments are conducted on five open-source software projects to discover nine smell types that mostly appear in codes. The proposed detection framework has an average of 94.24% precision and 93.4% recall. These accurate values are better than other search-based algorithms of the same field. The proposed framework improves code smell detection, which increases software quality while minimizing maintenance effort, time, and cost.

Additionally, the resulting classification rules are analyzed to find the software metrics that differentiate the nine code smells.

Keywords: Software engineering intelligence; search-based software engineering; code smell detection; software metrics; whale optimization algorithm; fisher criterion

1 Introduction

The complexity of software systems is rapidly increasing, which leads software houses to anticipate continuous change. Due to stakeholders' continuous demands for their reliance on these systems, software houses are under constant pressure to deliver the product on time [1]. Software systems contain high levels of complexity, and maintenance can prove to be difficult. Developers spend more than 60% of their time understanding the code before proceeding with its maintenance, leading to massive costs [2], which accounting for 50%–80% of software expenditure [3]. These expenses can be reduced by detecting and eliminating code smells in the early stages of development [4].

Fowler [5] defines code smells as signs in the program's source code indicating deeper issues that make it difficult to understand, modify, and maintain software. He [5] defines 22 types of code smells and refactoring opportunities. Our research focused on nine types: Large Class (Blob), Long Method (LM), Feature Envy (FE), Spaghetti Code (SC), Data Class (DC), Lazy Class (LC), Functional Decomposition (FD), Parallel Inheritance (PI), and Long Parameter List (LPL). Some of these smells appear in the design phase, whereas others appear in the implementation phase of the software development life cycle (SDLC) [6]. These negatively impact software maintainability and reliability, and damage software quality in the long term by incurring technical debt, introducing bugs, and increasing tension among team members [7].

We select these types because of their critical effect on software systems. Additionally, they continuously appear during the development process and have been used frequently in recent studies [8–13]. Our research will focus on optimizing detection rules for bad smells. We use a search-based technique that learns identification rules from software quality metrics to capture its structural architecture.

Search-based software engineering (SBSE) [14] defines detecting code smells as a search problem; an algorithm explores the search areas, guided by a fitness function that captures the properties of the desired solutions. Search space solutions represent the rules that identify code smells. To find the optimal detection rules, this research utilizes the Whale Optimization Algorithm (WOA) [15] as a classifier. WOA encircles the prey (optimal solution) and updates the search agents' positions to find the best solution using the Fisher criterion [16] as a fitness function. The Fisher criterion calculates the desirability of the rule by maximizing the between-class distance over the within-class variance. WOA improves the accuracy of code smell detection more than other genetic and evolutionary algorithms. It adopts if-then classification rules, which detect code smells during the life cycle of software development.

The rest of this paper is structured as follows: Section 2 provides an overview of code smells. Section 3 is a literature review. Section 4 presents and discusses the proposed SBSE solution for code smell detection. Section 5 presents and discusses experimental results. Section 6 concludes our findings and suggests future research directions.

2 Code Smell Overview

Code smells are symptoms of inadequate design or coding practices adopted by developers due to deadline pressure, lack of skills, or lack of experience [17]. These actions are also called bad-practices, anti-patterns, anomalies, or design defects. Fowler [5] defines 22 types of code smells, as well as refactoring opportunities. This research detects nine code smells types, explained as follows:

1. Large Class (Blob) is a class that monopolizes the behavior of the system. It can modify many fields and properties of the system without taking care of a principle of consistency. This smell type causes low cohesion and high coupling, and can need to difficulties in maintenance. It is also called Blob, Winnebago, and God class.
2. Long Method (LM (is a method that has many lines of code (LOC). If a method has ten lines of code or more, questions should be asked.
3. Feature Envy (FE) occurs when a method is more familiar with the properties of other classes than its own properties.
4. Spaghetti Code (SC) occurs when the necessary system structures are not included in the code. Object-oriented concepts are forbidden, such as polymorphism and inheritance.
5. Data Class (DC) is a class that has data fields but no methods calling the data. The only methods defined are the setters and the getters of these data.
6. Lazy Class (LC) is a useless class, which means that the class has low complexity and does not do much.
7. Functional Decomposition (FD) is found in code written by novice developers and refers to classes that are built to perform a single function.
8. Parallel Inheritance (PI) occurs when an inheritance tree is based on another inheritance tree after configuration. In other words, creating a sub-category for one category and then needing to create a sub-category for another category.
9. Long Parameter List (LPL) appears when a method contains many parameters in its signature; this code smell occurs when the method has more than four parameters.

The previous code smells are a direct result of bad practices from developers during SDLC. Testers found a list of software metrics related to them. This research is interested in studying 12 software metrics that constitute the features of the training and testing data detected by if-then rules. These metrics are as follows [7]:

1. Response for Class (RFC): The number of methods in an entity that can be executed when a message is received by an object belonging to the class.
2. Weighted Methods Per Class (WMC): The number of methodological complexities.
3. Lack of Cohesion between Methods (LCOM): the number of methods in a class that does not have at least one field in it.
4. Lines of Code (LOC): The number of lines in the code, except for abandoned lines and comments.
5. Coupling between Object Classes (CBO): The number of classes attached to a given class via field access, arguments, method calls, return type, and exceptions.
6. Number of Attributes Declared (NAD): The number of attributes declared.
7. Number of Parameters (NoParam): The number of parameters defined in a method.
8. Cyclomatic Complexity (CC): Indicates the complexity of a program. This measures the number of linearly independent paths within it. It is also called McCabe.
9. Number of Methods (NOM): The number of methods in a class specified.

10. Number of Overridden Methods (NMO): The number of methods overridden of an entity.
11. Depth of Inheritance Tree (DIT): The length of the hierarchy tree from the class to the parent class.
12. Number of Children (NOC): The number of the class's immediate descendants.

3 Literature Review

Over the last three decades, a variety of detection techniques has been used to detect code smell. Researchers have proposed manual technology [18,19], metric techniques [20–22], symptom-based techniques [23,24], probability-based techniques [25], visualization-based techniques [26,27], machine learning techniques [17,28–30], search-based software engineering [8,31–33], and other detection tools [34–37].

The code smell detection problem was formulated using search-based software engineering (SBSE) methods as an optimization issue. When a software engineering problem is formulated as a search issue, optimization algorithms can be used to solve it [14].

Kessentini et al. [31] employed the genetic programming (GP) to describe code smell detection rules. These rules are a blend of values and metrics that formulate the optimal identification of smells. The average accuracy of this approach is 70%.

Ouni et al. [32] developed a detection and correction technique. It has two targets: The first is the detection of code smell, and the second is the correction of it. They used GP for detection, a correction mission, and the non-dominated sorting genetic algorithm (NSGA-II). The average accuracy was 85% for precision and 90% for recall. This technique was applied to detect three types of code smells.

Boussaa et al. [33] implemented a detection technique that maximizes the coverage base of code smell examples and increases the number of "synthetic" software smells produced and not protected by the first population solutions. This technique used a competitive co-evolutionary algorithm (CCEA); it detected only three types with an average 80% for precision and recall. These approaches' accuracies were not promising as same as the technique presented by Ouni et al. [32], which also detected three types.

Usman et al. [8] introduced a multi-objective search-based technique to discover the most potent combination of metrics that not only maximizes the detection of code smell but also minimizes the detection of well-designed code. This research enhanced the accuracy of detection to 87% precision and 92% recall and increased it to five types.

Tab. 1 provides a comparison between the different approaches; to summarize

- The studies detected only a few code smells (between three and five) of the 22 types that Fowler [5] defined;
- As indicated, the detection accuracy was not high;
- Just one study defined rules for detection;
- They did not define the rules based on the size of software systems.

To address these shortcomings, this research constructs detection rules for both medium and large systems, and it enhances detection accuracy via a search-based technique. The proposed framework utilizes WOA [15] search capabilities to find the optimal metrics-based detection rules. They are guided by data given by five open-source systems on nine types, as defined in Section 2. This proposed framework uses the Fisher criterion, whose objective function maximizes the distance between different classes while minimizing the within-class distance. Hence, the detection

rules that best satisfy the objective will be chosen as the optimal solution. The WOA, Fisher criterion, and our proposed integration technique are introduced in Section 4.

Table 1: A comparison between the different search-based solutions

Comparison		Kessentini [31]	Ouni [32]	Boussaa [33]	Usman [8]
Algorithm		Genetic programming (GP)	Genetic programming and non-dominated sorting genetic algorithm (NSGA-II)	Competitive co-evolutionary algorithm (CCEA)	Multi-objective genetic programming (MOGP)
Goal		Defining rules for code smell detection	Detecting and correcting of code smells	Maximizing the coverage base of code smell	Maximizing the detection of code smell and minimizing the detection of well-designed code
No. of detected code smells		3	3	3	5
Accuracy (%)	Precision	70	85	80	87
	Recall		90		92

4 The Proposed SBSE Framework for Code Smell Detection

Harman describes metaheuristic search in SBSE as shifting from human-based to machine-based solutions to software engineering issues [14]. SBSE aims at reformulating software engineering issues as optimization topics [38,39] where, optimal or near-optimal solutions exist in the candidate solution search space. Driven by a fitness function, algorithms may differentiate between good and bad solutions to find the optimal one for the problem.

To solve the optimization problem of detection, we perform optimization with WOA [15], using the Fisher criterion [16] with software metrics as a fitness function. Fig. 1 shows the Business Process Model and Notation for the proposed SBSE framework for code smell detection. The process within the framework is discussed in the following subsections.

4.1 Prepare the Dataset Metrics in Columns and Define Code Smells Based on Rules of Detection

A metric or set of metrics is used to determine whether the code has bad smells or not. We identify code smells for each application based on rules that were used in earlier studies. Tab. 2 shows metrics used to define each type.

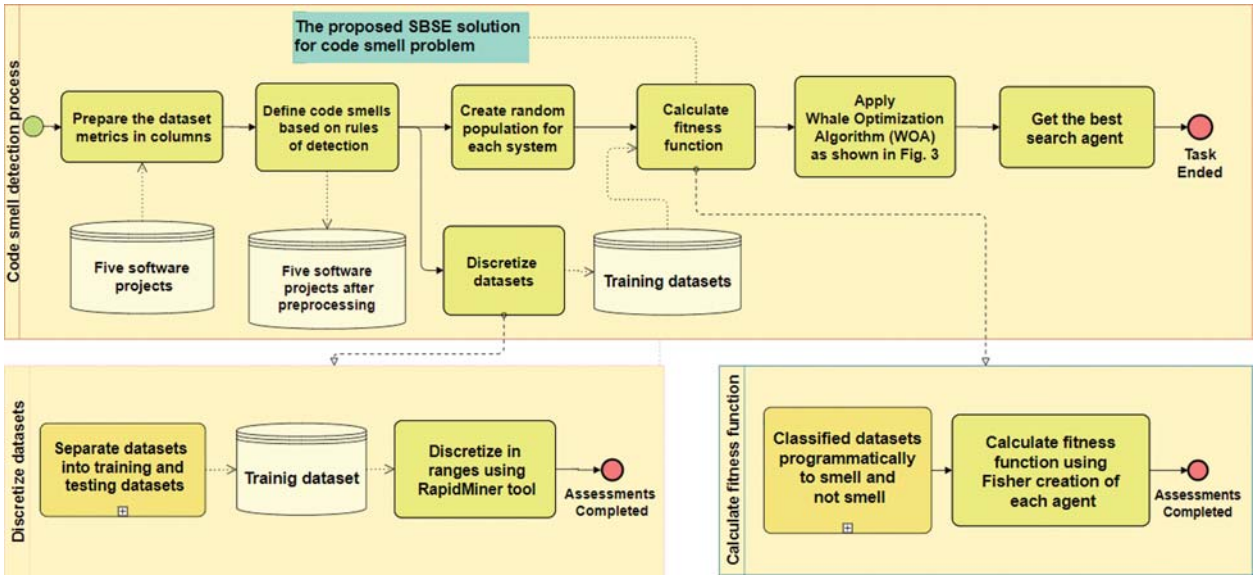

Figure 1: An overview of the proposed SBSE framework

Table 2: The sofware quality metrics used to define each type

Code smell type	Software quality metrics
Large Class (Blob)	LOC, NAD [8,40]
Spaghetti Code (SC)	McCabe, NOC [8,9]
Feature Envy (FE)	CBO, LCOM [40]
Long Method (LM)	CC [41]
Data Class (DC)	LCOM, WMC [40]
Long Parameter List (LPL)	NOPARAM [9]
Lazy Class (LC)	NOM, RFC [40]
Functional Decomposition (FD)	DIT, NMO [9]
Parallel Inheritance (PI)	DIT, NOC [40]

This research is conducted on five Java software projects. ArgoUML [42] is an application that helps stakeholders create complex and professional diagrams. Azure [43] is a file-sharing tool. Gantt Project [44] is a project or application-scheduling tool. Log4j [45] is a Java jar that helps in Java login. Xerces-J [46] is a tool used to parse XML. These projects are checked for the code smells based on the 12-software metrics using traditional software testing techniques. The metrics data and the code smells are used to prepare training and testing datasets, which we split using an 80:20 ratio. The training dataset is used to learn and train the algorithm to build the detection rules. The test data is used to evaluate our algorithm based on accuracy measures such as precision and recall. The number of modules and the detected code smells in each system are reported in Tab. 3.

4.2 Discretize Datasets

The probability of an exact match in metrics during the search is very low. To make the matching process possible, we discretize the search space using a binning technique to divide the

selected attributes into a user-specified number of ranges (bins). The range of numerical values for each metric (feature) is divided into equal-size segments (bins) by the Rapidminer tool [47].

Table 3: The number of modules and the detected code smells in each system

Project name	Release	Number of modules	Detected code smells								
			Blob	SC	FE	LM	DC	LPL	LC	FD	PI
ArgoUML [42]	V0.3	1470	16	47	38	79	24	17	22	51	14
Azure [43]	V2.3.0.6	1995	41	54	29	93	42	67	18	46	33
Gantt [44]	V1.10.2	229	5	14	8	12	11	10	0	9	3
Log4j [45]	V1.2.1	207	4	10	7	9	6	8	2	6	2
Xerces-J [46]	V2.7.0	911	13	29	25	43	21	19	20	29	12

4.3 Create Random Population for Each System (Probable Rules)

Optimization algorithms use a random initial population of probable solutions. The solution for the bad smell detection problem is a set of if-then rules defining the code smell types, and the initial population is a random sample of these if-then rules. The software metrics form the condition of each rule, and the code smell type forms the result. For example, "if ($f1 \in r1 \& f2 \in r2 \& \ldots \& f12 \in r1$), then CodeSmell = true" is a detection rule where, f is the software metric and r is the range. For each project, the population contains 100 individuals. The best if-then rule is the search agent that maximizes the calculated fitness function. A tolerance threshold is used to match search agents (rules) with the training data. The search agents with their corresponding fitness values become the input to the learning algorithm.

4.4 Calculate Fitness Function

Detection is solved using an optimization algorithm guided by a fitness function [38]. This evaluation function measures how a given solution is close to the optimum one for the problem (how fit a solution is). The fitness function evaluates the performance of each agent and gives agents with the most improvement for the highest probability of survival [48,49]. We use the Fisher criterion as a fitness function, which tests the distribution of inter-class scatter over in-class scatter [16].

A high fitness value means that the gap between any two classes is significant. Hence, the rules that maximize the distance between different classes are the fittest ones. The Fisher criterion between two groups, i and j, is defined as

$$f_{i,j} = \frac{|i - j|}{V_i + V_j} \tag{1}$$

where V_j is the variance of the jth class, defined as $V_j = \frac{1}{N-1}\sum_{k=1}^{N}(X_k - \mu_i)^2$, and μ_i represents the mean of the ith class, defined as $\mu_i = \frac{1}{N}\sum_{k=1}^{N} X_k$. X_k represents the k_{th} observation in class i, where, $1 \le k \le N$, and N is the number of observations. Each search agent is matched against the dataset instances. The one that maximizes the Fisher criterion (or maximizes the difference between the center of different classes while minimizing the differences between instances of the same class) is selected as the best search agent. The agents with the highest fitness function values represent the closest optimum solutions and are chosen as inputs for the WOA.

4.5 Apply Whale Optimization Algorithm (WOA) as a Classifier (Classification Rules)

We choose WOA as our classifier because several studies [50–52] showed that WOA [53] had the highest accuracy compared to the other state-of-the-art evolutionary algorithms, such as PSO [54] and GA [55]. WOA is a swarm-based metaheuristic algorithm. Inspired by the hunting behavior of humpback whales that prefer catching frogs or small fish near the surface of the water using traps, WOA mimics humpback whales in two phases [15]. The first is the exploitation phase, where whales encircle the prey and use the bubble net attacking method, and the second is the exploration phase where they randomly search for prey (Fig. 2).

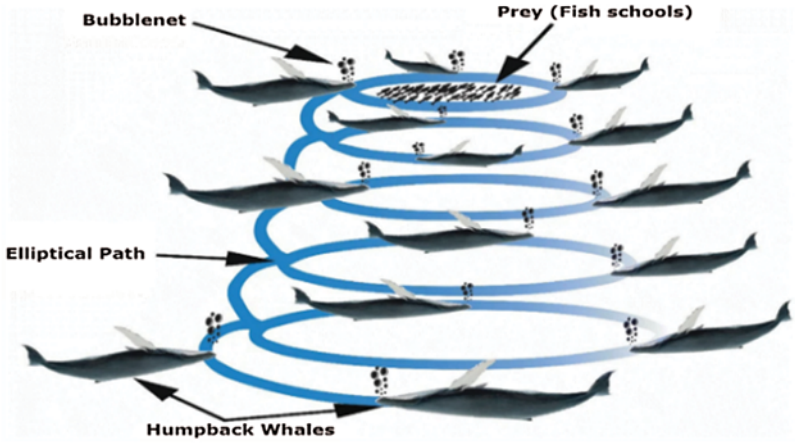

Figure 2: Whales encircle the prey [15]

4.5.1 Exploitation Phase (Encircling the Prey and Bubble-Net Attacking Method)

Since the optimal agent in the search space is not known, humpback whales decide on a location of the prey and encircle it. WOA assumes the target is near the best existing solution. Other search agents can mathematically move their positions closer to the best search agent after selecting it with these equations

$$\boldsymbol{D} = \boldsymbol{C} \cdot \boldsymbol{X}^{*} - \boldsymbol{X}(\boldsymbol{t}) \tag{2}$$

$$\boldsymbol{X}(\boldsymbol{t}+\mathbf{1}) = \boldsymbol{X}^{*}(\boldsymbol{t}) - \boldsymbol{A} \cdot \boldsymbol{D} \tag{3}$$

where C and A are coefficient vectors, X^* is the best-solution obtained, t is the real iteration, and X is the absolute value of the current solution. Values A and C are determined by

$$\boldsymbol{A} = \mathbf{2}\boldsymbol{a} \cdot \boldsymbol{r} - \boldsymbol{a} \tag{4}$$

$$\boldsymbol{C} = \mathbf{2} \cdot \boldsymbol{r} \tag{5}$$

where a is linearly decreasing vector from 2 to 0, and r is a random vector between 0 and 1. The agents update their positions according to Eq. (3) based on the best-known solution. Eqs. (4) and (5) aid by monitoring the areas where new solutions can be found in the best solution's neighborhood. The whales shrink around new solutions by reducing the value of a by

$$\boldsymbol{a} = \mathbf{2} - \boldsymbol{t} \cdot (\mathbf{2} \cdot \boldsymbol{MaxIteration}) \tag{6}$$

where t is the current iteration, and *MaxIteration* is the maximum number of allowed iterations.

After encircling shrinks, WOA calculates the distance between the current solution (X) and the best solution (X^*). The path between the humpback whale and the prey is represented as

$$\boldsymbol{X}(\boldsymbol{t}+\mathbf{1}) = \boldsymbol{D'} \cdot \boldsymbol{e}^{\boldsymbol{bl}} \cdot \boldsymbol{cos}(\mathbf{2}\boldsymbol{\pi}\boldsymbol{l}) + \boldsymbol{X^*}(\boldsymbol{t}) \tag{7}$$

where $D' = X^*(t) - X(t)$ is the distance between the whale X and the prey, b is the logarithmic spiral form, and l is a random value between -1 and 1. From previous equations, shrinking and moving in a spiral-shaped direction occur with a 50 percent probability (8)

$$\boldsymbol{X}(\boldsymbol{t}+\mathbf{1}) = \begin{cases} \boldsymbol{shrinking}\ (\text{Eq. 2}) & \boldsymbol{if}\ \ (\boldsymbol{p} < \mathbf{0.5}) \\ \boldsymbol{spiraling}\ (\text{Eq. 6}) & \boldsymbol{if}\ \ (\boldsymbol{p} > \mathbf{0.5}) \end{cases} \tag{8}$$

where p is a random number between 0 and 1.

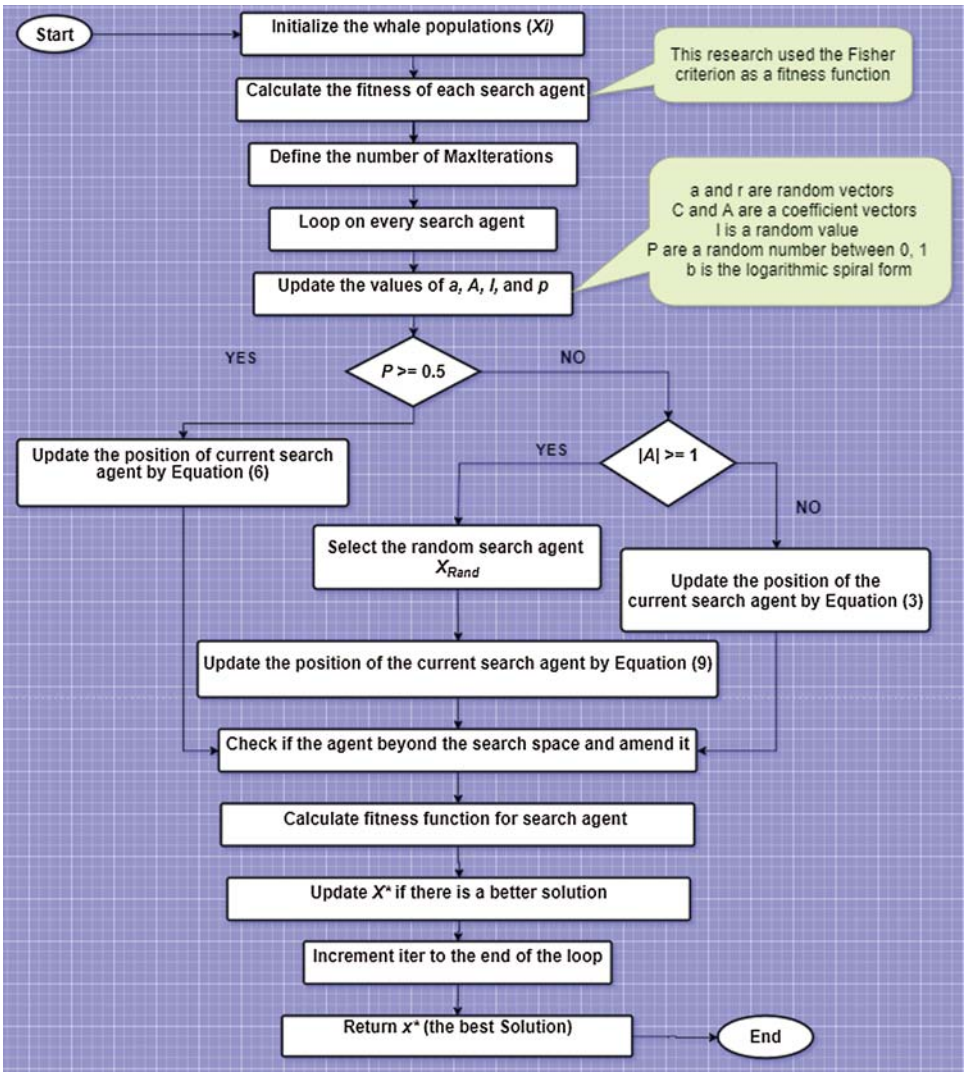

Figure 3: WOA flowchart

4.5.2 *Exploration Phase (Search for a Prey)*

Through updating the random solution chosen in this phase, WOA explores the optimal solution. To move suboptimal solutions away from the more popular search sites, the vector A is used to introduce random values less than -1 or greater than 1. As in the equation, it can be modeled mathematically in (9) and (10), respectively:

$$\boldsymbol{D} = \boldsymbol{C} \cdot \boldsymbol{X}_{\boldsymbol{Rand}} - \boldsymbol{A} \cdot \boldsymbol{D} \tag{9}$$

$$\boldsymbol{X}(\boldsymbol{T}+\boldsymbol{1}) = \boldsymbol{C} \cdot \boldsymbol{X}_{\boldsymbol{Rand}} - \boldsymbol{X} \tag{10}$$

where Xrand stands for a random whale picked from the current population. The flowchart for the previously outlined WOA phases is shown in Fig. 3.

5 Experimental Study and Evaluation

After completing the search, the resulting classification rules with the same software metrics are applied on a separate testing dataset to verify the efficiency of the proposed framework. In this section, we present the experimental setup and compare performance with previous approaches.

5.1 Experiment Setup

These experiments are simulated on a PC using an Intel(R) Core(TM) i5 CPU, 8 GB RAM and 1 TB hard disk using the following tools and jars:

- NetBeans IDE V. 8.2 [56] for data preprocessing and defining code smell types in training data;
- JDK 1.8 [57] for installing Java and coding;
- RapidMiner V. 5.3 [47] for the discretization process;
- JXL (Java Excel API) [58] and Apache POI (HSSf and SS) [59] for reading and writing in datasets Excel sheets;
- NetBeans IDE V. 8.2 [56] for the creation of random populations, calculating the fitness method, training the WOA, testing the WOA, and calculating accuracy.

The five software projects are traced for code smells using the corresponding software quality metrics and the rules explained in Section 4.1 (Tab. 2). The tracing process produces an initial version of the software metrics dataset that is further discretized to facilitate the incoming matching process. Afterward, the preprocessed dataset is fed into the integrated WOA framework, which initializes a population of probable classification rules and performs exploitation and exploration process of the search space. The resulting classification rules are determined by the Fisher criterion and the performance on testing data is measured.

5.2 Performance Measure

We use a confusion matrix model to measure the performance of our algorithm on test data. The performance parameters of the detection framework are shown in Tab. 4.

Table 4: A confusion matrix model

Actual class	Detected class	
	Non-smell	Smell
Non-smell	TN	FP
Smell	FN	TP

True positive (TP) indicates smell types detected correctly as a smell, false positive (FP) indicates non-smells detected incorrectly as a smell, True negative (TN) indicates non-smells detected correctly as it is not a smell, and false negative (FN) indicates smell types detected incorrectly as it is a non-smell.

We use precision and recall (Eqs. (11) and (12)) to evaluate the performance of our proposed smell detection framework. Precision is the number of relevant code smells among the found ones, while recall is the number of the total amount of relevant code smells that were retrieved.

$$\textbf{\textit{Precision}}\ (\textbf{\textit{PR}}) = \frac{\textbf{\textit{TP}}}{\textbf{\textit{TP}} + \textbf{\textit{FP}}} \tag{11}$$

$$\textbf{\textit{Recall}}\ (\textbf{\textit{RE}}) = \frac{\textbf{\textit{TP}}}{\textbf{\textit{TP}} + \textbf{\textit{FN}}} \tag{12}$$

Suppose that the proposed detection framework identifies 10 code smells in a system containing 14 modules (including both actual code smells and non-code smells). Of the 10 identified as code smells, 7 are code smells (TP), whereas the rest are not (FP). The model precision is 7/10, while its recall is 7/14. In this case, precision is "how beneficial the results of the detection model are," and recall is "how full the results are."

5.3 Experimental Results

Tab. 5 reports the performance evaluations (confusion matrix) of each software system with 80% training and 20% testing of data. A confusion matrix was used to calculate precision and recall on the five software systems and showed promising results. Our proposed framework achieves both high precision and recall.

Table 5: A confusion matrix of each software system

Project name	Release	Actual class	Non-smell	Smell	Precision	Recall
ArgoUML [42]	V0.3	Non-smell	260	1	97.2	97.2
		Smell	1	35		
Azure [43]	V2.3.0.6	Non-smell	336	2	96.5	94.9
		Smell	3	56		
Gantt [44]	V1.10.2	Non-smell	27	1	91.6	84.6
		Smell	2	11		
Log4J [45]	V1.2.1	Non-smell	21	1	92.8	92.8
		Smell	1	13		
Xerces-J [46]	V 2.7.0	Non-smell	181	3	92.8	97.5
		Smell	1	39		

5.4 Evaluation and Discussions

Tab. 6 compares the precision and recall between our proposed solution and different search-based algorithms such as MOGP [8], GP [32], CCEA [33], Multiobjective Immune Algorithm (MOIAS) [60], and GA [61]. The comparisons demonstrate the efficiency of the proposed solution in detecting code smells on large systems such as ArgoUML [42], Azure [43], and Xerces-J [46], and on medium systems such as Gantt [44] and Log4J [45] (Figs. 4 and 5). The WOA classifier

on the five software systems is better than the other optimization algorithms in terms of precision and recall, with averages of 94.42% and 93.4%, respectively (Tab. 6).

Table 6: The percentage of precision and recall of WOA and different search-based algorithms

Algorithm	WOA		MOGP [8]		MOAIS [60]		GP [32]		GA [61]		CCEA [33]	
	PR.	RE.	PR.	RE.	PR.	RE.	PR.	RE.	PR.	RE.	PR.	RE.
ArgoUML [42]	97.2	97.2	86	90	86	88	73	81	–	–	–	–
Azure [43]	96.5	94.9	86	95	86	92	76	97	–	–	71	74
Gantt [44]	91.6	84.6	79	90	76	86	63	83	84	87	–	–
Log4J [45]	92.8	92.8	–	–	–	–	86	83	83	82	–	–
Xerces-J [46]	92.8	97.5	85	90	85	90	71	83	87	82	93	88

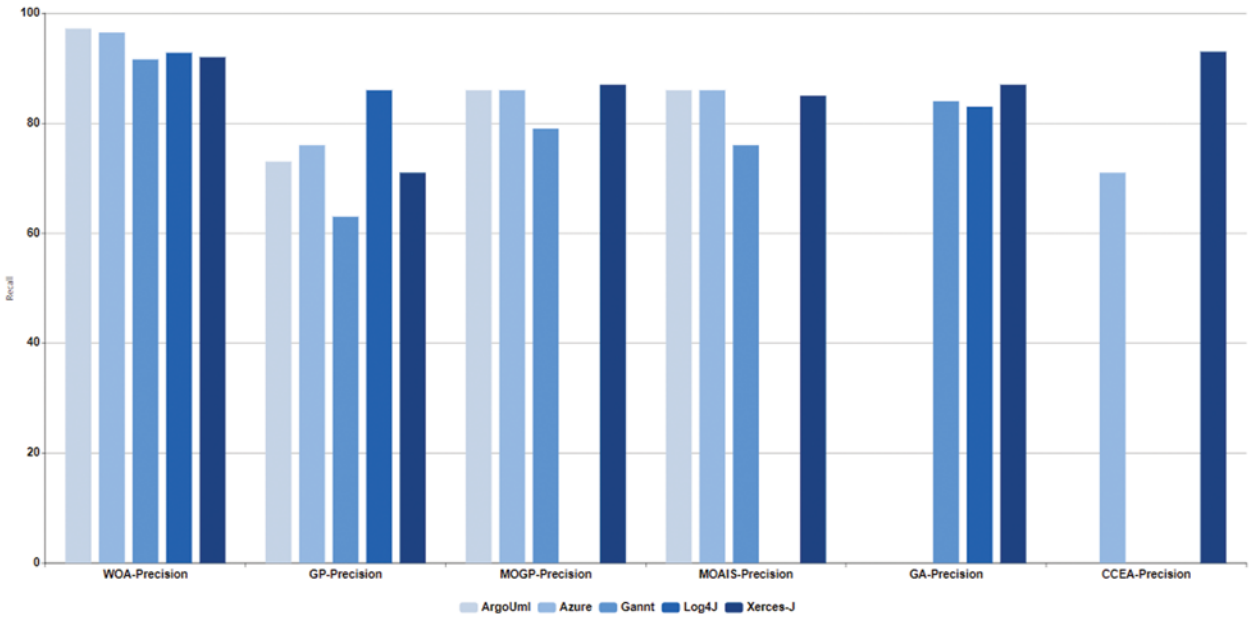

Figure 4: A comparison between WOA and other search-based algorithms using precision measure

Next, the most optimal set of search agents in the final WOA population is utilized as the code smell classification rules. The best solutions for both medium and large systems are used to further analyze the software metrics that distinguish the nine code smell types listed in Section 2. Tab. 7 provides the analysis results for both large and medium software systems used in experiments. Each type is characterized by one, two, or three software metrics. The ranges described in Tab. 7 indicate the bins resulting from discretization of the search space. For example, long method code smell (LM) is defined by the presence of McCabe numbers greater than or equal to 40 and greater than 15 for large and medium systems, respectively.

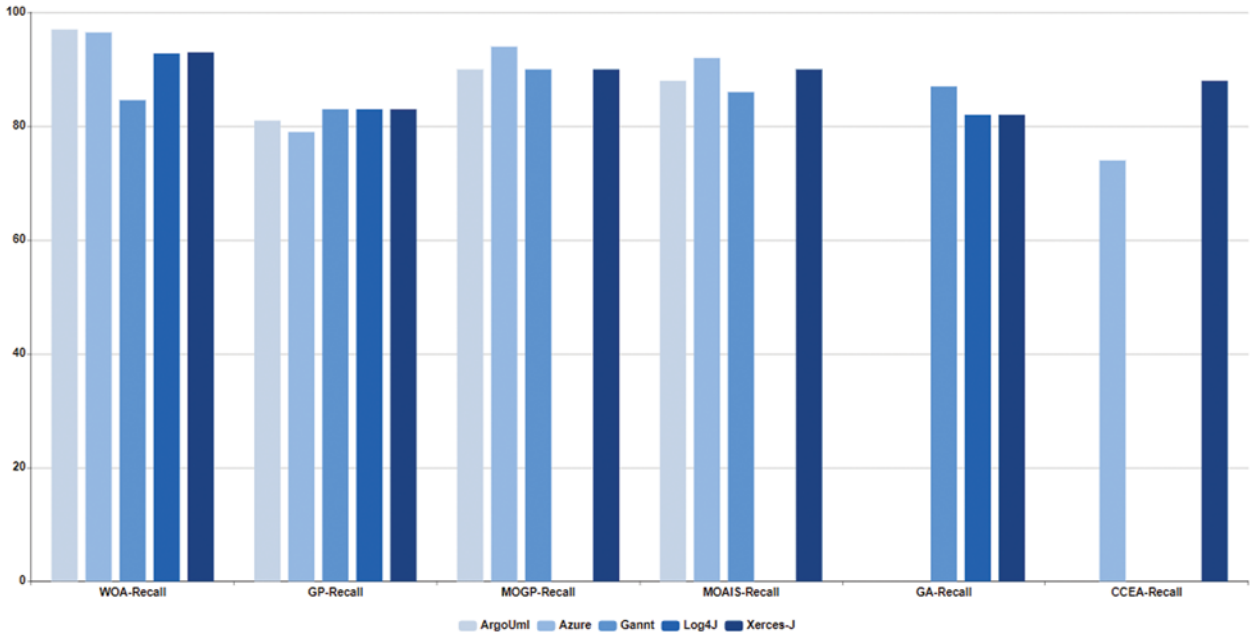

Figure 5: A comparison between WOA and other search-based algorithms using recall measure

Table 7: The detection rules of code smell types for both large and medium systems

Code smell type	Detection rules for large systems	Detection rules for medium systems
Large Class (Blob)	LOC >= 2700 ‖ NAD > 30	LOC > 200 ‖ NAD > 9
Spaghetti Code (SC)	NOC > 40	NOC > 17
Feature Envy (FE)	CBO >= 45 ‖ LCOM >= 75	CBO >= 10 ‖ LCOM >= 30
Long Method (LM)	McCabe >= 40	McCabe > 15
Data Class (DC)	WMC >= 133 ‖ LCOM > 110	WMC > 40 ‖ LCOM > 55
Long Parameter List (LPL)	NOPARAM >= 5	NOPARAM > 3
Lazy Class (LC)	Nom < 2 ‖ (RFC < 20 && LOC < 100)	Nom == 0 ‖ (RFC < 2 && LOC <= 10)
Functional Decomposition (FD)	DIT > 3 ‖ NMO > 4	DIT > 1 ‖ NMO > 0
Parallel Inheritance (PI)	DIT > 6 ‖ NOC > 17	DIT > 2 ‖ NOC > 4

6 Conclusion and Future Work

Due to additional customer requests, software houses have to implement continuous change in software systems. Hence, it is necessary to avoid software engineering problems that arise during SDLC by discovering and determining the appropriate solutions. Code smells are critical software design problems that emerge from continuous changes. This research detects code smell by finding the optimal classification rules that characterize them. WOA is utilized to search for the best classification rules using the exploration and exploitation processes. The Fisher criterion guides the WOA through the search by measuring the fittest probable classification rules after matching with the training dataset. The proposed framework detects nine types of code smell on five open-source Java software projects. Experimental results demonstrate a precision of 94.42% and recall of 93.4%, which are enhanced over the previous techniques in the literature. The proposed framework provides an efficient and feasible detection model that increases software quality while minimizing maintainability time, expenses, and efforts. Additionally, our framework can define classification rules for these types in medium and large systems; they are further analyzed to distinguish the software metrics characterizing each type of the detected code smell.

Future work includes verifying the validity of this framework with other types of code smells and testing it on more datasets. Another direction for future work is automatic correction of different types, which would be of a great help to software development teams and would minimize the SDLC time and expenditure.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] B. Vogel-Heuser, A. Fay, I. Schaefer and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *The Journal of Systems and Software*, vol. 110, pp. 54–84, 2015.

[2] M. Agnihotri and A. Chug, "Application of machine learning algorithms for code smell prediction using object-oriented software metrics," *Journal of Statistics and Management Systesms*, vol. 23, no. 7, pp. 1159–1171, 2020.

[3] G. Lacerda, F. Petrillo, M. Pimenta and Y. G. Gueheneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, pp. 1–36, 2020.

[4] I. M. Umesh and D. Srinivasan, "A study on bad code smell," *Ijltemas Journal*, vol. 4, no. 5, pp. 11–13, 2015.

[5] M. Fowler, K. Beck, J. Brant, W. Opdycke and D. Robert, "Bad smells in code," in *Refactoring: Improving the Design of Existing Code*, 2nd ed. Boston, US: Addison-Wesley, pp. 75–87, 2018.

[6] F. Palomba and A. Zaidman, "The smell of fear: On the relation between test smells and flaky tests," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2907–2946, 2019.

[7] F. Pecorelli, F. Palomba, D. D. Nucci and A. D. Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *IEEE/ACM 27th Int. Conf. on Program Comprehension*, Montreal, QC, Canada, pp. 93–104, 2019.

[8] U. Mansoor, M. Kessentini, B. R. Maxim and K. Deb, "Multi-objective code smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, 2017.

[9] N. Moha, Y. G. Gueheneuc, L. Duchien and A. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[10] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. G. Gueheneuc *et al.,* "Smurf: A svm-based incremental anti-pattern detection approach," in *19th Working Conf. on Reverse Engineering*, Kingston, Ontario, Canada, pp. 466–475, 2012.

[11] J. Aldallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231–249, 2015.

[12] D. Sahin, M. Kessentini, S. Bechikh and K. Deb, "Code smell detection as a bilevel problem," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 1, pp. 1–44, 2014.

[13] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto and A. D. Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," in *IEEE Int. Conf. on Software Maintenance and Evolution*, Victoria, BC, Canada, pp. 101–110, 2014.

[14] M. Harman, S. A. Mansouri and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 1–61, 2012.

[15] S. Mirjalili and A. Lewis, "The whale optimization algorithm," *Advances in Engineering Softwares*, vol. 95, pp. 51–67, 2016.

[16] H. Guo, Q. Zhang and A. K. Nandi, "Feature generation using genetic programming based on fisher criterion," in *15th European Signal Processing Conf.*, Poznan, Poland, 2007.

[17] M. I. Azeem, F. Palomba, L. Shi and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[18] G. Travassos, F. Shull, M. Fredericks and V. R. Basili, "Detecting defects in object-oriented designs: Using reading techniques to increase software quality," *ACM Sigplan Notices*, vol. 34, no. 10, pp. 47–56, 1999.

[19] O. Ciupke, "Automatic detection of design problems in object-oriented reengineering," in *Proc. of Technology of Object Oriented Languages and Systems*, Santa Barbara, CA, USA, pp. 18–32, 1999.

[20] G. Ganea, I. Verebi and R. Marinescu, "Continuous quality assessment with in code," *Science of Computer and Programming*, vol. 134, pp. 19–36, 2017.

[21] J. Dexun, M. Peijun, S. Xiaohong and W. Tiantian, "Detection and refactoring of bad smell caused by large scale," *International Journal of Software Engineering & Applications*, vol. 4, no. 5, pp. 1–13, 2013.

[22] R. Naveen, "Jsmell: A bad smell detection tool for java systems," M.S. Thesis. Maharishi Dayanand University, Rohtak, India, 2009.

[23] A. Rani and H. Kaur, "Detection of bad smells in source code according to object oriented metrics," *International Journal for Technological Research in Engineering*, vol. 1, no. 10, pp. 1211–1214, 2014.

[24] N. Moha, Y. Gueheneuc, A. Meur, L. Duchien and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Formal Aspects of Computing*, vol. 22, no. 3, pp. 345–361, 2010.

[25] N. Mathur, "Java smell detector," M.S. Thesis. San Jose State University, San Jose, California, USA, 2011.

[26] F. Simon, F. Steinbruckner and C. Lewerentz, "Metrics based refactoring," in *Proc. Fifth European Conf. on Software Maintenance and Reengineering*, Lisbon, Portugal, pp. 30–38, 2001.

[27] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proc. of the 5th Int. Sym. on Software Visualization*, New York, YK, USA, pp. 5–14, 2010.

[28] H. Grodzicka, A. Ziobrowski, Z. Lakomiak, M. Kawa and L. Madeyski, "Code smell prediction employing machine learning meets emerging java language constructs," *Data-Centric Business and Applications*, vol. 40, pp. 137–167, 2020.

[29] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, vol. 28, no. 3, pp. 1063–1086, 2020.

[30] N. Pritam, M. Khari, L. H. Son, R. Kumar, S. Jha *et al.,* "Assessment of code smell for predicting class change proneness using machine learning," *IEEE Access*, vol. 7, pp. 37414–37425, 2019.

[31] M. Kessentini, H. Sahraoui, M. Boukadoum and M. Wimmer, "Search based design defects detection by example," in *Int. Conf. on Fundamental Approaches to Software Engineering*, Saarbrucken, Germany, pp. 401–415, 2011.

[32] A. Ouni, M. Kessentini, H. Sahraoui and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[33] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh and S. Chikha, "Competitive coevolutionary code smells detection," in *Int. Sym. on Search Based Software Engineering*, Petersburg, Russia, pp. 50–65, 2013.

[34] JDeodorant, "Eclipse marketplace," 2020. [Online]. Available: http://marketplace.eclipse.org/content/jdeodor.

[35] S. M. Adnan, M. Ilyas, S. Razzaq, F. Maqbool, M. Wakeel *et al.,* "Code smell detection and refactoring using AST visitor," *Technical Journal*, vol. 25, no. 1, pp. 59–65, 2020.

[36] A. P. Mathew and F. Capela, "An analysis on code smell detection tools," in *17th SC@ RUG 2020*. Groningen, Netherlands, pp. 57–62, 2020.

[37] R. Ibrahim, M. Ahmed, R. Nayak and S. Jamel, "Reducing redundancy of test cases generation using code smell detection and refactoring," *Journal of King Saud University—Computer and Information Sciences*, vol. 32, no. 3, pp. 367–374, 2020.

[38] M. Harman and B. F. Jones, "Search based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[39] M. Harman, J. Krinke, J. Ren and S. Yoo, "Search based data sensitivity analysis applied to requirement engineering," in *Proc. of the 11th Annual Conf. on Genetic and Evolutionary Computation*, Montreal, Canada, pp. 1681–1688, 2009.

[40] F. A. Fontana, M. V. Mantyla, M. Zanoni and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[41] M. Lanza and R. Marinescu, "Evaluating the design," in *Object-Oriented Metrics in Practice*, 1st ed. Berlin, Germany: Springer-Verlag Berlin and Heidelberg GmbH & Co. KG, pp. 45–72, 2006.

[42] ArgoUML, "ArgoUML-tigris," 2020. [Online]. Available: http://argouml.tigris.org/.

[43] Azure, "Vuze bittorrent client," 2020. [Online]. Available: http://vuze.com.

[44] Gantt, "Gantt project," 2020. [Online]. Available: www.ganttproject.biz.

[45] Log4j, "Log4j," 2020. [Online]. Available: http://logging.apache.org/log4j/2.x/.

[46] Xerces-J, "Xerces java parser," 2020. [Online]. Available: http://xerces.apache.org/xerces-j/.

[47] RapidMiner Studio, "RapidMiner," 2020. [Online]. Available: https://rapidminer.com/get-started/.

[48] J. R. Sherrah, R. E. Bogner and A. Bouzerdoum, "The evolutionary pre-processor: Automatic feature extraction for supervised classification using genetic programming," in *Genetetic Programming Proc. Second Annual Conf.*, New Jersey, NJ, USA, pp. 304–312, 1997.

[49] M. Kotani, S. Ozawa, M. Nakai and K. Akazawa, "Emergence of feature extraction function using genetic programming," in *Int. Conf. Knowledge-Based Intelligent Electronic Systems Proceedings*, Adelaide, SA, Australia, pp. 149–152, 1999.

[50] A. Kaveh and M. I. Ghazaan, "Enhanced whale optimization algorithm for sizing optimization of skeletal structures," *Mechanics Based Design of Structures and Machines*, vol. 45, no. 3, pp. 345–362, 2017.

[51] S. K. Cherukuri and S. R. Rayapudi, "A novel global MPP tracking of photovoltaic system based on whale optimization algorithm," *International Journal of Renewable Energy Development*, vol. 5, no. 3, pp. 225–232, 2016.

[52] H. J. Touma, "Study of the economic dispatch problem on ieee 30-bus system using whale optimization algorithm," *International Journal of Engineering Technology and Sciences*, vol. 5, no. 1, pp. 11–18, 2016.

[53] M. A. ElAziz, A. A. Ewees and A. E. Hassanien, "Whale optimization algorithm and moth-flame optimization for multilevel thresholding image segmentation," *Expert Systems with Applications*, vol. 83, no. 3, pp. 242–256, 2017.

[54] F. E. Junior and G. G. Yen, "Particle swarm optimization of deep neural networks architectures for image classification," *Swarm and Evolutionary Computation*, vol. 49, no. 5, pp. 62–74, 2019.

[55] S. M. Elsayed, R. A. Sarker and D. L. Essam, "A new genetic algorithm for solving optimization problems," *Engineering Applications of Artificial Intelligence*, vol. 27, pp. 57–69, 2014.

[56] NetBeans IDE, "NetBeans IDE," 2020. [Online]. Available: https://netbeans.org/downloads/8.2/rc/.

[57] Java Development Kit, "Java development kit," 2020. [Online]. Available: https://www.oracle.com/java/technologies/javase-jdk11-downloads.html.

[58] Java Excel API Jar, "Java excel api," 2020. [Online]. Available: http://www.java2s.com/Code/Jar/j/Downloadjxl26jar.htm.

[59] Apache Poi, "Apache poi," 2020. [Online]. Available: http://poi.apache.org/download.html.

[60] M. Gong, L. Jiao, H. Du and L. Bo, "Multiobjective immune algorithm with nondominated neighbor-based selection," *Evolutionary Computation*, vol. 16, no. 2, pp. 225–255, 2008.

[61] D. E. Goldberg, "Introduction to genetics based machine learning," in *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley, pp. 217–259, 1989.