Indian Journal of Science and Technology, Vol 11(20), DOI: 10.17485/ijst/2018/v11i20/122472, May 2018
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Prioritizing Code Smell Correction Task using Strength Pareto Evolutionary Algorithm

G. Saranya[1], H. K.Nehemiah[1]* A. Kannan[3] and V. Pavithra[4]

[1]Ramanujan Computing Centre, College of Engineering Guindy, Anna University, Chennai - 600025, Tamil Nadu, India; nehemiah@annauniv.edu
[3]Information Science and Technology, College of Engineering Guindy, Anna University, Chennai - 600025,Tamil Nadu, India
[4]Computer Science and Engineering, College of Engineering Guindy, Anna University, Chennai - 600025, Tamil Nadu, India

Abstract

Objective: Code smells indicate the design decay in software applications. The code smells existence in the software will hinder the understandability of code and possibly increases changes and fault proneness. **Methods / Statistical Analysis**: To remove the code smells' from the software applications refactoring operations are applied which in turn improves the software system structure without changing its overall behaviour. Generally, in a large sized system, code smell cannot be fixed automatically. Therefore based on the maintainer's preference, the prioritized list of refactoring sequences to fix the code smells is essential. **Findings:** Majority of the refactoring just rely on the structural information, which fails to preserve the construct semantics, minimization of changes and the use of development history. To overcome this, in this work, the Strength Pareto Evolutionary Algorithm (SPEA) is used to prioritize the list of refactoring operations that maximize the quality improvement, constructs semantics coherence and preserving the consistency with the previous refactoring. This work is carried out on two open source software Xerces-J and J Hot Draw. Blob, shotgun surgery, functional decomposition, data class, Swiss army knife and schizophrenic class code smells' are considered for prioritizing refactoring operations in these open source system. SPEA is evaluated using the metrics Code smell Correction Ratio (CCR) and Refactoring Meanings (RM). **Application / Improvements:** SPEA is compared with other algorithms namely Non-dominated Sorting Genetic Algorithm II (NSGA II) and Chemical Reaction Optimization (CRO), to prove its efficiency in prioritizing code smell correction tasks.

Keywords: Code Smells, Maintenance, Prioritizing, Refactoring, Search Based Software Engineering, Strength Pareto Evolutionary Algorithm (SPEA)

1. Introduction

Software plays a vital role in all areas of human effort to produce progression. A software system consists of programs, configuration files, project files, and system documentation and user documentation files[1]. The software systems design exhibit several problems during the initial construction or during software systems aging, where the system quality degenerates over time. Some researchers view these issues in software design either as noncompliance with the system design principles or violations of software design heuristics. One such major design problem in software systems is the code smells[2]. Code smells indicate poor software design and different system implementation choices that hinder code comprehension which possibly increases the changeability and liability of faults in the software[3]. Such symptoms may originate from activities performed by developers during the submission of project. Generally code smells indicate the weakness in design that may increase the risk of failure in future.

Removing the code smells is done by an appropriate refactoring, i.e. an improvement in the software system structure without any of the modification of its behaviour. Moreover, the simplicity and the cumulative effect of successive refactoring solutions on the software design problem can be significant. Detecting the code smells in the modules of the software and identifying the correct refactoring operation for the detected code smell is

*Author for correspondence

a very challenging task. Generally for the large sized system, not all code smells can be fixed automatically; fixing the refactoring operations is very large for the number of code smells' detected. Therefore based on the maintainer's preference, the prioritized list of refactoring sequence is needed to fix the code smells for the large systems.

In this work, for the number of detected code smells', a sequence of refactoring operations is suggested based on the maintainer's preferences using multi objective optimization approach Strength Pareto Evolutionary Algorithm (SPEA)[4]. The sequence of refactoring operation preserves the construct semantics, maximize the code smells correction and use the development history as the powerful source for maintenance tasks.

The paper is organised as follows: In Section 2, a brief description of the existing literatures such as code smell detection and correction are discussed. Section 3 presents the methodology for the sequence of refactoring operations to the detected code smell using SPEA. Empirical study definition and design is discussed in section 4. In section 5, the analysis of experiments and their results are explained; Section 6 presents the threats that could affect the validity of the sequence of refactoring operation using SPEA. Finally, conclusion and scope for future work are discussed in section 7.

2. Literature Survey

The code smell detection and correction task has been addressed in the literature from different perspectives. Those detection and correction of code smells approaches are discussed in this section.

Refactoring was first introduced by[5], who provided a catalogue of refactoring that could be applied in specific situations. Using the concept of refactoring the software quality aspects such as maintainability, extensibility and reusability are improved[3].

The researchers' in[6] investigated and suggested the use of quality metrics for design flaws detection and correction. The researchers used inheritance and coupling metrics namely, number of methods inherited, number of children, number of methods added, coupling between object classes, number of methods overridden, data abstraction coupling, others method-method export coupling to detect the design flaws. Finally, the researchers developed a tool called OO1 for correction. This tool uses a transformation that is applied automatically in order to improve quality estimated by the metrics. The limitation of this tool is, it requires some form of human intervention and acknowledgement before applying the suggested transformation.

The researchers' in[7] analyzed the circumstances in which refactoring operations improve the source code by having high cohesion and low coupling. The researchers also presented a guideline for applying refactoring under these circumstances. This guideline was validated in the Apache Tomcat open source software system which exhibits cohesion and coupling characteristics. The authors in the study[8] proposed a fully automatic approach for code smell correction technique based on relational concept analysis (RCA). The proposed work RCA is extended from the Formal Concept Analysis (FCA). The framework RCA is combined with the cohesion and coupling metrics to suggest the refactoring operations. To validate this approach the researchers used four open source software namely, Azureus, Log4J, Lucene and Nutch. The approach of automatic suggestion of refactoring operation is illustrated using the design defect blob. The drawback of this approach is the association between the design defect detection and correction is not obvious, which is difficult to the software maintainers.

The authors in the study[9] presented a concept lattice based approach for identifying least cohesive classes and guidelines for refactoring operation. Using this approach, the researchers suggest the guidelines for refactoring operation such as, move method, extract class, remove unused attribute and localize attributes to the less cohesive classes. This approach was validated using cohesion and coupling metrics.

The researchers' in[10] proposed a framework for detection and correction of design defects. The researchers investigated the list of object oriented metrics for detecting the design defect and suggested refactoring operation for correcting them. This framework is illustrated on the Java Expert System Shell (JESS) rule engine.

In a study, the researchers'[11] presented an approach to progress the reconstruction of refactoring over an integrated development environment. To improve the reconstruction of refactoring, the researchers used graph transformation approach to detect the changes. They compared their approach with Ref-Finder[12]. The approach indentifies Move method and Rename method more accurately than Ref-Finder. The limitation of this work is, the developers require a change-recorded plug in to detect the changes.

Other than manual and semi-automated approaches, the researchers[13] used program invariants to discover refactoring candidates. The researcher developed an invariant pattern matcher to infer the refactoring operation. This work is applied to Nebulous a component of aspect browser.

In[14] the researchers' developed a tool called ROSE to predict future changes and give warning about the missed changes in software. To perform the prediction regarding changes, the ROSE tool needs the version history of the software which is under maintenance. This tool retrieves the changes from the version histories and uses Apriori algorithm to compute the association rules for those changes. The ROSE tool is evaluated using eight large open source software's namely, Eclipse, GCC, GIMP, JBOSS, JEDIT, KOFFICE, POSTGRES and PYTHON. This approach was validated using the metrics precision and recall. The average precision is above 50% and the average recall is above 70 %.

Authors' in[15] proposed formulation of refactoring operations for the number of detected code smells based on graph transformation. The graph transformation is limited only to the structural and syntactic information. The researchers used Ant Colony Optimization (ACO) algorithm for the formulation of refactoring operation. The approach ACO is evaluated on the Local Area Network (LAN) presented by[16] for analyzing the steps in refactoring operation operations.

The researchers' in[17] presented an approach for suggesting the refactoring operations with all necessary conditions using genetic algorithm. The researchers used a fitness function that depends on the existing set of object oriented metrics such as Response For Class (RFC), Information flow based Coupling (ICP), Tight Class Cohesion (ICH), Lack of Cohesion (LCOM5), Weighted Method Count (WMC) and Number of Methods (NOM). The approach is evaluated on the open source software J Hot Draw.

In[18] proposed a sequence of refactoring operation for code smell correction using Chemical Reaction Optimization (CRO) a met heuristic search based approach. Using this approach, the maintainers maximize the number of refactoring solution to the code-smell fixed by software maintainers. The approach CRO is compared with other approaches such as Genetic Algorithm (GA), Simulated Annealing (SA) and PSO. The approach CRO outperforms other approaches in terms of four prioritization measures such as severity, risk, priority and importance of code smells. CRO was evaluated on five open source system namely, Xerces-J, Jfree chart, Gantt project, J Hot Draw and Artofillusion and seven types of code smell were considered namely, blob, data class, spaghetti code, functional decomposition, schizophrenic class, shotgun surgery and feature envy. The approach when tested with five open source systems yielded an accuracy of 90% for all the code smells fixed.

The researchers' in[19] defined search based refactoring approaches based on Genetic Algorithm (GA), Genetic Programming (GP), parallel evolutionary algorithm and Non dominated Sorting Genetic Algorithm (NSGA II). To improve the search based refactoring the researchers has used Pareto optimality with metrics to give sequence of refactoring operations. The concept of Pareto optimality was evaluated on three open source software namely, J Hot Draw, Maven and XOM. The drawback of this approach is the Pareto optimal search focused explicitly on suggesting the refactoring, not on detecting the defects in the code.

In another study, the researchers' in[20] proposed an approach REMODEL based on genetic programming and a set of software metrics to automatically generate the most appropriate set of refactoring to the software design. The researchers used Quality Model for Object Oriented Design (QMOOD)[21] a suite of OO metrics to improve the quality of the software design and introduce design patterns to improve the maintainability of the software design. REMODEL was validated using Repository for Model-Driven Development (REMODD) a web based software system proposed by[22] that support research and education in Model-Driven Development (MDD). The limitation of the approach is, the researches focused mainly on suggesting the suitable refactoring operation and the design defects are not detected explicitly.

The authors' in the study[23] described how to predict refactoring operations for open source system on short time duration. To predict the refactoring operation, a process undergoes three stages such as, data understanding, pre-processing and classifiers. To predict the refactoring the researchers used data mining algorithms such as J48, Logistic Model Tree (LMT), Repeated Incremental Pruning (RIP) and Nearest Neighbor Generalization (NNge). The researchers evaluated the prediction model using 10-fold cross validation on two open source software namely, Argo UML and spring framework.

3. Proposed Method

This proposed work is to prioritize the code smell correction tasks using Strength Pareto Evolutionary Algorithm (SPEA)[4]. The objective is to improve the software quality and maintenance of the object oriented software system by identifying the code smells in the classes placed in package of the software and to find the optimal refactoring solution to fix the code smells.

3.1 Strength Pareto Evolutionary Algorithm (SPEA)

The SPEA algorithm was proposed by Zitler and Thiele in 1998 for solving the real world optimization problems, such as network optimization problem, the grid scheduling problem, Quadratic Assignment Problem (QAP), routing problem and Resource Constrained Project Scheduling Problem (RCPSP). SPEA is a Meta heuristic search based optimization algorithm. The notion of SPEA is to make a non dominated solution or pareto set from the population (sequence of refactoring solution) to solve multi objective optimization problem. Using this algorithm, it is easy to find the near optimal solution, which is called as the non-dominated solution.

In this work, refactoring solutions is considered as an individual and are represented as vectors with the order of application of the refactoring operations corresponding to the positions in the vector. For each refactoring operation, the controlling parameters such as classes/methods/fields are randomly selected from the source of the software system to be refactored. The initial population for SPEA is generated, by randomly choosing refactoring solutions containing sequences of refactoring operations. The description about the fitness functions and the operations used are as follows.

3.2 Fitness Functions

Semantic similarity fitness and history of change fitness functions are used in this work to assess the quality of the individuals.

3.2.1 Semantic Similarity Fitness

The semantic similarity fitness is measured with values obtained from the structural similarity and the semantic coherence value.

$$SemanticSimilarityfitness = \frac{1}{n}\sum_{i=0}^{n-1} SRH(RO_i) \quad (2)$$

Where, *SRH= SimilarityRefactoringHistory*

$$SRH(RO) = \sum_{i=1}^{n} W_{RO_i} * contextsimilarity(RO, RO_i) \quad (3)$$

Where n is the number of recorded refactoring operations collected from different software systems. w_i is a refactoring weight that reflects the similarity between the recommended refactoring operation RO and the recorded refactoring operation RO_i.

$$CS(RO, RO_i) = \alpha + \frac{(CBOSim(C_{RO}, C_{RO_i}) + DSim(C_{RO}, C_{RO_i}))}{2} + \beta * SS(C_{RO}, C_{RO_i}) \quad (4)$$

Where, *CS* is the contextsimilarity and *SS* is the semantic similarity. The contextsimilarity aims at calculating the context similarity between the refactoring operation *RO* applied to the code fragment C_{RO} and the refactoring operation RO_i applied to the code fragment C_{ROi}. And *CBOSim* refers to the coupling between objects similarity, *DSim* refers to the dependency based similarity. Where α and β are the coefficients for the two components representing the structural and semantic similarities and α+β=1.

$$CBOSim(C_{RO}, C_{RO_i}) = \left| CBO(C_{RO}) - CBO(C_{RO_i}) \right| \quad (5)$$

$$DSim(C_{RO}, C_{RO_i}) = \left| coup(c_1, c_2) - coup(c'_1, c'_2) \right| \quad (6)$$

where $coup(c_1,c_2)$ returns the number of relationships between the two classes c_1 and c_2 and $coup(c'_1,c'_2)$ returns the number of relationships between the two classes c'_1 and c'_2.

$$Semantic\ sim(C_{RO}, C_{RO_i}) = \left| sim(c_1, c_2) - sim(c'_1, c'_2) \right| \quad (7)$$

where $sim(c_1,c_2)$ returns the cosine similarity between the two classes c_1 and c_2 and it is calculated as follows

$$sim(c_1, c_2) = \cos(\vec{c_1} . \vec{c_2}) = \frac{\vec{c_1} . \vec{c_2}}{\left\| \vec{c_1} \right\| * \left\| \vec{c_2} \right\|} = \frac{\sum_{i=1}^{n} (W_{i,1} * W_{i,2})}{\sqrt{\sum_{i=1}^{n} (W_{i,1})^2} \sqrt{\sum_{i=1}^{n} (W_{i,2})^2}} \quad (8)$$

Where $\vec{c_1}$ is the term vector corresponding to the class c_1 and $\vec{c_2}$ is the term vector corresponding to the class c_2. The weights W_{ij} is computed using information retrieval based technique named as term-frequency-inverse term frequency method.

3.2.3 History of Changes Fitness

The History of change fitness function[24] is defined as the number of changes applied in the past to the same code elements which is shown in the Eq. (9)

$$HistoryMeasure(RO_i) = \sum_{i=1}^{n} t(e) \quad (9)$$

Where, *t(e)* is the number of times the code element *e* was refactored in the past, and *n* is the number of possible refactoring operations. If this number is high, it is a good indication that this code element is badly designed, thus representing a refactoring opportunity.

3.3 Crossover and Mutation of the Individuals

Single random crossover is used in SPEA to improve the quality of prioritization of refactoring operations[4]. Two refactoring solutions for crossover are selected randomly. From them, the refactoring operations are interchanged to produce two new off springs. The crossover operation on the refactoring operation is shown in the Figure 1.

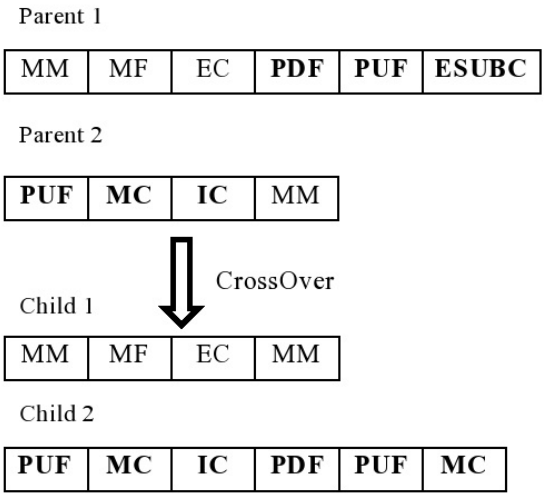

Figure 1. Crossover applied for the selected refactoring operations.

The refactoring solutions for mutation are selected randomly. From them, the position of the refactoring operation and their controlling parameters in the vector is interchanged. For interchanging the position in the vector the pre and post conditions are used. Figure 2 shows the effect of the mutation operation.

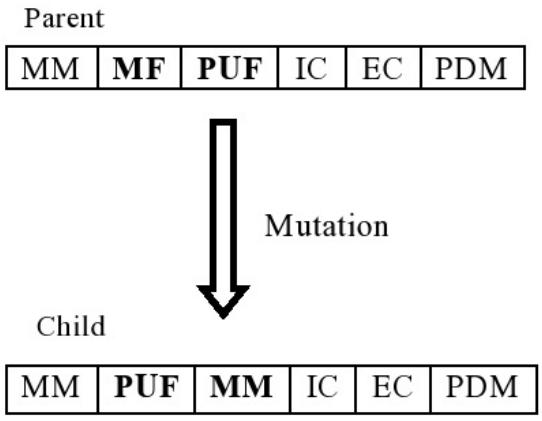

Figure 2. Mutation applied for the crossover individuals.

In the above operations, MM refers to the move method, MF to the move field, EC to the extract class, PDF to the push down field, PUF to the pull up field, ESUBC to the extract subclass, PDM to the push down method and IC to the inline class. All these represent the different refactoring operations.

3.4 Adaptation of SPEA for Prioritizing Code Smell Correction Task

Input: *Initial Model, Set of Quality metrics, Set of design defect examples*

Output: *Best sequence of refactoring solutions*

Generate an initial population P /* sequence of refactoring operations with the controlling parameters */

Create an empty pareto optimal $\overline{P} = \varphi$

Repeat

While stopping criteria not reached **do**

P' =fast_non_dominated_sort (P) /* Calculate the external pareto solution from the current population*/

$P'' = P' + \overline{P}$ /* Extended pareto set is obtained*/

While ($P'' > N$) **do**

Calculate reduce extended pareto set by clustering (P'') and go to line 3.
End
$\overline{\overline{P}} = P'' + P$ /*extended population is obtained */
For all i in $\overline{\overline{P}}$ do
Calculate the Semantic similarity fitness and change fitness function for the extended population ($\overline{\overline{P}}$)
End for
Compare the fitness of individuals (refactoring solutions)
Perform single random crossover
Perform random mutation
End
P: = generate new_ population (P)
It=it+1;
Until it=max_it or fitness function (best solution);
Return best solution
End

The algorithm description is as follows: In line 1-2, the algorithm generates randomly, a sequence of refactoring operations with controlling parameters as initial population P and creates an empty pareto optimal set $\overline{P}$. In line 3-6, the external pareto solution P' is calculated and the extended pareto set P'' is formed. The external pareto solution is calculated using the fast_non_dominated_sort from the current population P . Then the refactoring solution from the fast_non_dominated_sort P' is combined with the empty pareto optimal set $\overline{P}$, to obtain the extended pareto set P'' . In line 7-9, if the extended pareto set is greater than the maximum size of the entire population N then reduce the set with use of hierarchical clustering algorithm and then go to line 3 else go to line 10. In line 10, the extended pareto set P'' combines with the population P to create the extended population $\overline{\overline{P}}$. Line 11-13 calculates the fitness functions such as Semantic similarity fitness and change fitness function for the extended population. In Line 14 two individuals are selected randomly to compare their fitness value and select the best fitness to the matting pool. From line 15-19 the individuals are selected randomly from the matting pool to perform the genetic crossover and mutation operation according to their probabilities to generate the new population. The algorithm terminates after the given maximum number of iterations or the best_solution (line 20) and returns the best_solution (line 21).

3.5 Fast Non Dominated Sort Algorithm

Initially the SPEA algorithm creates an empty pareto-optimal set externally. This external pareto-optimal set is used to evaluate each individual (refactoring solution) in the population. At each point of time the external pareto set contains the non-dominated solutions of the search space. The goal of the algorithm is to find new non dominated solutions. The individuals are evaluated in dependence of the number of pareto points by which they are covered. First the fast non dominated sort is calculated from the initial population to update the extended pareto population. The pseudo code of the fast non dominated sort algorithm is as follows[25].

Input: *Population P*
Output: *Non-dominated fronts* F_i
P' =fast_non_dominated_sort (P)
For each refactoring solution, p ∈ P
$S_p = \phi$ /* A set of refactoring solution which the sominates */
$n_p = 0$ /* Number of refactoring solution which dominates the solution p*/ **for each** refactoring solution, q ∈ P
If (p ≺ q) **then** /*If p dominates q*/
$S_p = S_p \cup \{q\}$ /*add q to the set S_p*/
Else if (q ≺ p) **then** /*if p is dominated by q*/
$n_p = n_p + 1$ /*Increment n_p*/
End for
If $n_q = 0$ **then** /*If no refactoring solution dominates p*/
$p_{rank} = 1$
End for
$F_1 = F_1 \cup \{p\}$ /*p is a refactoring solution of the first front*/
i = 1 /*Initialize the front counter*/
While $F_i \neq \varphi$
$Q = 0$ /*used to store the solutions of the next front*/
For each $p \in F_i$ /*for each solution p in F_i*/
For each $q \in S_p$ /*modify each solution from the set S_p*/
$n_q = n_q - 1$ /*decrement n_q by one*/
if $n_q = 0$ **then** /*if n_q is zero then q is a solution of a list Q*/
$q_{rank} = i + 1$
$Q = Q \cup \{q\}$
End for
i = i + 1
End for
$F_i = Q$ /*Current front is formed with all refactoring solution of Q*/
End

For each refactoring solution the two fitness functions namely, semantic similarity fitness and change fitness

are calculated and compared with another refactoring solution. If the refactoring solution p dominates another refactoring solution q then add q in the set S_p, a set of refactoring solutions where the refactoring solution p dominates. If the refactoring solution q dominates p then increment n_p, the number of refactoring solution which dominates the refactoring solution p. Identify the refactoring solution which does not dominate the solution p have $n_p=0$ and $P_{rank}=1$and put those solutions in a list F_1 called as current front. Now, for each refactoring solution in F_1 visit each solution q in its set S_p and reduce its n_q count by one. By doing so, for any refactoring solution in q the count becomes zero, we put all these solution in a new list Q. When all the refactoring solution of the current front has been checked, it is declared that the refactoring solution in the list F_1 has solution of the first front. To continue this process the newly identified front Q is used as the current front. This process continues until all fronts are identified.

3.6 Clustering Algorithm

In this work, the Pareto optimal set can have extremely large number of solutions which depends on the size of the software system. An average linkage based hierarchical clustering algorithm is used to reduce the Pareto set. This clustering algorithm works iteratively by combining the adjacent clusters until the required number of groups is obtained. The algorithm is given in the following steps.

Step 1: Initialize cluster set C; each individual (refactoring solution) i ∈ P constitute a distinct cluster.

Step 2: If number of clusters |C|≤ N the total population set (sequence of refactoring operation), then go to Step 5, and else go to Step 3.

Step 3: Calculate the distance of all possible pairs of clusters. The distance dxy between two clusters C1 and C2 is defined as the average Euclidean distance of all pairs of solutions (x ∈ C1 and y ∈ C2). It is calculated using the following Eq.(10):

$$d_{xy} = \frac{1}{n_1 * n_2} \sum_{x \in c_1, y \in c_2} d(x, y) \qquad (10)$$

Where, n_1 is the numbers of individuals (refactoring solutions) in clusters C1 and n_2 is the numbers of individuals (refactoring solutions) in clusters C2

Step 4: Determine two clusters with minimal distance dxy. Merge these clusters together. This reduces the number of clusters by one. Go to Step 2.

Step 5: For each cluster, the centroid should be found and the nearest refactoring solutions are selected to that centroid and then all other individuals (refactoring solutions) from the clusters are removed.

Step 6: Compute the reduced non dominated set by uniting the representatives of the clusters.

Output: Optimal set of refactoring suggestions is provided

4 Empirical Study Definitions and Design

The ***goal of this study*** is to observe the prioritized list of refactoring sequence using SPEA algorithm in software systems. The ***quality focus*** is on the correction accuracy of code smells using SPEA when compared to the correction accuracy of the same code smell with NSGA II[26,27] and Chemical Reaction Optimization (CRO)[18] approach, while the ***perspective*** of other researchers, who want to evaluate the effectiveness of the approach in prioritizing the list of refactoring operations using SPEA to build better recommenders for developers. The ***context*** of the study consists of two open source software, namely, Xerces-J and J Hot Draw. Xerces-J is a library completely written in Java for parsing, validating and manipulating XML documents. and J Hot Draw is an open-source project and is basically a Java GUI. One of the main intend behind its development was to port it to new Java GUI toolkits.

4.1 Research Question, Data Analysis and Metrics:

This work aims at addressing the following two research questions:

RQ1: How does SPEA perform when compared to another multi-objective algorithm?

RQ2: To what extent can the proposed approach recommend the refactoring operations in the situation where the change history is not available?

To answer RQ1, the authors compared SPEA with other multi-objective algorithms such as NSGA II and CRO using the same fitness function and the metrics Refactoring Meaningfulness (RM) and Code smell Correction Ration (CCR)[28]. The answer to this question is given in section

5.1. To answer RQ2, we compare the recommended refactoring operation in the situation where the change history is available and not available using the metric CCR.

4.2 Refactoring Meaningfulness (RM)

RM is shown in the Eq. (11) which gives the ratio of the number of meaningful refactoring operations, in terms of construct semantic coherence on total number of evaluated refactoring.

$$RM = \frac{Meaningful\,\mathrm{Re}\,factorings}{\mathrm{Pr}\,oposed\,\mathrm{Re}\,factorings} \quad (11)$$

4.3 Code smells Correction Ratio (CCR)

CCR is given in Eq. (12) that calculates the ratio of number of corrected code smells after applying the proposed refactoring sequence by the total number of code smells detected before applying the proposed refactoring sequence.

$$CCR = \frac{Number\ of\ corrected\_codeSmells}{Number\ of\ codeSmells\_before\_applying\_refactoring} \quad CCR \in (0,1) \quad (12)$$

4.4 Experimental Setup

This section describes in detail the subject, process and results of two case studies carried out for the Meta heuristic approach. The experiments are carried out using two different open source systems Xerces-J and J Hot Draw and their evaluation parameters are given in Table 1. Six different types of code smells' are detected from the open source systems are given in Table 2. The code smells used in this experiment are detected using in Fusion, I Plasma and DÉCOR tools. The number of code smells detected from the open source software with respect to the several versions are collected and tabulated below in Table 3. After detecting the code smell, the changes in each class are tracked with the use of G it Hub repositories and the change frequency is calculated using eq. (10). The similarity measures are calculated using the context similarity with corresponding similarity weights of the refactoring operations. Hence context similarity is tracked with the use of structural and semantic similarity using eq. (3). Finally, the quality fitness is calculated. Experimental results show the effectiveness of prioritizing code smell correction using SPEA approach with the normal detection strategy.

Table 1. Evaluation parameters for two open source systems

Metrics	Xerces-J	J Hot Draw
Lines of Code	21088	5280
Number of Classes	87	46
Number of Methods	698	433
Quality Deficit Index	351.8	36.2
Total No. of Children in all Classes	4	23
Number of Packages	13	1
Flawed Classes	3	2
Flawed Methods	46	6

5. Evaluation of Results

A preliminary evaluation of SPEA approach was performed on well-designed open-source system, namely

Table 2. Code smells description

Code Smells	Description
Blob	It is a class which implements many methods and declares several fields and operations with low cohesion.
Data Class	The class that has only variables and unused methods for accessing them. The purpose of the class is only to store the data and cannot independently operate on the data they own.
Swiss Army Knife	It is a complex class that provides more services. The class with more interfaces and inheritance.
Functional Decomposition	A class where the object oriented principles are poorly used. It has more private variables and single function methods.
Schizophrenic Class	It occurs in a class when a public interface of a class is larger and used non- cohesively by client methods.
Shotgun Surgery	It is a smell, when a change occurs in the source code then the developer has to change many classes and methods in source code.

Table 3. Detected code smells

Systems	Blob	Shotgun Surgery	Functional Decomposition	Schizophrenic Class	Swiss Army Knife	Data Class
Xerces J 2.6	16	5	0	7	25	9
Xerces J 2.7	20	10	0	9	15	10
Xerces J 2.8	20	11	41	6	8	17
JHotDraw 5.2	11	0	7	3	0	2
JHotDraw 5.3	13	4	23	4	4	8
JHotDraw 6.0	15	5	34	8	3	22

Xerces-J and J Hot Draw. SPEA Optimization has found promising results on two open source systems and six types of code-smell. The approach is more effective by prioritizing the code smell correction task with respect to the reference of prioritization and maintainer's preferences to automate the refactoring operation. The evaluation is aimed at investigating the correction accuracy of the code smells using SPEA approach. To test the accuracy of SPEA approach, the measures, namely, RM and CCR has been calculated.

5.1 Result of RQ1

To answer this question, this work uses the RM and CCR values to compare the existing approach NSGA II and CRO with the SPEA approach. In Table 4 the RM and CCR values for the two open source software are given. It is observed that the approach SPEA achieved good results when compared to the other approaches CRO and NSGA II in terms of CC and RM. Overall of the two studied projects J Hot Draw and Xerces-J, our approach SPEA provides 93% of CCR and 85% of RM, while CRO and NSGA II provides only 81%, 70% CCR and 77%, 71% RM respectively. For example, after applying the proposed prioritized refactoring operations, it is found that for J Hot Draw 21 out of 23 detected code smells were fixed with an average of 92% of CCR. At the same time, 90% of the refactoring operations were evaluated for J Hot Draw using the metric RM. There is a trade-off between the metrics CCR and RM: when CCR increases then RM decreases. This provides evidence that the quality of the refactoring solutions is in conflict with the construct semantics. Due to this reason, the SPEA multi-objective approach is used.

Table 4. Comparison of SPEA with CRO and NSGA II on two open source systems

Systems	Approach	CCR (%)	RM (%)
JhotDraw	SPEA	92 (21/23)	90
	CRO	79 (18/23)	79
	NSGA II	57 (1/23)	76
Xerces-J	SPEA	94 (99/105)	78
	CRO	84 (88/105)	75
	NSGA II	72 (75/105)	65
Average for all the Systems	SPEA	93	85
	CRO	81	77
	NSGA II	70	71

5.2 Result of RQ2

Table 5 presents the answer for this question. It is observed that the majority of suggested refactoring by our approach (where the change history is not available) succeeded in improving significantly the code quality with good correction scores. After applying the prioritized refactoring solutions, it is observed that SPEA achieved good CCR values when compared to NSGA II and CRO. The corrected code smells were of different types, which are discussed in Table 1. It is observed that the approach SPEA achieved superior CCR values with respect to the other approaches. For instance, shotgun surgery and Swiss army knife provides the same results of 0% in J Hot Draw for all the approaches. For the schizophrenic class our approach SPEA and CRO yielded the same result of 66% (2 out of 3) detected code smells were fixed in the open source software J Hot Draw. Where as in Xerces-J for the code smell functional decomposition 90% (10 out

Table 5. Code Smell Correction Ratio for the two systems

Systems	Approach	Code Smell Correction Ratio (CCR) %					
		Blob (%)	Functional Decom-position (%)	Shotgun Surgery (%)	Data Class (%)	Swiss Army Knife (%)	Schizo-phrenic Class (%)
JHotDraw	**SPEA**	90(10/11)	95(6/7)	0(0/0)	100(2/2)	0(0/0)	66(2/3)
	CRO	81(9/11)	57(5/7)	0(0/0)	50(1/2)	0(0/0)	66(2/3)
	NSGA II	64(7/11)	57(4/7)	0(0/0)	50(1/2)	0(0/0)	33(1/3)
Xerces-J	**SPEA**	100(20/20)	90(10/11)	95(39/41)	100(6/6)	88(7/8)	89(17/19)
	CRO	90(18/20)	90(10/11)	82(34/41)	83(5/6)	63(5/8)	84(16/19)
	NSGA II	75(15/20)	81(9/11)	73(30/41)	83(5/6)	50(4/8)	63(12/19)

of 11) detected code smells were fixed in both the approach SPEA and CRO. SPEA obtained better results when compared to NSGA II.

In this work the following refactoring operations are used such as move method, move field, extract class, extract interface, move class, pull up field, pull up method, push down field, inline class, push down method, extract subclass, extract super class and extract method. This study suggests that most of the refactoring operations are related to move method, move field, and extract class for two systems studied. The graphical representations of distributions of different refactoring types for the open source software J Hot Draw and Xerces-J are given in Figure 3,4.

From the graph it is inferred that the move method has the highest number of refactoring operation in both the open source systems J Hot Draw and Xerces-J. Then the move field, inline class and extract class has the highest refactoring operation for the open source system J Hot Draw, whereas for Xerces-J, the move field, extract class,

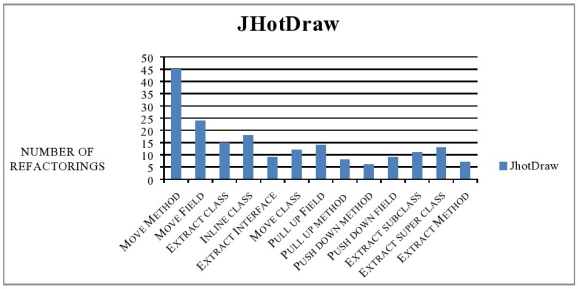

Figure 3. Suggested refactoring distribution for J Hot draw.

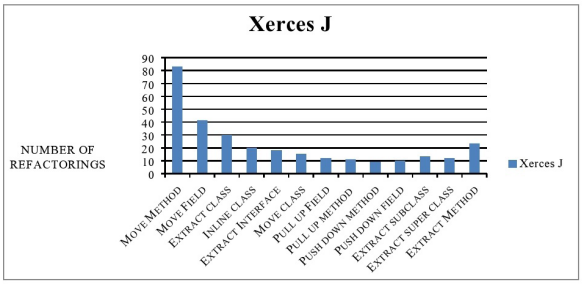

Figure 4. Suggested refactoring distribution for Xerces-J.

extract method and inline class has the highest number of refactoring operations.

6. Threats to Validity

External validity is the extendibility of the findings in other environment. In this work, the experiments are performed on two open source software, the evaluation parameters of two open source software's are described in Table 3. However, the result stated cannot be generalized to other programming languages, industrial applications and to other practitioners. To confirm the extendibility of this approach, future work is needed.

Construct validity is concerned with the relationship between theory and what is observed. Most of what is measured in this experiment are standard metrics such as refactoring meaningfulness and code smell correction ratio that are widely accepted for quality of code-smell correction solutions. In future, this approach will be compared using different meta-heuristics search algorithms.

7. Conclusion and Future Work

In this paper, Strength Pareto Evolutionary Algorithm is used in suggesting automatic refactoring solutions to fix the detected code-smells' with concern to the software maintainer's preferences. In this work, six different types of code smells are considered. Then the available refactoring operations are collected for the detected code smells. The fitness functions used in this optimization is based mainly on three objectives: maximize design quality, construct semantic preservation and the re-use of the history of changes applied to the similar contexts. For these detected code smells the refactoring solutions are obtained in a prioritized manner. SPEA optimization has found a promising result on two open source systems and six types of code-smell. The Prioritization of code smell correction task using SPEA is compared with other multi-objective Meta heuristics algorithm such as NSGA II and CRO, it is found that most of the detected code smells are corrected with a good correction score of 89% using the SPEA approach. As part of future work, it is planned to compare this study with different meta-heuristics search algorithm and it is considered to conduct an experimental analysis to understand the association between corrected code-smells and new types of code smells in the source code for effective software maintenance and also to prove its quality to retain for a long time and to fix other code-smells implicitly.

8. References

1. Grubb P, Takang AA. Software maintenance: concepts and practice. World Scientific. 2003. Crossref
2. Brown WH, Malveau RC, McCormick HW, Mowbray TJ. AntiPatterns: Refactoring software, architectures, and projects in crisis. John Wiley and Sons. 1998.
3. Fowler M. Refactoring: Improving the design of existing code. Proceedings of 11th European Conference. Jyväskylä, Finland. 1997; p.1–337.
4. Zitzler E, Thiele L. An evolutionary algorithm for multi objective optimization: The strength pareto approach. TIK-report. 1998; p.1–43.
5. Opdyke WF. Refactoring: A program restructuring aid in designing object-oriented application frameworks. [PhD thesis], University of Illinois. 1992.
6. Sahraoui HA, Godin R, Miceli T. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? IEEE Proceedings of ICSM. 2000. p.154–62. Crossref
7. Du Bois B, Demeyer S, Verelst J. Refactoring-improving coupling and cohesion of existing code. IEEE Proceedings of 11th Reverse Engineering, 2004. p.144–51.
8. Moha N, Hacene AM, Valtchev P, Guéhéneuc YG. Refactorings of design defects using relational concept analysis. Proceedings of International Conference on Formal Concept Analysis. Springer Berlin Heidelberg. 2008. p. 289–304. Crossref
9. Joshi P, Joshi RK. Concept analysis for class cohesion. IEEE Proceedings of 13th European Conference on CSMR.. 2009. p. 237–40. Crossref
10. Tahvildari L, Kontogiannis K. A metric-based approach to enhance design quality through meta-pattern transformations. IEEE Proceedings of 7th European Conference on Software Maintenance and Reengineering, 2003; p.183–92. Crossref
11. Soetens QD, Perez J, Demeyer S. An initial investigation into change-based reconstruction of floss-refactorings. IEEE Proceedings of 29th ICSM. 2013. p.384–7. Crossref
12 Kim M, Gee M, Loh A, Rachatasumrit N. Ref-Finder: a refactoring reconstruction tool based on logic query templates. Proceedings of 18th ACM SIGSOFT international symposium on Foundations of software engineering. 2010. p.371–2. Crossref
13. Kataoka Y, Notkin D, Ernst MD, Griswold WG. Automated support for program refactoring using invariants. Proceedings of the IEEE ICSM. 2001 November, p.736. Crossref
14. Zimmermann T, Zeller A, Weissgerber P, Diehl S. Mining version histories to guide software changes. IEEE Transactions on Software Engineering. 2005; 31(6):429–45. Crossref
15. Qayum F, Heckel R. Local search-based refactoring as graph transformation. IEEE Proceedings of 1st International Symposium on. Search Based Software Engineering. 2009. p.43–6. Crossref
16. Mens T, Taentzer G, Runge O. Analysing refactoring dependencies using graph transformation. Software and Systems Modeling. 2007; 6(3):269–85. Crossref
17. Seng O, Stammel J, Burkhart D. Search-based determination of refactorings for improving the class structure of object-oriented systems. Proceedings of 8th annual conference on Genetic and evolutionary computation, ACM. 2006. p.1909–16. Crossref
18. Ouni A, Kessentini M, Bechikh S, Sahraoui H. Prioritizing code-smells correction tasks using chemical reaction optimization. Software Quality Journal. 2015; 23(2):323–61. Crossref
19. Harman M, Tratt L. Pareto optimal search based refactoring at the design level. Proceedings of 9th annual conference on Genetic and evolutionary computation, ACM. 2007. p.1106–13. Crossref
20. Jensen AC, Cheng BH. On the use of genetic programming for automated refactoring and the introduction of design patterns. Proceedings of 12th annual conference on Genetic and evolutionary computation, ACM. 2010. p. 1341–8 Crossref

21. Bansiya J, Davis CG. A hierarchical model for object-oriented design quality assessment. IEEE Transactions on software engineering. 2002; 28(1):4–17. Crossref
22. France R, Bieman J, Cheng BH. Repository for model driven development (ReMoDD). Proceedings of International Conference on Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg. 2006. p. 311–7.
23. Ratzinger J, Sigmund T, Vorburger P, Gall H. Mining software evolution to predict refactoring. IEEE Proceedings of first International Symposium on Empirical Software Engineering and Measurement (ESEM 2007). 2007. p.354–63. Crossref
24. Olbrich SM, Cruzes DS, Sjøberg DI. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. Proceedings of ICSM. 2010. p.1–10. Crossref
25. Deb K, Pratap A, Agarwal S, Meyarivan TA. A fast and elitist multi objective genetic algorithm: NSGA-II. IEEE transactions on evolutionary computation. 2002; 6(2):182–97. Crossref
26. Ouni A, Kessentini M, Sahraoui H. Search-based refactoring using recorded code changes. Proceedings of 17th IEEE European CSMR. 2013. p. 221–30. Crossref
27. Deb K. Multi-objective optimization using evolutionary algorithms. John Wiley and Sons. 2001.
28. Ouni A, Kessentini M, Sahraoui H, Inoue K, Hamdi MS. Improving multi-objective code-smells correction using development history. Journal of Systems and Software. 2015, 105:18–39. Crossref