Amarjeet Prajapati* and Jitender Kumar Chhabra

Optimizing Software Modularity with Minimum Possible Variations

https://doi.org/10.1515/jisys-2018-0231
Received May 18, 2018; previously published online December 4, 2018.

Abstract: Poor design choices at the early stages of software development and unprincipled maintenance practices usually deteriorate software modularity and subsequently increase system complexity. In object-oriented software, improper distribution of classes among packages is a key factor, responsible for modularity degradation. Many optimization techniques to improve the software modularity have been proposed in the literature. The focus of these optimization techniques is to produce modularization solutions by optimizing different design quality criteria. Such modularization solutions are good from the different aspect of quality; however, they require huge modifications in the existing modular structure to realize the suggested solution. Thus these techniques are costly and time consuming if applied at early stages of software maintenance. This paper proposes a search-based optimization technique to improve the modularity of the software system with minimum possible variation between the existing and produced modularization solution. To this contribution, a penalized fitness function, namely, penalized modularization quality, is designed in terms of modularization quality and the Move or Join Effectiveness Measure metric. Furthermore, this fitness function is used in both single-objective genetic algorithm (SGA) and multi-objective genetic algorithm (MGA) to generate the modularization. The effectiveness of the proposed remodularization approach is evaluated over five open-source and three random generated software systems. The experimentation results show that the proposed approach is able to generate modularization solutions with improved quality along with lesser perturbation compared to their non-penalty counterpart and at the same time it performs better with the MGA compared to the SGA. The proposed approach can be very useful, especially when total remodularization is not feasible/desirable due to lack of time or high cost.

Keywords: Modularization; restructuring; optimization; genetic algorithm; refactoring.

1 Introduction

The majority of software systems are designed and developed by decomposing their overall structure into smaller independent units or modules [35]. Such decomposition helps in reducing the system complexity and therefore improves design quality. In an object-oriented (OO) software system, classes play the role of modules which encapsulates the methods and variables. For large and complex OO systems, it has been reported that a package can play the role of a module which groups a set of collaborating classes together to provide well-identified services to the rest of the system [40]. It has been observed that a software system consisting of modules that exhibit low coupling and high cohesion is easier to understand and maintain [21].

It has been found that in regular maintenance of the software system, the maintainers usually do not follow the principles of the module design guidelines, which in turn deteriorates modularity quality [3]. In case of OO software, improper distribution of classes among packages is a key factor, responsible for modularity degradation. The poor modular structure makes the software system difficult to understand and evolve [35]. To improve the modularity of the system, the elements of software need to be reorganized into appropriate modules based on different module design principles. The reorganization of software elements into

***Corresponding author: Amarjeet Prajapati,** Department of Computer Science and Engineering, JIIT, Noida 201309, Uttar Pradesh, India, e-mail: amarjeetnitkkr@gmail.com
Jitender Kumar Chhabra: Department of Computer Engineering, NIT Kurukshetra, Haryana 136119, India

modules based on quality criteria is generally termed as software remodularization. Since software remodularization is considered to be the most crucial NP-hard problem, a large number of search-based optimization techniques have been proposed in the literature to solve the problem (e.g. [16, 17, 26, 29, 31]).

Even after a significant progress made in remodularization, most of the research works focus on improving the modularity quality (e.g. coupling and cohesion) of software as much as possible, without considering the variation between the existing and the produced modular structure. Such approaches can be useful when the software system's quality has deteriorated up to the point where further working with the system is not possible and the system needs complete overhauling. However, in case of early stages of software maintenance, these approaches cannot be feasible because remodularization of the system as a completely new modularization solution compared to the original modular structure is the costly and time-consuming process.

To overcome the aforementioned difficulties and challenges, this paper proposes a search-based optimization technique to improve the modularity of the software system with minimum possible variation between the existing and produced modular structure. To this contribution, a penalized fitness function, namely, penalized modularization quality (PMQ), is designed in terms of modularization quality (MQ) and Move or Join Effectiveness Measure (MoJoFM) metric. Furthermore, this fitness function is used in search-based metaheuristics (single- and multi-objective) to drive the remodularization solution which reflects high modularity quality with minimum variation from the existing modular structure.

Software remodularization with minimum possible perturbation can also be achieved by applying the constraints on the movement of classes among the packages, but this approach may lead towards a suboptimal solution. The main advantage of applying the PMQ fitness function is that it helps in exploring all possible feasible solutions in the search space. Moreover, PMQ helps in guiding the search-based meta-heuristic algorithms towards a good quality solution by minimizing changes in the original modular structure. To confirm this assumption, PMQ is evaluated with the SGA (simple genetic algorithm) and MGA (multi-objective genetic algorithm, i.e. non-dominated sorting genetic algorithm – NSGA-II) proposed by Deb et al. [19]. We chose these search-based metaheuristic algorithms, in particular, because they have been used in related literature [1, 16] to solve similar software remodularization problems. Apart from the genetic algorithm, other search-based metaheuristics can also be used to evaluate the PMQ. However, they will need a huge amount of time for parameter tuning, and if the parameters are not tuned properly, the generated result may be a suboptimal solution. The advantage of using the mentioned genetic algorithm-based optimization technique is that their parameter values have been tuned by the previous researchers. The major contributions of this paper are summarized as follows:

- The paper presents a search-based optimization technique to the problem of improving the modular structure of an existing OO software package organization with regard to minimum possible perturbation.
- To guide the search algorithms towards a solution with improved quality with minimum possible modification, a novel fitness function, namely PMQ, has been proposed.
- To confirm the supremacy of the PMQ fitness function, it has been introduced and optimized with the SGA and MGA to address the remodularization problem.
- An empirical study is conducted to evaluate the effectiveness of the proposed method over five real-world and three random systems. The primary finding of the study are as follows: (1) the proposed approach with penalized optimization is able to achieve the similar level of MQ with minimum perturbation with global optimization in both single- and multi-objective GA; (2) the MGA performs better for both penalized and global optimization than SGA.

The rest of the paper is organized as follows: Section 2 presents the related work material on software remodularization. Section 3 provides the description of the problem. Section 4 describes the proposed methodology. Section 5 presents the experimentation details. Section 6 presents the finding and analysis. Section 7 concludes the paper.

2 Related Work

In the context of various programming languages such as C, COBOL and Pascal, the software remodularization research field is relatively old. However, it is still really important and requires innovative approaches to deal with the complexity of modern systems especially those developed in OO programming languages [12]. A lot of works in the context of OO software have been proposed for modularizing the classes into module/packages in order to improve the software design. The studies [11, 13, 18, 22, 27, 32, 33] impart that software remodularization is an important and challenging problem in the field of software engineering.

In the last two decade, many remodularization approaches have been proposed by researchers and academicians working in the software engineering field. Wiggerts [37] was the first who established a theoretical concept regarding software remodularization. They presented a software remodularization problem as a software clustering problem which can be solved by using clustering techniques and cluster evaluation criteria. They discussed various similarity criteria of the software entities useful in clustering evaluation and provided a summary of applicable clustering algorithms of clustering techniques. Later many deterministic and intelligence-based approaches to solve the remodularization problem as a clustering problem was proposed in the literature (e.g. [2, 4, 6, 10, 11, 25, 28, 29]).

The development of various intelligence techniques specially designed to solve the different science and engineering problem opens a new avenue for the clustering problem [5, 6, 8, 9, 15, 23, 25, 29, 30, 34, 38, 39]. Although remodularization was done over the procedural systems, many conclusions may be applied to OO systems as well. Mancoridis et al. [25] introduced a search-based clustering technique to create a high-level view of software organization. Anquetil and Lethbridge [14] conducted an intensive study on the application of clustering techniques for software remodularization. Their empirical study includes a comparison between different clustering algorithms, different representation schemes and different coupling metrics between files.

Later, Mitchell and Mancoridis [29] used the same clustering techniques and developed a tool Bunch, which support the automatic software module clustering. Abdeen et al. [2] proposed a single-objective optimization approach for reducing the dependences between the packages of existing software organization. Recently, Praditwong et al. [31] formulated the software clustering problem as a search-based multi-objective optimization problem. They use the genetic-based two-archive multi-objective evolutionary algorithm.

The above-mentioned approaches utilize structural, dynamic, semantic and conceptual information to design various quality measures for suggesting the software remodularization solution. Most of the remodularization approaches utilize the structural information to derive the quality measure [26, 31]. Bavota et al. [17] used the structural information and proposed an interactive multi-objective optimization approach for software remodularization. Barros [16] performed an empirical study to analyze the effect of composite objectives in multi-objective software modularization.

Most of the existing approaches performed software remodularization from scratch rather than improving the existing software modular structure. In literature, few research works addressed the problem of software remodularization within existing software decomposition [4, 28]. Recently Abdeen et al. [1] proposed a single-objective software remodularization approach based on the simulated annealing (SA) technique. Their approach aimed to reduce the package coupling and improve the package cohesion by moving the classes into the existing packages. However, as a single-objective remodularization approach, SA can optimize some objective on the cost of another objective. To address these limitations, the same authors [3] proposed a multi-objective optimization for software remodularization on the existing package organization. Although the approach is promising and effective, they have used the limited aspects of relationships contributing to the coupling between software elements. Inspired by the software remodularization approach by Abdeen et al. [3], we propose search-based remodularization for improving the existing package organization. The proposed approach ensures that the optimization is carried out in a way that the existing package organization gets altered to the minimum possible extent. The advantage of such methodology is that cost of remodularization remains low.

3 Problem Description

During the maintenance of large and complex software systems, the quality of the original program design degrades. To improve the quality of the existing program design, the software systems are often repaired using the remodularization approach. The automatic remodularization approach which is based on clustering techniques generally suggests a totally new modularization solution compared to the original package organization. The implementation of such a modularization solution is costly and difficult to understand. Hence, to minimize the cost, the perturbation made over the original modular structure with regard to quality improvement needs to be controlled.

In order to remodularize the software system, the maintainers require a wide range of structural information of the software elements for designing the different quality criteria. The formal information such as the relationship with their strength among classes is widely used. Formally, the definition of our remodularization problem is to improve the specified modularity quality criteria of an existing package organization of an OO software system with regard to minimum possible perturbation. For the remodularization problem, package structure, class relations and class coupling strength are constrained by the following assumptions:

- The package of an OO system is referred to as module and classes are referred to as software entities.
- The module is defined as a cohesive group of classes, meaning that all classes within a package have strong coupling strength.
- A very important constraint to consider is that any class in a software system must be contained in one and only one package in the resulting modularization solution.
- During remodularization, the number of packages does not increase or decrease.
- The classes within the system can be connected by a structural relationship.
- The relationships can be weighted or unweighted.

4 Software Remodularization Approach

In this work, a software remodularization approach aiming to improve the quality of the existing package structure of the OO system has been designed and developed. In particular, the main goal of the proposed approach is to improve modularity quality, particularly the MQ metric of the existing package structure with minimal possible modification in the original package organization. To achieve the goal, the software remodularization problem is formulated as a search-based single- and multi-objective optimization problem where software modularity is optimized along with minimum possible perturbation in the existing modular structure with the help of a genetic algorithm. Specifically, the optimization process of the genetic algorithm is controlled by incorporating a penalized fitness function, namely PMQ.

In single-objective optimization PMQ is maximized, and in multi-objective optimization the same PMQ is optimized with other supporting quality criteria such as follows: (1) maximize MQ; (2) minimize package coupling; (3) maximize package cohesion. The supporting objective functions are used to just guide the optimization process towards a better primary objective (PMQ).

The general structure of software remodularization is illustrated in Figure 1. It takes as an input the original package organization of the OO software system and penalized objective criteria. The remodularization process generates an output of the remodularization suggestion needed to be applied to the software system in order to improve the system quality. In the following subsection, the detailed descriptions are given.

4.1 System Structure Representation

In order to formulate the software remodularization problem as a search-based optimization problem, the software system needs to be represented in a way such that various operators can be applied. In this paper, we represent the system with weighted graph (G_w) and unweighted graph (G_u). The weighted graph G_w is defined as a 3-tuple $G_w = (V, E, W)$, where $V = \{v_0, v_1,...,v_n\}$ is the set of vertices where each vertex represents a class, $E = \{e(v_i, v_j)\} \subset V \times V$ is the set of directed edges and each edge represents a connection

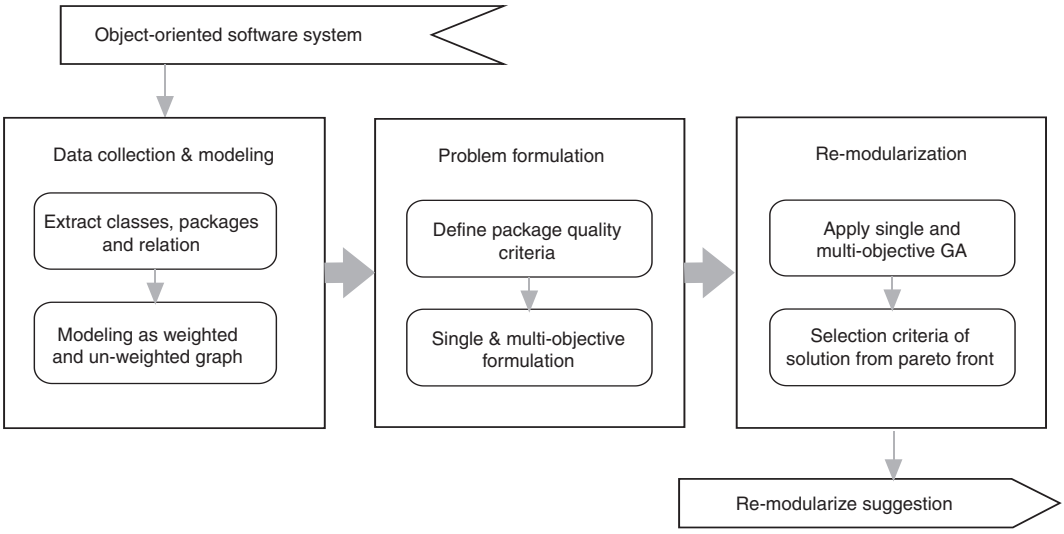

Figure 1: An Approach for Software Remodularization.

between two classes, and W is the set of weights for each edge and it can be any real value depending on connections among the respective classes. The unweighted graph G_u is also defined as a 2-tuple $G_w = (V, E)$, where V and E have the same meaning as in the weighted graph. The presence of an edge shows that there exists at least one connection among the two classes.

In an OO software system, a class can be linked with another class by zero or more relations with different types. Such links are called connection. The connection weight is computed by considering the three aspects: (1) types of relations; (2) number of instances of relations; (3) weights of each type of relations. To calculate the connection weight, the eight well-known relationships [i.e. extends (EX), Has Parameter (HP), Reference (RE), Calls (CA), Implement (IM), Is of Type (IT), Return (RE), and Throws (TH)] as discussed in Amarjeet and Chhabra [6, 8, 9] have been considered in this paper. The connection weight CW_{ij} between classes c_i and class c_j is defined as follows:

$$\mathrm{CW}_{ij} = \begin{cases} \text{Undefined} & \text{if } i = j \\ \sum_{k=\mathrm{EX}}^{\mathrm{TH}} w_k n_k(c_i, c_j) & \text{otherwise} \end{cases} \tag{1}$$

where $n_k(c_i, c_j)$ denote the total number of instances of the k-type relation between classes c_i and c_j; and w_k represents the weight of the class k-type relation. The weight of each relation in this paper is considered to be equal to 1. For example, Figure 2 illustrates two calls and one reference relation between class C_1 and class C_2. Hence, according to the definition the connection weight is 3 in the weighted graph while the connection weight is 1 in the unweighted graph.

To represent the weighted and unweighted graph as a chromosome, a simple array is used, where the ith element indicates the package to which the ith class is assigned. A modularization solution with the same value for all elements means that all classes are placed in the same package. The modularization solution representation of the hypothetical OO software system given in Figure 2 can be represented as {1, 3, 3, 1, 1, 2, 2, 2}. For example, classes C_0, C_3 and C_4 are in the same package (i.e. package 1). The same representation for the chromosome is used in the SGA and MGA.

4.2 Penalized Fitness Function

To guide the optimization process of the genetic algorithm towards the improved modularization solution exhibiting minimum variation from the existing modular structure, an adequate fitness function is required.

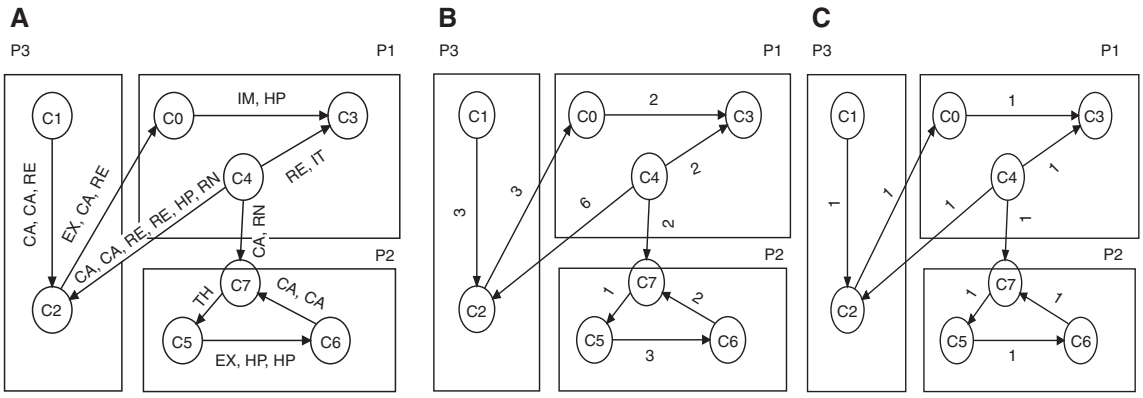

Figure 2: Weighted and Unweighted Version of a Hypothetical Object-Oriented Software System.

In this paper, we define a novel fitness function, namely PMQ, where the modularity quality criterion, i.e. MQ metric, is penalized with the MoJoFM metric. The MQ and PMQ metrics are defined as follows:

– **Modularization quality (MQ):** The MQ metric is designed to evaluate the modularly of a software system. It is formulated as the sum of modularization factors (MFs) and MF is measured in terms of the inter-package coupling and intra-package coupling.

$$\mathrm{MQ} = \sum_{k=1}^{n} \mathrm{MF}_k \quad \text{where} \quad \mathrm{MF}_k = \begin{cases} 0 & \text{if } i = 0 \\ \dfrac{i}{i + \frac{1}{2}j} & \text{if } i > 0 \end{cases} \tag{2}$$

where i is the intra-package coupling and j is the inter-package coupling and n is the total number of packages. MQ shows a tradeoff between coupling and cohesion. The other metric such as basic MQ [25] can also be used to evaluate the modularity. The major disadvantage of basic MQ is that it cannot be used to measure the quality of modularization solution obtained from graphs having weighted edges.

– *PMQ:* In PMQ, we redefine the MQ metric by multiplying perturbation degree (PD) as a penalty. The PD is defined in terms of the move and join operation of remodularization and it is derived from the MoJoFM metric [36].

$$\mathrm{PMQ} = \mathrm{MQ} \times \mathrm{PD} \quad \text{where} \quad \mathrm{PD} = 1 - \frac{\mathrm{mno}(M_{\mathrm{new}}, M_{\mathrm{org}})}{\max(\mathrm{mno}(\forall M_{\mathrm{new}}, M_{\mathrm{org}}))} \tag{3}$$

where $\mathrm{mno}(M_{\mathrm{new}}, M_{\mathrm{org}})$ is the minimum number of Move or Join operations to transform the modularization solution M_{new} into the modularization solution M_{org} and $\max(\mathrm{mno}(M_{\mathrm{new}}, M_{\mathrm{org}}))$ is the maximum possible distance of any new modularization solution M_{new} from the original modularization solution M_{org}. The purpose of the MoJoFM metric is to compute the similarity between two modularization solutions. There are other similarity metrics such as architecture-to-architecture (a2a) [24] and cluster-to-cluster coverage ($\mathrm{c2c}_{\mathrm{cvg}}$) [20] that exist in the literature. However, the MoJoFM metric is more appropriate than a2a and $\mathrm{c2c}_{\mathrm{cvg}}$ in this context. If the software remodularization solutions being compared consist of the same classes (as in our case), a2a and $\mathrm{c2c}_{\mathrm{cvg}}$ will give results with a small range of variation, which makes it difficult to differentiate the remodularization solutions. Hence, in our cases, the MoJoFM metric is more appropriate than a2a and $\mathrm{c2c}_{\mathrm{cvg}}$.

The rationale of introducing PMQ is to penalize the improvement of MQ, to keep the minimum possible restructuring cost. The smaller the PD value, the smaller is the number of movements of classes among the existing packages. However, our objective is not to minimize the PD in an absolute way, but to ensure that the achieved improvement of the package structure is made at the cost of minimum possible class movements. Apart from MQ and PMQ quality criteria, we also consider the other conflicting criteria such as coupling,

cohesion and number of isolated packages. Based on these quality criteria, we formulate the remodularization problem as a single- and multi-objective optimization problem which is described in the following subsections.

4.2.1 Single-Objective Remodularization

In a single-objective software optimization problem, only the single objective is optimized. It determines a modularization M^* for which

$$F(M^*) = \min/\max F(M) \mid M \in \psi \tag{4}$$

where ψ is the set of all feasible modularizations. M is the software remodularization solution such as F: $\psi \longrightarrow R$ is an objective function. Here function F can be a minimization function or maximization function. Most of the software modularization problems are based on the single-objective optimization problem. Different single-objective optimization approaches vary with the optimization function F and optimization method. In the previous remodularization approaches the MQ has been widely used as design quality criteria [26, 31]. In this paper, we optimize the PMQ metric as a fitness function and use the single-objective GA.

4.2.2 Multi-objective Remodularization

In multi-objective software optimization, more than one objective is optimized. It determines a set of modularizations M^* for which

$$F(M^*) = \min/\max(F_1(M), F_2(M), \ldots, F_m(M)) \mid M \in \psi \tag{5}$$

where ψ is the set of all feasible modularizations and m is the number of objective functions. F_i represents the ith objective function. In multi-objective software optimization, there is usually no single best solution, but there can be more than one non-dominated modularization solution. For two modularization solutions $M_1, M_2 \in \psi$, solution M_1 is said to dominate solution M_2 (denoted as $M_1 \leq M_2$) if and only if

$$\forall\, i \in (1, \ldots, m)\, F_i(M_1) \leq F_i(M_2) \wedge \exists\, i \in (1, \ldots, m)\, F_i(M_1) < F_i(M_2) \tag{6}$$

Otherwise, M_1 and M_2 are said to be non-dominated solutions. The set of all non-dominated solutions in objective space is called Pareto front. The multi-objective modularization techniques provide flexible modularization solutions where the developer has more options for selection of the best solution based on his or her requirements.

The reason for the use of multi-objective optimization is to improve the single-objective function PMQ with the help of other supporting objective functions. Motivation is similar to one of Praditwong et al. [31], which demonstrates that the MQ value of the software system improves more as multi-objective optimization with the support of other conflicting objective functions such as coupling and cohesion, as compared to improvement through single-objective optimization. We consider the PMQ metric as a primary objective and cohesion, coupling, and number of isolated packages as supporting objectives. The goal of the proposed multi-objective optimization approach is to maximize the PMQ and package cohesion and minimize the package coupling and number of isolated packages.

5 Experimental Setup

This section explains the experimental setup conducted to assess the proposed approach. The whole experimentation is divided into three major parts and it is done under the scenario of the SGA vs. MGA and unweighted vs. weighted system model: (1) assessment of global optimization, where no constraints or penalty is applied; (2) assessment of penalized optimization, where penalty is incorporated to limit the perturbation; (3) comparison of penalized optimization and global optimization.

5.1 Software Systems Studied

The experiment studies the application of the proposed approach to five different real-world open source OO software systems based on Java language and three random systems. The real-world systems include JavaCC, JUnit, Java Servlet API, XML API DOM, DOM4J and the random systems include Random50, Random100 and Random 100. The software systems are modeled into two types. The first type is a weighted model where the edge weight is assigned according to the method discussed in the previous section. The second type is an unweighted model where edge weight is assigned a binary value. The details about the selected problem instances are given in Table 1.

We choose these OO software systems for our assessment since they range from a medium to a large number of classes and packages and have a different level of complexity. The different sizes and complexities of a software system can provide a clear insight into the modularization techniques. It also helps to mitigate the biasing of the results. These systems have also been used in similar problems by other previous researchers to evaluate their methods for the remodularization problem.

5.2 Algorithmic Parameters

In this paper, the SGA is used for single-objective optimization and the NSGA-II for multi-objective optimization. The NSGA-II is a meta-heuristic genetic algorithm that is based on the non-domination sorting concepts of the multi-objective optimization technique. It generates a set of non-dominated solutions that is known as the Pareto set. This paper uses the same parameter configuration for these GAs as also used in the literature [7, 16, 25, 31]. The parameter values are as follows: (1) population size is 10 times the number of classes (N), (2) single-point crossover operator and uniform mutation operator, (3) crossover probability is set to 0.8, while the mutation probability to 0.004 $\log_2(N)$, (4) the maximum number of generations is 200 times the number of classes (N).

5.3 Collecting Results from Experiment

In the proposed remodularization approach, the main goal is to improve the MQ value of the existing package organization with the minimum possible movements of classes among packages. Hence, we are only interested in the modularization with the highest PMQ value, although they might not be one with the highest values for other objectives. The motivation is similar to the one of Praditwong et al. [31], which used MQ to select the best solution in the Pareto fronts of the multi-objective evolutionary algorithm. Each SGA and MGA are executed 31 times on each of the real-world and random systems. As the SGA generates only one solution with the highest PMQ value at each execution. As for the MGA, we again select the modularization with the highest PMQ value at each execution.

Table 1: Characteristics of Software Systems.

Systems	Version	Abbreviation	No. of connections	No. of packages	No. of classes
Real-world problem					
JavaCC	1.5	JC	722	6	154
JUnit	3.81	JU	276	6	100
Java Servlet API	2.3	JS	131	4	63
XML API DOM	1.0.b2	XA	209	9	119
DOM 4J	1.5.2	DJ	930	16	195
Random problems					
Random50	NA	R1	218	7	50
Random100	NA	R2	342	12	100
Random150	NA	R3	534	17	150

5.4 Results Assessment Criteria

To assess the solutions obtained by the proposed approach, we use the MQ measure to evaluate the quality of modularization and rate per refactoring of achieved improvement (RRAI) measure proposed by Abdeen et al. [3] to measure the perturbation. The RRAI with respect to MQ measurement is defined as follows:

$$\mathrm{RRAI(MQ)} = \frac{\mathrm{RPMC(MQ)}}{\mathrm{RPC(MQ)}} \tag{7}$$

where RPC(MQ) represents the rate per class of MQ measurement and it is computed as follows: RPC(MQ) = MQor/|C|, where MQor is the value of MQ of the original software package structure and C is the set of all classes. The RPMC(MQ) represents the rate per moved class of MQ measurement and it is defined as follows:

$$\mathrm{RPMC(MQ)} = \frac{\Delta \mathrm{MQ}}{\mathrm{NC}} \tag{8}$$

where ΔMQ is the increased value of MQ in new modularization and NC is the number of classes that change their packages in new modularization. The larger the value of RRAI(MQ), the smaller the modification with respect to the MQ measurement. Ideally, the RRAI(MQ) value should always be greater than 1 (i.e. RPMC(MQ) > RPC(MQ).

6 Results and Analysis

This section presents the results of the empirical study. The results concern two optimizations, global and penalized optimization, two genetic algorithms, single- and multi-objective GA, and two system models, weighted and unweighted graph. The usefulness and effectiveness of the suggested modularization solution of the proposed penalized optimization are assessed through MQ quality measures and RRAI measure [3] and further compared with global optimization.

Since the metaheuristic algorithms are a stochastic optimizer, a pairwise statistical analysis using the Wilcoxon test ($\alpha = 0.05$) is performed to compare the results of two metaheuristic approaches. The main reason behind using the Wilcoxon test is that it is more effective for the non-normal distribution while the other alternate test such as the *t*-test is more appropriate in case of the normal distribution.

6.1 Modularization Quality

This section presents the results of the experiments that compare the MQ values obtained from both global and penalized optimization in all scenarios discussed in Section 5. Table 2 presents the results of the mean, median and standard deviation of the MQ values produced by the SGA and MGA in global optimization over unweighted and weighted software systems. Similarly, Table 4 presents the results obtained by the proposed penalized optimization. Figure 3 shows the comparison of mean MQ values results between the global and penalized optimization.

6.1.1 Single-Objective vs. Multi-Objective GA

In this part, we analyze the MQ values produced by the SGA and MGA in both global and penalized optimization over both unweighted and weighted software systems. The detailed analysis is given as follows: (1) global optimization and unweighted system: the results presented in Table 2 show that the MGA performs better than the SGA in six cases out of eight cases in which two cases are significantly better; (2) global optimization and weighted system: Table 2 shows that the MGA performs better than the SGA in all problem instances in which six cases are significantly better; (3) penalized optimization and unweighted system: Table 3 shows

Table 2: Mean, Median and Standard Deviation of MQ Measure Obtained by Global Optimization.

Problems	MGA			SGA			MGA vs. SGA		
	Mean	Median	Standard deviation	Mean	Median	Standard deviation	Δ	*p*-Value	Winner
Unweighted									
JavaCC-1.5	4.007	4.132	0.015	3.854	4.051	0.062	+0.081	0.862	≈
Junit-3.8.1	4.298	4.325	0.013	4.224	3.621	0.027	+0.704	0.012	⟵
Servlet API-2.3	3.547	3.621	0.023	3.407	3.687	0.046	−0.066	0.394	≈
XML API-1.0.b2	8.249	8.327	0.063	8.160	7.214	0.055	+1.113	0.001	⟵
DOM 4J-1.5.2	6.557	6.471	0.049	6.752	6.453	0.042	+0.018	0.256	≈
Random 50	4.077	5.018	0.075	4.019	4.034	0.089	+0.984	0.112	≈
Random 100	6.230	6.501	0.053	5.997	5.674	0.081	+0.927	0.135	≈
Random 150	7.358	7.234	0.081	7.138	8.443	0.039	−1.209	0.002	⟶
Weighted									
JavaCC-1.5	6.607	6.213	0.045	6.096	6.151	0.086	+0.062	0.024	≈
Junit-3.8.1	6.455	6.316	0.072	6.061	5.281	0.071	+1.035	0.006	⟵
Servlet API-2.3	6.574	6.415	0.047	5.113	5.211	0.054	+1.204	0.015	⟵
XML API-1.0.b2	8.721	8.101	0.037	7.123	8.015	0.046	+0.086	0.637	≈
DOM 4J-1.5.2	9.754	10.031	0.079	8.170	8.042	0.081	+1.989	0.001	⟵
Random 50	6.356	6.581	0.098	5.901	5.312	0.069	+1.269	0.006	⟵
Random 100	8.316	8.531	0.065	7.895	7.154	0.097	+1.377	0.008	⟵
Random 150	9.244	9.643	0.104	9.020	8.179	0.071	+1.464	0.017	⟵

The symbol ' ⟶ ' denotes the cases where the MGA exhibited the superior performance in the pairwise Wilcoxon test at 95% significance level ($\alpha = 0.05$); the symbol ' ⟵ ' indicates the cases where the SGA exhibited the superior performance; and the symbol '≈' indicates cases in which there is no statistical difference between the MGA and SGA. The delta values Δ denote the difference between the median values MGA and SGA.

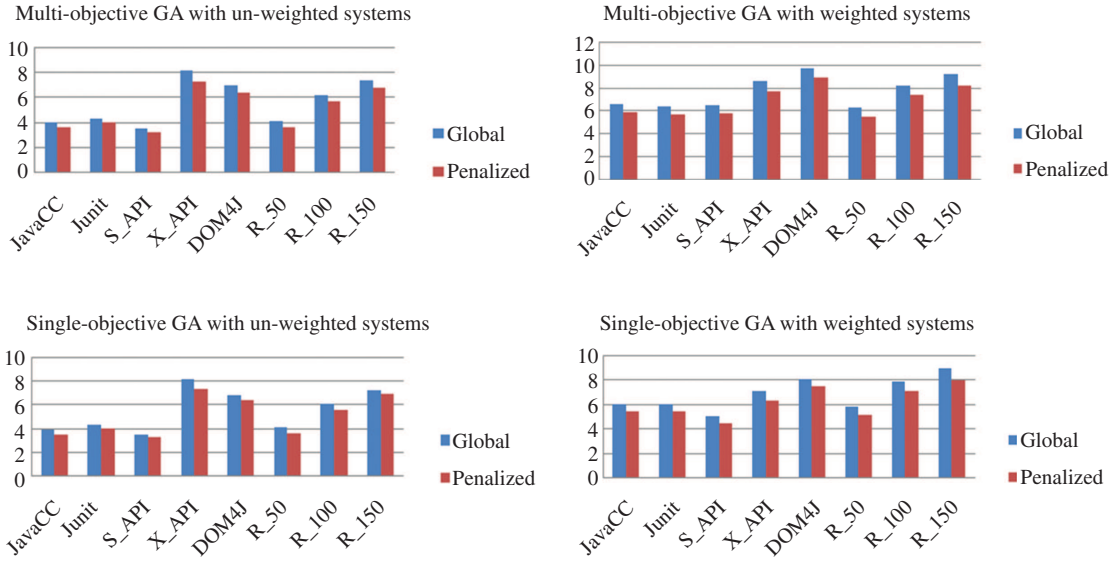

Figure 3: Analysis of MQ Values in Penalized Optimization from Global Optimization.

that the MGA performs better than the SGA in six cases out of eight in which one case is significantly better; (4) penalized optimization and weighted system: Table 3 shows that the MGA performs better than the SGA in all problem instances in which five cases are significantly better.

Table 3: Mean, Median and Standard Deviation of MQ Measure Obtained by Penalized Optimization.

Problems	MGA			SGA			MGA vs. SGA		
	Mean	Median	Standard deviation	Mean	Median	Standard deviation	Δ	*p*-Value	Winner
Unweighted									
JavaCC-1.5	3.563	3.621	0.012	3.458	3.566	0.062	+0.055	0.759	≈
Junit-3.8.1	3.975	3.923	0.023	3.917	3.865	0.027	+0.058	0.127	≈
Servlet API-2.3	3.186	3.045	0.028	3.276	3.187	0.046	−0.142	0.035	⟶
XML API-1.0.b2	7.274	7.387	0.067	7.243	7.314	0.055	+0.073	0.654	≈
DOM 4J-1.5.2	6.356	6.143	0.052	6.367	6.652	0.042	−0.509	0.042	⟶
Random 50	3.573	3.498	0.071	3.551	3.434	0.089	+0.064	0.132	≈
Random 100	5.716	5.824	0.047	5.564	5.176	0.081	+0.648	0.025	⟵
Random 150	6.819	6.756	0.074	6.896	6.745	0.039	+0.011	0.212	≈
Weighted									
JavaCC-1.5	5.983	5.812	0.052	5.536	5.645	0.086	+0.167	0.082	≈
Junit-3.8.1	5.764	5.713	0.063	5.453	5.689	0.071	+0.024	0.126	≈
Servlet API-2.3	5.812	5.756	0.058	4.532	4.437	0.054	+1.319	0.015	⟵
XML API-1.0.b2	7.763	7.834	0.029	6.364	6.431	0.046	+1.403	0.002	⟵
DOM 4J-1.5.2	8.967	8.823	0.087	7.549	7.672	0.081	+1.151	0.015	⟵
Random 50	5.564	5.678	0.124	5.196	5.675	0.069	+0.003	0.126	≈
Random 100	7.462	8.521	0.085	7.127	7.254	0.097	+1.267	0.008	⟵
Random 150	8.231	8.376	0.074	8.082	7.679	0.071	+0.697	0.017	⟵

The symbol ' ⟶ ' denotes the cases where the MGA exhibited the superior performance in the pairwise Wilcoxon test at the 95% significance level ($\alpha = 0.05$); the symbol ' ⟵ ' indicates the cases where the SGA exhibited the superior performance; and the symbol '≈' indicates cases in which there is no statistical difference between the MGA and SGA. The delta values Δ denote the difference between the median values MGA and SGA.

6.1.2 Global vs. Penalized Optimization

Figure 3 shows the percentage loss in MQ values in penalized optimization compared to global optimization. The results clearly indicate that there is very small percentage loss in MQ values of penalized optimization than global optimization in both single- and multi-objective optimization algorithm.

6.2 Achieved Optimization vs. Applied Modification

This section presents the results of experiments that compare the degree of modification obtained from both global and penalized optimization in all scenarios discussed in Section 5. Table 4 presents the results of a number of moved classes and RRAI values produced by the SGA and MGA in global optimization over unweighted and weighted software systems. Similarly, Table 5 presents the results produced by the proposed penalized optimization. Figure 4 shows the comparison of percentage movement of classes for the global and penalized optimization.

6.2.1 Single-Objective vs. Multi-Objective GA

We compare the experimental results produced by the SGA and MGA in terms of the number of moved classes and RRAI values. The comparison is performed in the following scenario: (1) global optimization and unweighted system: the results for a number of moved classes presented in Table 4 show that the MGA performs significantly better than the SGA in all problem instances except Junit and Random150. Similar to the number of moved classes, the RRAI values, the MGA performs significantly better than the SGA for all problem instances except Junit and Random100; (2) global optimization and weighted system: in this scenario, the results for a number of moved classes given in Table 4 show that the MGA performs significantly better than the SGA in all problem instances except Junit and Random150. Similar to the number of moved

Table 4: Mean Values of Moved Classes and RRAI in Global Optimization.

Problem	Moved classes					RRAI				
	MGA	SGA	Δ	*p*-Value	Winner	MGA	SGA	Δ	*p*-Value	Winner
Unweighted										
JavaCC-1.5	40.03	47.72	−3.73	0.012	⟶	0.001	0.001	+0.0003	0.015	⟵
Junit-3.8.1	27.44	18.27	+5.34	0.091	≈	0.001	0.002	−0.0012	0.023	≈
Servlet API-2.3	18.37	21.00	−2.13	0.014	⟶	0.002	0.002	+0.0007	0.017	⟵
XML API-1.0.b2	35.10	37.16	−1.61	0.017	⟶	0.004	0.003	+0.0015	0.024	⟵
DOM 4J-1.5.2	47.55	56.52	−5.43	0.001	⟶	0.001	0.001	+0.0002	0.018	⟵
Random 50	21.24	21.87	−0.91	0.021	⟶	0.004	0.003	+0.0014	0.035	⟵
Random 100	21.96	28.86	−4.31	0.013	⟶	0.003	0.002	+0.0012	0.001	⟵
Random 150	46.32	39.10	+7.86	0.085	≈	0.002	0.002	−0.0002	0.085	≈
Weighted										
JavaCC-1.5	21.76	25.52	−2.16	0.012	⟶	0.003	0.002	+0.0016	0.022	⟵
Junit-3.8.1	19.93	22.72	−1.14	0.034	⟶	0.005	0.004	+0.0018	0.014	⟵
Servlet API-2.3	15.91	20.40	−2.13	0.014	⟶	0.011	0.004	+0.0342	0.036	⟵
XML API-1.0.b2	24.28	26.28	−1.19	0.018	⟶	0.006	0.004	+0.0023	0.003	⟵
DOM 4J-1.5.2	45.19	47.14	−1.98	0.011	⟶	0.002	0.001	+0.0001	0.007	⟵
Random 50	11.54	14.17	−2.43	0.023	⟶	0.017	0.011	+0.0041	0.008	⟵
Random 100	20.85	22.08	−1.67	0.017	⟶	0.007	0.006	+0.0001	0.014	⟵
Random 150	27.30	31.28	−2.13	0.002	⟶	0.005	0.004	+0.0001	0.012	⟵

The symbol ' ⟶ ' denotes the cases where the MGA exhibited the superior performance in the pairwise Wilcoxon test at the 95% significance level ($\alpha = 0.05$); the symbol ' ⟵ ' indicates the cases where the SGA exhibited the superior performance; and the symbol '≈' indicates cases in which there is no statistical difference between the MGA and SGA. The delta values Δ denote the difference between the median values MGA and SGA.

Table 5: Mean Values of Moved Classes and RRAI in Penalized Optimization.

Problem	Moved classes					RRAI				
	MGA	SGA	Δ	*p*-Value	Winner	MGA	SGA	Δ	*p*-Value	Winner
Unweighted										
JavaCC-1.5	27.56	31.58	−3.84	0.012	⟶	3.152	2.526	+0.83	0.002	⟵
Junit-3.8.1	16.21	22.34	−4.12	0.032	⟶	1.935	1.597	+0.86	0.003	⟵
Servlet API-2.3	13.23	14.67	−1.24	0.013	⟶	1.365	1.234	+0.13	0.015	⟵
XML API-1.0.b2	22.45	24.34	−1.67	0.029	⟶	1.511	1.454	+0.15	0.017	⟵
DOM 4J-1.5.2	28.78	33.87	−4.15	0.003	⟶	1.214	1.927	+0.93	0.008	⟵
Random 50	12.61	13.54	−0.91	0.028	⟶	1.319	1.198	+0.14	0.024	⟵
Random 100	14.87	18.33	−3.26	0.017	⟶	1.501	1.440	+0.56	0.035	⟵
Random 150	29.25	31.39	−1.86	0.008	⟶	1.398	1.221	+0.92	0.028	⟵
Weighted										
JavaCC-1.5	15.82	17.45	−1.23	0.005	⟶	4.447	3.257	+1.19	0.005	⟵
Junit-3.8.1	12.27	13.87	−1.16	0.016	⟶	3.737	2.738	+1.21	0.008	⟵
Servlet API-2.3	9.12	11.39	−2.07	0.031	⟶	3.993	1.275	+2.86	0.004	⟵
XML API-1.0.b2	14.81	16.67	−1.36	0.001	⟶	8.183	4.972	+3.01	0.012	⟵
DOM 4J-1.5.2	25.23	27.34	−2.08	0.003	⟶	3.230	1.382	+2.12	0.015	⟵
Random 50	7.34	8.48	−1.11	0.024	⟶	3.255	2.241	+1.24	0.018	⟵
Random 100	13.56	15.35	−1.29	0.005	⟶	2.569	1.875	+1.01	0.006	⟵
Random 150	16.23	18.26	−1.54	0.018	⟶	2.785	2.282	+1.11	0.007	⟵

The symbol ' ⟶ ' denotes the cases where the MGA exhibited the superior performance in the pairwise Wilcoxon test at the 95% significance level ($\alpha = 0.05$); the symbol ' ⟵ ' indicates the cases where the SGA exhibited the superior performance; and the symbol '≈' indicates cases in which there is no statistical difference between the MGA and SGA. The delta values Δ denote the difference between the median values MGA and SGA.

classes, the results of RRAI values also show that the MGA performs significantly better than the SGA in all problem instances except Junit and Random150; (3) penalized optimization and unweighted system: Table 5 shows that in all problem instances the number of moved classes with the MGA is significantly smaller than

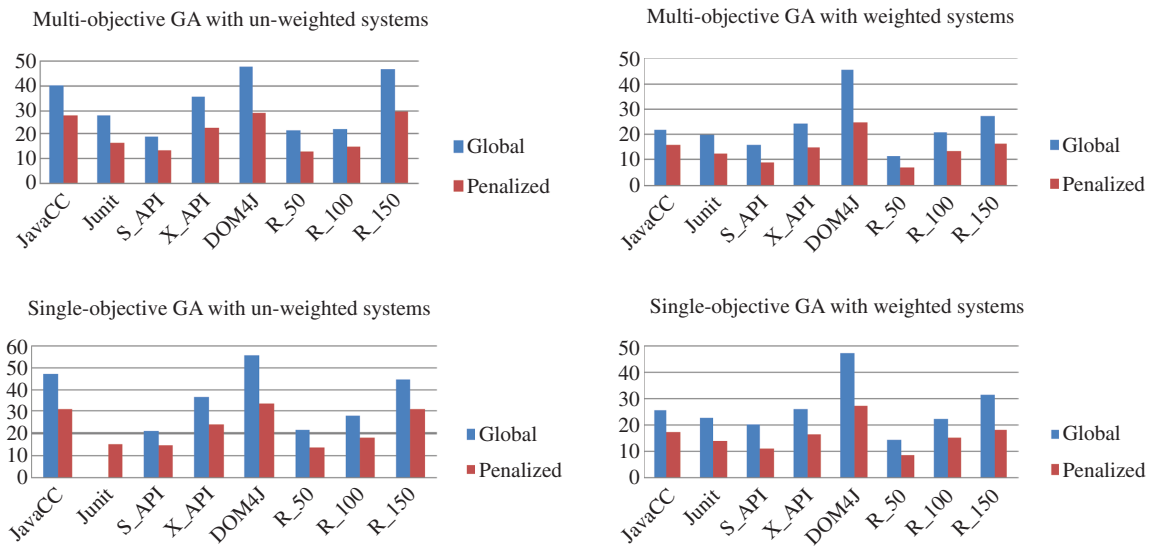

Figure 4: Percentage of Class Movement in Global and Penalized Optimization.

that with the single-objective GA. Similar to the number of moved classes, the RRAI values, for all problem instances the MGA also performs better than the single-objective GA. In this scenario the mean of the RRAI values in both SGA and MGA is larger than the baseline value (which is 1); (4) penalized optimization and weighted system: Table 5 shows that in all problem instances the number of moved classes with the MGA is significantly smaller than that with the single-objective GA. Similar to the number of moved classes, the RRAI values, for all problem instances the MGA also performs better than the single-objective GA. In this scenario also the mean of RRAI values in both SGA and MGA is larger than the baseline value (which is 1).

6.2.2 Global vs. Penalized Optimization

Now we compare the percentage decrement in the number of class movements in both global and penalized optimization. Figure 4 sows the comparison of both optimizations. The data clearly indicate that there is very large percentage decrement in class movement of penalized optimization than global optimization in both single- and multi-objective optimization algorithm.

6.3 Compromised Quality vs. Reduced Perturbation

Figure 5 shows the percentage reduction in the number of moved classes and MQ values in penalized optimization over global optimization. The *x*-axis represents the problem instances and the *y*-axis represents the percentage reduction.

Figure 5 clearly indicates that a relatively large percentage of moved classes among the packages can be reduced in both SGA and MGA compared to MQ. For example in the multi-objective and weighted case, penalized optimization reduced moved classes on average by approximately 37% for all problem instances at the cost of a reduction in MQ values just on average by approximately 10%. Hence results proved that the proposed penalized optimization technique is able to reduce the significant movement of classes among the existing package organization without much compromising in the MQ values.

The overall experimentation results provide significant evidence that the presented penalized optimization approach for software remodularization is able to improve the modularization quality of the existing package organization by doing minimum possible perturbation. The empirical results also show that multi-objective formulation, MGA, outperforms single-objective formulation, SGA, in mostly all scenarios except some cases. The reason for outperforming the MGA is that MGA formulation is more capable of exploring all possible modularization search space compared to the SGA. In the SGA only a single aspect of quality is

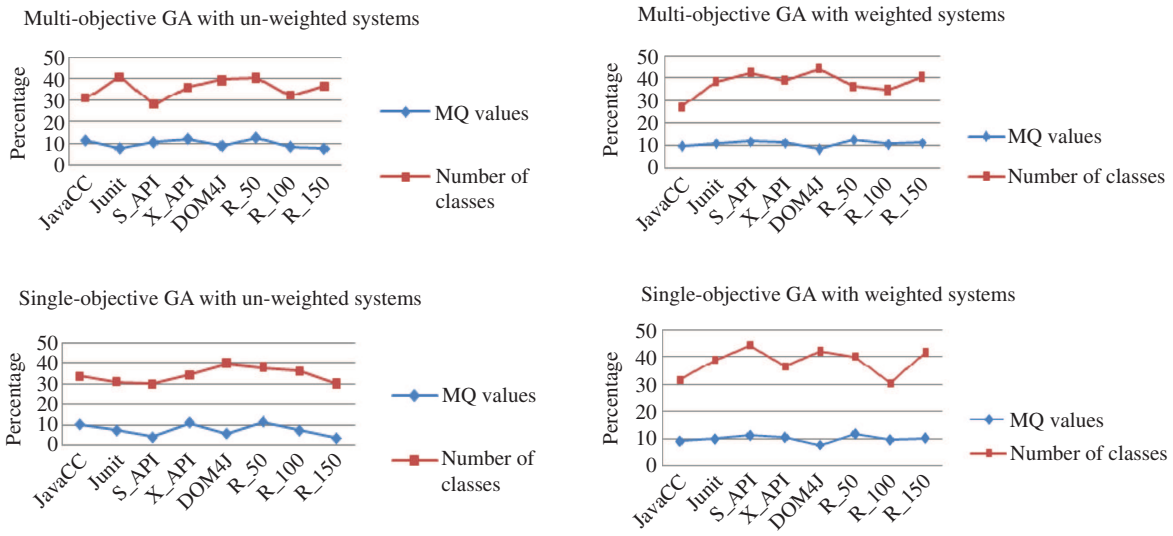

Figure 5: Percentage Reduction in the Number of Moved Class and MQ Values.

optimized, and in the MGA more than one objective is optimized simultaneously. Overall, the above remodularization approach is an effective and useful way of improving the existing package organization of OO software systems.

7 Conclusion and Future Works

This paper presented a new approach for OO software remodularization to improve the quality of the existing package organization with minimum possible perturbation. Such software restructuring exhibiting lesser perturbation is highly useful for maintainers to obtain a significant improvement in modularization quality of software without opting for total remodularization because that can be very costly, time consuming as well as hard to interpret. To achieve the goal, a PMQ metric in terms of the original MQ and MoJoFM metric has been designed as a fitness function. The approach has been evaluated on eight real and random weighted and unweighted software systems. The obtained results provided sufficient empirical evidence that the proposed approach is able to improve the quality of the existing package organization by modifying the original package organization as less as possible. The significance of the results is that by much lesser perturbations, we are able to improve the almost same quality level, which could have been achieved by total remodularization. Hence it can be concluded that the approach proposed in this paper is very useful for the maintainers to improve the structural quality of software with lesser cost and time. The major limitation of the work is that the exploration of the genetic algorithm degrades if the number of objectives increases by more than three. To overcome this limitation, multi-objective-based genetic algorithms can be used. Future work in this direction is possible to include other additional objectives and constraints such that more improvement of the package structures is possible with even lesser perturbation so that this activity can be used more frequently during maintenance.

Bibliography

[1] H. Abdeen, S. Ducasse, H. A. Sahraoui and I. Alloui, Automatic package coupling and cycle minimization, in: *Proceedings of WCRE' 2009*, pp. 103–112, IEEE Computer Society, Lille, 2009.

[2] H. Abdeen, S. Ducasse and H. A. Sahraoui, Modularization metrics: assessing package organization in legacy large object-oriented software, in: *Proceedings of WCRE' 2011*, pp. 394–398, IEEE Computer Society Press, Limerick, 2011.

[3] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil and S. Ducasse, Towards automatically improving package structure while respecting original design decisions, in: *20th Working Conference on Reverse Engineering (WCRE)*, pp. 212–221, IEEE, Koblenz, 2013.
[4] F. B. Abreu and M. Goulao, Coupling and cohesion as modularization drivers: are we being over-persuaded? In: *Proceedings of CSMR' 2001*, pp. 47–57, IEEE, Lisbon, Portugal, 2001.
[5] P. Amarjeet and J. K. Chhabra, An empirical study of the sensitivity of quality indicator for software module clustering, in: *7th International Conference on Contemporary Computing (IC3), 2014*, pp. 206–211, IEEE, Noida, India, 2014.
[6] P. Amarjeet and J. K. Chhabra, TA-ABC: two-archive artificial bee colony for multi-objective software module clustering problem, *J. Intell. Syst.* **27** (2017), 619–641.
[7] P. Amarjeet and J. K. Chhabra, Improving package structure of object-oriented software using multi-objective optimization and weighted class connections, *J. King Saud Univ. Comput. Inf. Sci.* **29** (2017), 349–364. Available online 2 November 2015.
[8] P. Amarjeet and J. K. Chhabra, Harmony search based remodularization for object-oriented software systems, *Comput. Lang. Syst. Struct.* **47** (2017), 153–169.
[9] P. Amarjeet and J. K. Chhabra, Improving modular structure of software system using structural and lexical dependency, *Inf. Softw. Technol.* **82** (2017), 96–120.
[10] P. Amarjeet and J. K. Chhabra, Many-objective artificial bee colony algorithm for large-scale software module clustering problem, *Soft Comput.* **22** (2018), 6342–6361.
[11] P. Amarjeet and J. K. Chhabra, FP-ABC: Fuzzy-Pareto dominance driven artificial bee colony algorithm for many-objective software module clustering, *Comput. Lang. Syst. Struct.* **51** (2018), 1–21.
[12] N. Anquetil, S. Denier, S. Ducasse, J. Laval and D. Pollet, Software (re)modularization: fight against the structure erosion and migration preparation, 2010.
[13] N. Anquetil and T. C. Lethbridge, Experiments with clustering as a software modularization method, in: *Working Conference on Reverse Engineering*, pp. 235–255, IEEE CS Press, Washington, DC, USA, 1999.
[14] N. Anquetil and T. C. Lethbridge, Comparative study of clustering algorithms and abstract representations for software re-modularization, *IEE Proc. Softw.* **150** (2003), 185–201.
[15] S. F. Ardabili, Computational intelligence approach for modeling hydrogen production: a review, *Eng. Appl. Comput. Fluid Mech.* **12** (2018), 438–458.
[16] M. Barros, An analysis of the effects of composite objectives in multiobjective software module clustering, in: *Proceedings of the 14th International Conference on Genetic and Evolutionary GECCC-12*, Terence Soule (Ed.). pp. 1205–212, ACM, New York, NY, USA, 2012.
[17] G. Bavota, A. D. Lucia, A. Marcus and R. Oliveto, Software re-modularization based on structural and semantic metrics, in: *Proceedings of WCRE' 2010*, pp. 195–204, IEEE, Beverly, Massachusetts, USA, 2010.
[18] J. Corwin, D. F. Bacon, D. Grove and C. Murthy, MJ: A rational module system for Java and its applications, in: *Proceedings of the 18th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 241–254, New York, NY, USA, 2003.
[19] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolut. Comput.* **6** (2002), 182–197.
[20] J. Garcia, D. Le, D. Link, A. S. Pooyan Behnamghader, E. F. Ortiz and N. Medvidovic, An empirical study of architectural change and decay in open-source software systems, *Tech. Rep.* USC-CSSE, 2014.
[21] V. Gupta and J. K. Chhabra, Package Coupling measurement in object-oriented software. *J. Comput. Sci. Technol.* **24** (2009), 273–283.
[22] Y. Ichisugi and A. Tanaka, Difference-based modules: a class independent module mechanism, in: *Proceedings ECOOP 2002*, 2374 of *LNCS*, Springer Verlag, Malaga, Spain, 2002.
[23] S. M. R. Kazemi, Novel genetic-based negative correlation learning for estimating soil temperature, *Eng. Appl. Comput. Fluid Mech.* **12** (2018), 506–516.
[24] D. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian and N. Medvidovic, An empirical study of architectural change in open source software systems, Technical Report USC-CSSE-2014-509, Center for Systems and Software Engineering, University of Southern California, 2014.
[25] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. F. Chen and E. R. Gansner, Using automatic clustering to produce high-level system organizations of source code, in: *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98*, pp. 45–53, IEEE, Ischia, Italy, 1998.
[26] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. F. Chen and E. R. Gansner, Bunch: recovery and maintenance of software system structures, in: *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 50–59, IEEE, Oxford, UK, 1999.
[27] S. McDirmid, M. Flatt and W. Hsieh, Jiazzi: new age components for old fashioned Java, in: *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pp. 211–222, ACM, New York, NY, USA, 2001.
[28] H. Melton and E. Tempero, The CRSS metric for package design quality, in: *Proceedings of ACSC' 2007*, pp. 201–210, Australian Computer Society Inc., Darlinghurst, Australia, 2007.

[29] B. S. Mitchell and S. Mancoridis, On the automatic modularization of software systems using the bunch tool, *IEEE Trans. Softw. Eng.* **32** (2006), 193–208.
[30] R. Moazenzadeh, Coupling a firefly algorithm with support vector regression to predict evaporation in northern Iran, *Eng. Appl. Comput. Fluid Mech.* **12** (2018), 584–597.
[31] K. Praditwong, M. Harman and X. Yao, Software module clustering as a multi-objective search problem, *IEEE Trans. Softw. Eng.* **37** (2011), 264–282.
[32] Y. Smaragdakis and D. Batory, Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.* **11** (2002), 215–255.
[33] R. Strnisa, P. Sewell and M. Parkinson, The java module system: core design and semantic definition, in: *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pp. 499–514, ACM, New York, NY, USA, 2007.
[34] R. Taormina, Neural network river forecasting through base flow separation and binary-coded swarm optimization, *J. Hydrol.* **529** (2015), 1788–1797.
[35] P. Tonella, Concept analysis for module restructuring. *IEEE Trans. Softw. Eng.* **27** (2001), 351–363.
[36] Z. Wen and V. Tzerpos, An effectiveness measure for software clustering algorithms.*12th IEEE International Workshop on Program Comprehension*, pp. 194–203, 2004.
[37] T. A. Wiggerts, Using clustering algorithms in legacy systems re-modularization, in: *Working Conference on Reverse Engineering*, pp. 33–43, IEEE, Amsterdam, The Netherlands, 2000.
[38] C. L. Wu, Rainfall-runoff modeling using artificial neural network coupled with singular spectrum analysis, *J. Hydrol.* **399** (2011), 394–409.
[39] S. W. Zhang, Dimension reduction using semi-supervised locally linear embedding for plant leaf classification, *Lect. Notes Comput. Sci.* **5754** (2009), 948–955.
[40] Y. Zhao, Y. Yang, H. Lu, Y. Zhou, Q. Song and B. Xu, An empirical analysis of package-modularization metrics: implications for software fault-proneness. *Inf. Softw. Technol.* **56** (2015), 186–203.