Journal of King Saud University – Computer and Information Sciences (2017) **29**, 349–364

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

Improving package structure of object-oriented software using multi-objective optimization and weighted class connections

Amarjeet *, **Jitender Kumar Chhabra**

Department of Computer Engineering, NIT Kurukshetra, Haryana, India

Received 13 February 2015; revised 21 July 2015; accepted 3 September 2015
Available online 2 November 2015

KEYWORDS

Multi-objective optimization;
Package restructuring;
Modularization;
Weighting scheme;
Maintenance

Abstract The software maintenance activities performed without following the original design decisions about the package structure usually deteriorate the quality of software modularization, leading to decay of the quality of the system. One of the main reasons for such structural deterioration is inappropriate grouping of source code classes in software packages. To improve such grouping/modular-structure, previous researchers formulated the software remodularization problem as an optimization problem and solved it using search-based meta-heuristic techniques. These optimization approaches aimed at improving the quality metrics values of the structure without considering the original package design decisions, often resulting into a totally new software modularization. The entirely changed software modularization becomes costly to realize as well as difficult to understand for the developers/maintainers. To alleviate this issue, we propose a multi-objective optimization approach to improve the modularization quality of an object-oriented system with minimum possible movement of classes between existing packages of original software modularization. The optimization is performed using NSGA-II, a widely-accepted multi-objective evolutionary algorithm. In order to ensure minimum modification of original package structure, a new approach of computing class relations using weighted strengths has been proposed here. The weights of relations among different classes are computed on the basis of the original package structure. A new objective function has been formulated using these weighted class relations. This objective function drives the optimization process toward better modularization quality simultaneously ensuring preservation of original structure. To evaluate the results of the proposed approach, a series of experiments are conducted over four real-worlds and two random software applications. The experimental results clearly indicate the effectiveness of our approach in

* Corresponding author.
E-mail addresses: amarjeetnitkkr@gmail.com (Amarjeet), jitenderchhabra@nitkkr.ac.in (J.K. Chhabra).
Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2015.09.004

improving the modularization quality of existing package structure by doing very small movement of classes between packages of original software modularization.

1. Introduction

The quality of the software modularization has a major impact on many software system quality parameters such as understandability and maintainability (Tonella, 2001; Praditwong et al., 2011). For Object-oriented software systems, the modularization is largely dependent on classes. Classes are nothing but collection of data and associated methods. Generally for small software system, the classes are considered as modules, however for large and complex systems, it has been reported that a set of collaborating classes (i.e. packages) can be a better module of system organization than a class (Gupta and Chhabra, 2009) and bigger software systems are generally designed and developed by using these modularization criteria.

As the software system evolves over the time, addition, removal and modification of classes influence the original software modularization in an adverse manner. It has been observed that due to short deadlines, developers often do not follow the original package design rules during maintenance for early completion of the work leading to modular structure deterioration (Marcio et al., 2014). As a result, the original modularization structure gets modified to the extent that it looses its structural quality due to sub-optimal placement of classes in different packages (Gui and Scott, 2006). Software maintenance is a continuous ongoing phenomenon but the deteriorated structure quality makes the maintenance difficult and negatively affects the software evolution (Mitchell and Mancoridis, 2006). Hence, the re-modularization of a software system becomes essential whenever the system quality gets degraded to a point from where the further evolution is not feasible within permissible time and cost. The software remodularization problem has been solved by many researchers in the past by formulating it as a search-based optimization problem and solved it using meta-heuristic techniques (Mitchell, 2002; Mahadavi et al., 2003; Patel et al., 2009; Abdeen et al., 2009, 2013; Cui and Chae, 2011; Praditwong et al., 2011; Barros, 2012).

Most of the search-based remodularization approaches improve the software structure by optimizing coupling and cohesion criteria (Mahadavi et al., 2003; Praditwong et al., 2011; Barros, 2012). These approaches improved the coupling and cohesion in absolute terms, but the newly suggested package modularization solution usually became so complex and different from the original one, that it would be hardly acceptable to the software maintainer (Marcio et al., 2014). Such methods can be useful when system requires complete overhauling. Such a situation comes once in a while, but not during initial/regular maintenance. At initial/regular maintenance, system needs to be re-modularized with an improved modular structure with less restructuring cost. Hence, the quality criteria need to be modeled in such a way, so that it can drive optimization process with preservation of the original package modularization. In literature, some researchers (Abdeen et al., 2009, 2013; Bavota et al., 2014) have tried to address these problems by controlling the optimization process using some constraints. The authors (Abdeen et al., 2009, 2013) improved the package structure by improving the package coupling, package cohesion and package cyclic dependencies as a single and multi-objective optimization problem. They controlled the optimization process by applying the constraints on movement of the classes among packages. However, defining such constraints is not easy and maintainers must have a deep insight of the original design decisions of software modularization. In most of the cases the maintainers are not the developers of the original modularization (Bavota et al., 2014). In such situations finding the constraints that can drive the optimization process toward design decision of original modularization becomes very difficult. Instead of optimizing through constraints, another approach has been proposed recently by Bavota et al. (2014) where the involvement of end-user becomes compulsory. Their approach is based on structural and semantic dependencies. They controlled the remodularization process by putting the software end-users in the every iteration for requesting the feedback. In this method an end-user must have thorough understanding of design decision of original software modularization, which is again not feasible practically most of the times. Hence it can be stated that importance of original structure of the software plays an important role in the process of remodularization, but existing methods of its inclusion in the optimization process are highly person-specific, and availability of such persons is always a limitation of such approaches. So there is an immense need to incorporate the characteristics of original modular-structure, preferably without necessity of individuals having clear insight of original design. This paper attempts to solve this issue and the proposed approach is able to include the original structure characteristics from the source code, without any need of well-aware end-user/original developer.

This paper presents a multi-objective optimization approach for improving the existing package structure of an object-oriented system aiming at preserving the original design decision of software modularization. To this contribution, the optimization process is controlled by objective functions which are formulated in terms of newly proposed weighted relations that reflect the nature of original design decision. The weights of each type of existing relations are calculated in terms of locality (intra and inter relations) of that relation in original software modularization. Further, these weighted relations are used to calculate overall connection strength among pair of classes. This connection strength helps in keeping the optimization process around original software modularization.

The multi-objective formulation includes package cohesiveness index, package connectedness index, intra-package connection density and package size index as the objective functions. To solve the multi-objective optimization, we used Non-Dominated Sorting Genetic Algorithm (NSGA II) (Deb et al., 2002) a widely-accepted multi-objective evolutionary algorithm. We considered this algorithm, in particular, because

it has been applied successfully previously to solve the similar software remodularization problem domain (Barros, 2012; Abdeen et al., 2013). In order to evaluate the proposed optimization approach, we perform a comparative study with two single-objective optimization algorithms i.e., Simulated Annealing (SA) (Kirkpatrick et al., 1983) and Hill-Climbing (HC) of Bunch API (Mancoridis et al., 1999). A series of experiments were conducted for four real-world and two random software applications. The experimentation results indicate the effectiveness of our approach and we found that our planned multi-objective optimization approach improves the existing package modularization with partially preserving the original package modularization decision.

The remainder of this paper is organized as follows: Section 2 provides related research works. Section 3 presents the multi-objective concepts of software remodularization problem. Section 4 presents the summary of proposed methodology. Section 5 describes the experimental setup. Section 6 presents results and analysis. Section 7 discusses main contribution and limitation of proposed approach and finally Section 8 concludes with future works.

2. Related work

In the previous three decades, a lot of works have been devoted to the automatic remodularization of software systems aiming at improving the quality of system by restructuring the software architecture. Most of the software remodularization approaches are based on the clustering techniques (Anquetil and Lethbridge, 2002; Wu et al., 2005; Bittencourt and Guerrero, 2009). The approaches have been adapted for the various purposes such as for the software module regrouping (Praditwong et al., 2011), module extraction (Mitchell and Mancoridis, 2006), logical component extraction (Erdemir and Buzluca, 2014), etc. These approaches are characterized on the basis of used techniques (e.g., search based, consensus-based, hierarchical and partitioned based), type of entities (e.g., variable, method, class, source file), types of features between entities (e.g., conceptual, semantic, static, dynamic) and type of user interaction (e.g., automatic or semi-automatic) (Erdemir and Buzluca, 2014).

Wiggerts (1997) first presented the theoretical background of clustering based software remodularization algorithms to improve the modularity of system. They also classified the modularization approaches into graph theoretical approach, construction approach, optimization approach, and hierarchical approach. They did not provide any evaluation and just gave the concepts of similarity measurement criteria and the remodularization algorithms. Later Anquetil and Lethbridge (1999) conducted an empirical study to test some of the remodularization algorithms proposed by Wiggerts and performed an empirical study to compare their weakness and strength. Subsequently, Tzerpos and Holt (1999) proposed MOJO metric for the evaluation and comparison of modularizations obtained from different approaches. The MOJO metric is used to measure the distances between two modularizations and also helpful for testing the stability of remodularization algorithm. The MOJO metric is also being used widely in the literature to compare the modularizations (Tzerpos and Holt, 2000). Wu et al. (2005) conducted an empirical study for comparative study of clustering based remodularization algorithms in the context of software evolution. Their results showed that the clustering algorithms based on analytical analysis cannot be adapted for large and complex software systems. Maqbool and Babri (2007) performed a study on the applicability of various hierarchical clustering algorithms in the domain of software architecture recovery. They investigated various similarity and distance measures used in remodularization domain. The remodularization of software systems was also addressed using the pattern clustering (Tzerpos and Holt, 2000), Lexical feature clustering (Bittencourt and Guerrero, 2009), semantic clustering (Kuhn et al., 2007), and consensus- techniques (Forestier et al., 2000; Kashef and Kamel, 2000).

Even after the development of many clustering based software remodularizations, they all show inefficiency for large and complex problems, because of their deterministic nature. To make the software remodularization efficiently solvable, application of search-based meta-heuristic techniques can play a significant role (Harman and Jones, 2001). The software remodularization problems exhibit various features that can be used to formulate it as a search based optimization problem. Mancoridis et al. (1999) were the first who formulated and solved the software remodularization problem using search based optimization technique. The author formulated the remodularization problem as a single-objective optimization problem and solved it by using Hill-Climbing algorithm. Similar to the above approach, Doval et al., 1999 also formulated the software clustering problem as a single-objective optimization problem and proposed a new genetic algorithm to solve the problem. The bad representations of software engineering problem as search problem directly affect the size of search space and thus execution time. To address this issue Harman et al., 2002 proposed a normalized representation for search-based software clustering problem, which reduced the size of the search space and helped to improve the outcome of Genetic Algorithms. The simple Hill Climbing algorithm (Mancoridis et al., 1999) has shown the local minima and efficiency problem. To compare the robustness of various software clustering algorithms Harman et al. (2005) performed an empirical study. They examined the robustness of module clustering fitness function MQ with clustering function EVM (Tucker et al., 2001). Instead of genetic algorithms, the use of the evolution strategy also gained attention in software clustering. Khan et al. (2008) proposed a new software clustering approach based on evolution strategy and their approach produced better results in most of the cases.

Most of the search-based software remodularization has been formulated as a single-objective optimization problem (Mancoridis et al., 1999; Doval et al., 1999). Recently, multi-objective optimization based software remodularization approaches have gained more attention. Praditwong et al. (2011) first formulated the software modularization problem as a multi-objective optimization problem and also proposed a multi-objective evolutionary algorithm (i.e., two-archive genetic algorithm). They concluded that multi-objective formulation of software modularization problem is more useful compared to single-objective formulation. The authors have proposed two multi-objective formulations, i.e., Equal Cluster-size Approach (ECA) and Maximizing Cluster Approach (MCA) and compared with Hill-climbing algorithm. Subsequently, Barros et al. (2012) evaluated the effectiveness of ECA and MCA formulation by deleting and incorporating a

new objective. Their empirical study showed that it could be possible to achieve similar results as of MCA and ECA with reduced objective. Abdeen et al. (2009) proposed single-objective software remodularization to improve the package structure by improving the package coupling, package cohesion and package cyclic dependencies. Later, same authors (Abdeen et al. (2013)) formulated the same work as the multi-objective optimization problem and solved using NSGA-II algorithm. Recently, Barros et al. (2014) performed a case study that addressed the applicability of search-based remodularization technique in the context of software recovery of large and open source software systems. Their study showed that the software remodularization based on search-based optimization techniques requires better model to drive the optimization process instead of current coupling and cohesion measurement. Bavota et al. (2014) proposed a remodularization approach based on structural and semantic dependencies. Their approach is based on the Relational Topic Models (RTM), a probabilistic modeling techniques. The approach succeeded in coupling reduction from 10% to 30% among software modules.

Most of the current approaches on software remodularization focused on improving some quality metrics (e.g., coupling and cohesion), resulting into a totally new modularization to the developer, irrespective of existing module organization (Harman et al., 2002; Mancoridis et al., 1999; Praditwong et al., 2011; Wu et al., 2005). These approaches proposed better modularization in terms of quality metrics, but its structure is totally new from the original package organization. Consequently, modularization solution proposed by such approaches can be difficult to understand and/or to validate.

This paper targets to improve the package structure by moving less number of classes between the packages of the existing software modularization. To accomplish the task, we have proposed weighting scheme for the class relations on the basis of their position in original software modularization and using this concept a new mechanism of software remodularization for package restructuring has been proposed. To the best of our knowledge, such approach has been proposed for the first time in the literature. As different types of relations with different weight are being considered for the first time, instead of just binary values, the proposed approach is bound to provide better results than the former methods. Our experimentation results have supported our belief, and thus an effective and optimal software remodularization approach of existing package organization has been formulated in this paper.

3. Multi-objective software remodularization

To solve the software remodularization problem, the software system is abstracted away as a graph where nodes are modeled with classes and edges with their connections. The software remodularization problem is formulated as a graph partitioning problem where different partitions represent different modularizations. A graph with a set of N nodes can be partitioned into a collection of disjoint subset of N whose union is N. For example, a graph with set of node $N = \{a, b, c\}$ is partitioned into five alternatives $\{\{a\}, \{b\}, \{c\}\}$, $\{\{c\}, \{a, b\}\}$, $\{\{a\}, \{b, c\}\}$, $\{\{b\}, \{a, c\}\}$, $\{\{a, b, c\}\}$. The number of ways to partition a set of n elements into k nonempty subsets is computed by the Stirling numbers of the second kind, $S(n, k)$ (Harris et al., 2000). The $S(n, k)$ is defined as follows:

$$S(n,k) = k \times (S(n-1,k) + S(n-1,k-1), \;\; n \geqslant 1$$

The total number of disjoint subset of a set of n element can be counted by the nth Bell number which is defined as follows:

$$B_n = \sum_{k=1}^{n} S(n,k)$$

where, B_n is the total number of partitions of a software system with n number of classes. The size of B_n grows exponentially with n. For example, $B_1 = 1$, $B_3 = 5$, $B_5 = 52$, $B_7 = 877$, $B_{15} = 1{,}382{,}958{,}545$. The searching for a feasible alternative partition from a graph for the software system becomes problematic as n (i.e. the number of classes) grows. Hence, the graph partitioning problem is categorized into NP-hard problem (Farrugia, 2004). The deterministic or exhaustive search approach cannot solve these problems in reasonable computing time. Hence, formulation of software modularization problem as a search-based optimization problem is the best alternative to find a near optimal solution. The search-based optimization can be single-objective or multi-objective according to the number of objective functions to be optimized. The brief description of single-objective and multi-objective optimization formulation for software remodularization problem is given in the following sub-sections.

3.1. Single objective formulation

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

$$F(M^*) = min/maxF(M)|M \in \Psi$$

where Ψ is the set of all feasible modularizations. M is the software modularization such as F: $\Psi \rightarrow R$ is an objective function. Here function F can be 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 optimization function F and optimization method. Even though single objective optimization methods have been widely applied, still they have some weakness. (1) These single objective methods attempt to optimize just one objective function and this may restrict the modularization solution to a particular software structure property. (2) A single fixed modularization solution returned by single objective approach may not be suitable for the software modularization with multiple potential structures.

3.2. Multi-objective formulation

In multi-objective software optimization, more than one objective is optimized. It determines modularization solutions M^* for which

$$F(M^*) = min(F_1(M), F_2(M), \ldots, F_m(M))|M \in \Psi$$

where m is the number of objective functions and 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 solution. For two modularization solutions M_1, $M_2 \in \Psi$, solution M_1 is said to dominate solution M_2 (denoted as $M_1 \leqslant M_2$) if and only if

$$\forall i \in (1,\ldots,m)\ F_i(M_1) \leqslant F_i(M_2)$$
$$\wedge \exists i \in (1,\ldots,m)\ F_i(M_1) < F_i(M_2)$$

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 developer has more options for selection of best solution based on their requirements.

To deal with software engineering problems with multiple and conflicting design quality criteria, the multi-objective optimization (MOO) technique can be a more suitable mechanism to solve the problem. As mentioned in Section 2 above the remodularization of software system (with minimum changes in original package structure) can be treated as a multi-objective optimization problem and has been attempted by the researchers; but an in-depth knowledge of the original structure related decisions was necessary for all such proposed solutions. As original developers are rarely available for maintenance, we need an approach of remodularization without the need of persons having knowledge of original structure.

4. Proposed approach

The aim of the proposed approach is to improve the quality of the software through remodularization such that original structure is kept into consideration during the process. This can be ensured by lesser movement of classes among packages and non creation/deletion of new packages. To achieve this, the problem is formulated as a multi-objective optimization problem and solved using multi-objective evolutionary algorithm. The general structure of the proposed work is shown in Fig. 1. The different steps of the proposed multi-objective optimization approach are as follows. In the first step, the classes and various types of class relations based on structural aspects are extracted from the software system. In the second step, all extracted relations are assigned a weight according to defined weighting scheme. In the third step, based on the weights of relations the connection strength between the classes is calculated. In the fourth step, a weighted class connection graph (WCCG) is generated and then the representation and multi-objective formulation of WCCG are done. Finally, the multi-objective evolutionary algorithm is applied using multi-objective objective functions.

4.1. Extraction of classes and relation

In object-oriented software system, the classes are considered as an essential unit of organization, and it encapsulates attributes and methods as its elements. The classes in the software system are correlated with each other by various mechanisms. These mechanisms can be determined from different aspects (e.g., structural, dynamic, semantic and conceptual) (Mancoridis et al., 1999; Praditwong et al., 2011; Kuhn et al., 2007; Maqbool and Babri, 2007). But this paper considers only those mechanisms that are based on the structural aspect. Different mechanisms constitute the various types of relations, and hence different kinds of coupling between the

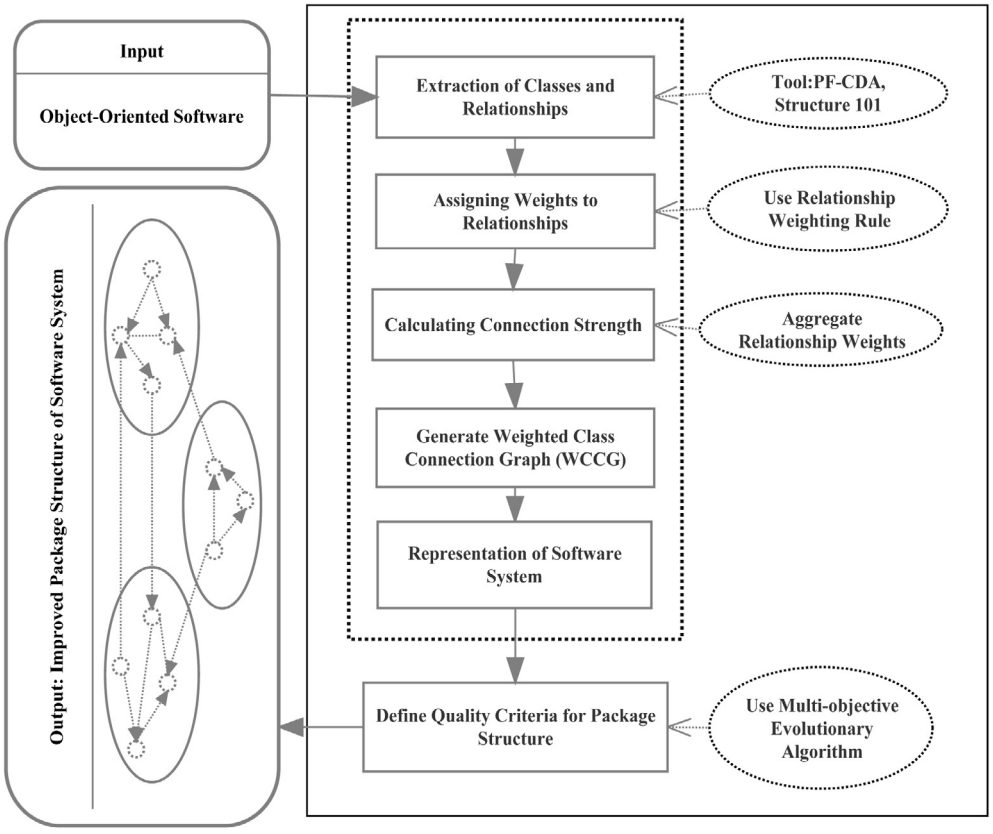

Figure 1 Proposed methodology for software remodularization.

Table 1 Mechanisms that constitute relations between classes.

#	Mechanism	Eder et al. (1994)	Briand et al. (1997)	Hitz and Montazeri (2005)	Erdemir and Buzluca (2014)	Maffort et al. (2015)
1	Method share data (public attribute etc.)	Yes	No	No	Yes	Yes
2	Method references attribute	No	No	Yes	Yes	Yes
3	Method invoke method	Yes	Yes	Yes	Yes	Yes
4	Method receive pointer to method	No	Yes	No	Yes	Yes
5	Class is type of class attribute	Yes	Yes	Yes	Yes	Yes
6	Class is type of method's parameter or return type	Yes	No	Yes	Yes	Yes
7	Class is type of method's local variable	Yes	No	Yes	Yes	Yes
8	Class is type of parameter of a method invoked from within another method	Yes	No	No	Yes	Yes
9	Class is ancestor of another class	Yes	No	Yes	Yes	Yes
10	A class is an exception thrown in a method of other class	No	No	No	No	Yes

classes. Table 1 shows the different mechanisms that form various relations and that have been used by the previous researchers to design the coupling metrics.

Based on the mechanisms presented in Table 1, various relation types can be derived. For example a relation type "extends" can be derived from "*class is ancestor of another class*" mechanism. In software package restructuring, where the classes are moved into a suitable packages, based on various coupling relations; it is necessary to collect all possible relation types from the source code that can be helpful to investigate a software quality indicator. According to different mechanisms, there can be many types of structural relations between classes, but for the sake of simplicity, we consider only eight main types of relevant relations. These eight types of structural relations are the most commonly used relations used by various researchers (Erdemir and Buzluca, 2014; Maffort et al., 2015) as well as tools (Structure 101, stan4j). The authors have studied other kinds of relations also such as contains, declared exception, create instance, public method utilization, but their contribution toward coupling/cohesion gets covered through these eight types, as we have computed these eight types accordingly by widening the scope of their definitions. The brief description of these relations is given as follows:

- *Extends (EX)* – An extend relations (also called an inheritance) implies that one specialized class extends other general class. For example, if class A inherits the methods and attributes of class B, then class A is said to have "Extends" relation with class B.
- *Has Parameter (HP)* – In this type of relation, an object reference of one class is passed to another class as a method parameter. Further using this reference the method and variable related to that object can be accessed. For example, let M_a be a method of class A, there is a "Has Parameter" relation between class A and class B, if class B is type of a parameter of method M_a.
- *Reference (RE)* – In reference class relation one class contains the objects as reference of other class, and by using these objects class makes reference to the attribute of that class. The containing class is responsible for creation and deletion of the contained class reference. For example, there is a reference relation between class A and class B, if a method M_a of class A has an object of class B, and using this object, method M_a references an attribute of class B
- *Calls (CA)* – In this type of relation the method in one class, calls the method in other class by using the reference of that class. For example, there is a "calls" relation between class A and class B, if a method Ma of class A has an object of class B, and using this object, method M_a invokes a method M_b of class B.
- *Implement (IM)* – An implement relation exists between two classes, when one of them must implement, or realize, the behavior specified by the other. For example, if class A implements the methods M_b declared in class B.
- *Is of Type (IT)* – In Is-of Type relation (also called an aggregation relation) a class stores the references to other class for later use. For example, there is an "*Is of Type*" relation between class A and class B, if class B is the type of an attribute of class A.
- *Return (RT)* – A return relation between classes is established, if a class has a method which returns an object of another class. For example let M_a is a method of class A, there is a "return" relation between class A and class B, if class B is the return type of method M_a.
- *Throws (TH)* – A throws relation between classes is established, whenever a method in one class, throws an exception object to an exception handler method, and the method exists in another class. For example class A throws an exception to class B.

All above mentioned relations are directional in nature, i.e. if class A is related to class B, then it is not necessary that class B is also related to class A. A class may be dependent on another class, but the reverse may not be true. For example, if we assume that a class A depends on class B, then it means that any changes in implementation of class B is likely to initiate some changes in the implementation of the class A. But any implementation of class A will not necessarily demand changes in the implementation of the class B. Thus relation between a pair of classes must be considered as a directional entity.

4.2. Assigning weights to relations

The modularization quality of a software system is defined in terms of various quality characteristics and most of these quality characteristics are evaluated in terms of how the classes of the system are connected with each other. Coupling and cohesion are two most important quality characteristics of

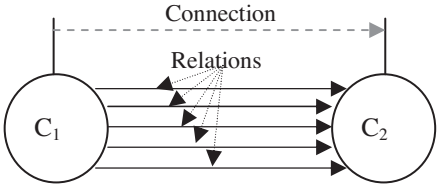

Figure 2 Explanation of class connection and relations.

the software system and a good software modularization exhibits low coupling (inter-module connection strength) and high cohesion (intra-module connection strength). In object-oriented software system, each class is connected with other classes with zero or more relations. Hence, the connection strength between the classes varies according to underlying relation instances and their weights exist between those classes. Fig. 2 depicts the connection and relations between two classes $C1$ and $C2$.

The optimization based software remodularization approaches improve the package structure by optimizing various quality criteria. Hence, the quality criteria defined in terms of connection strength have a major influence on the optimization process as well as produced solution. To improve the meaningfulness of modularization solution produced by our proposed optimization approach, we assign weights to the relations depending on their locality of being external or internal with respect to the original software modularization. The relation weights are calculated as follows:

- Let C denote the set of classes $C = \{c_1, c_2, \ldots, c_n\}$, of the original package organization of software system S; The $|C| = n$ is the number of classes in S.
- Let the set of relation categories $R = \{r_1, r_2, \ldots, r_m\}$, between classes C_i and C_j.
- Let N_k (C_i, C_j) is the number of instances of relation r_k, between classes C_i and C_j.
- Let N_i is the set of classes that are in the same package as class i.
- Let relation weight w_k of class category r_k is calculated as follows:

$$w_k = 1 + \frac{\sum_{i=1}^{n}\sum_{j\in N_i} N_k(C_i, ,C_j)}{\sum_{i=1}^{n}\sum_{j\in N_i} N_k(C_i, C_j) + \sum_{i=1}^{n}\sum_{j\notin N_i} N_k(C_i, C_j)} \quad (1)$$

The weight w_k is varying in the range of [1, 2]. The assignment of weights to relations in such way drives the optimization algorithms toward the original package organization. This weighting scheme technique for relations helps to improve the software package structure as well as partially preserves the original package organization instead of producing totally new software modularization.

4.3. Calculating connection strength

A connection from class A to class B exists if there is/are some relation/s from class A pointing to class B. Like the relations, the connection is also a directed entity. The connection strength from a class to another class is computed by considering the following three aspects:

- The different types of relations between classes that collectively constitute connection.
- The number of instances of relations of a particular type.
- The weights of each type of relations.

The first aspect, the different type of relation between classes, has to be defined when defining the measurement goal. In a class connection there can be more than one type of relations as well as more than one instances of that type of relation, so relation type and number of instances of that type need to be considered. Different types of relations contributed different weights for computing the connection strength. The following notations are used to describe the methods to compute the connection strength (CS) between the classes:

- Let $N_k(C_i, C_j)$ denote the total number of instance of k-relation types, between classes C_i and C_j.
- To compute the connection strength CS (C_i, C_j) from classes C_i to C_j we use the following formula;

$$CS(C_i, C_j) = \begin{cases} \text{Udefined} & \text{if } (i = j) \\ \sum_{k=EX}^{TH} w_k N_k(C_i, C_j) & \text{otherwise} \end{cases} \quad (2)$$

With the help of the formula 2, the strength of connections existing among the classes of software system can be computed. The idea behind the calculation of connection strengths is to palace highly connected classes in the same package and loosely connected classes in a different package.

4.4. Representation of software system

After computing the connection strength among classes, we generate a weighted class connectivity graph (WCCG) for the object oriented software system. The WCCG $G < V, E >$, where V is the set of vertices $\{v_i\}$ corresponding to classes and E is the set of edges $\{e_i\}$ corresponding to connection between the classes. Module (i.e., package) M_i in class connectivity graph is a group of correlated classes or interfaces connected with each other and that can also be connected with other module in the system. The definition of module is as follows:

- M_i is the group of classes and subset of graph G; $M_i \subseteq G$ such that $M_i = < V_i, E(V_i) >$, where V_i is the set of classes in $M_i, V_i \subseteq V$ and $E(V_i)$ is the set of all connections between classes in V_i.
- Module set $M = \{M_1, M_2, \ldots, M_k\}$, and $|M| = k$ is the number of module.
- M_i is the non empty set of classes: $V_i \neq, \phi$, and $E(V_i) \neq, \phi$, $i = 1,2, \ldots ,k$.
- Modules are disjoint set. Two different modules M_i and M_j cannot have common classes and connections $M_i \cap M_j = \phi, \forall(1 \leqslant i, j \leqslant k \wedge (i \neq j))$. This property follows that $V_i \cap V_j = \phi \wedge E(V_i) \cap E(V_j) = \phi \forall(1 \leqslant i, j \leqslant k \wedge (i \neq j)).)$

An example of class connectivity graph of a hypothetical object-oriented software system with eight classes is illustrated in Fig. 2, where,

V = {$C0$, $C1$, $C2$, $C3$, $C4$, $C5$, $C6$, $C7$}.
E = {$e0$, $e1$, $e2$, $e3$, $e4$, $e5$, $e6$, $e7$, $e8$}, where $e0$ = {CA, CA, RE}, $e1$ = {EX}, $e2$ = {RE, CA}, $e3$ = {IM}, $e4$ = {CA}, $e5$ = {HP,TH, CA, RE}, $e6$ = {CA, CA}, $e7$ = {HP}, $e8$ = {CA, RE, RE}.
M = {$M1$, $M2$, $M3$}, where $M1$ = Package-1, $M2$ = Package-2 and $M3$ = Package-3.
$M1$ = {$C0$, $C3$, $C4$}, $M2$ = {$C5$, $C6$, $C7$}, $M3$ = {$C1$, $C2$}

To apply the evolutionary algorithm on software modularization problem, the representation of software system needs to be defined (Praditwong et al., 2011). In case of software remodularization, we model the software system as the WCCG. To represent the WCCG, a simple array is used, where classes map to array index and packages to array elements. The array representation of the WCCG for hypothetical software system given in Fig. 3 is represented as {1, 3, 3, 1, 1, 2, 2, 2}. For example classes $C0$, $C3$ and $C4$ are in same module (i.e. Package-1).

4.5. Remodularization objectives

After defining the suitable representation of software system, next we have to formulate the quality characteristics that can be used as an objective function to guide the optimization process toward a better package structure. The inter-module and intra-module connections are widely used criteria for the various measurement goals (i.e., maintainability, understandability and reusability) of the package structure optimization. In software package re-structuring, beyond these quality characteristics, use of some other quality criteria can also be helpful for driving solution space toward the better quality package structure. That is the reason of incorporation of multi-objective optimization approach to improve the quality of package structure. The detailed descriptions of these objective functions are given as follows:

4.5.1. Package cohesiveness index (PCI_{coh})

The package cohesiveness index (PCI_{coh}) is used to measure the extent to which the classes within each package are connected to other classes of the same package. The package cohesiveness $PC_{coh}(M)$ measures the total intra-package connection strength of a modularization M for the software system. For a given modularization M, with total n number of classes, it can be calculated by:

$$PC_{coh}(M) = \sum_{i=1}^{n} \sum_{j \in N_i} CS(C_i, C_j) \tag{3}$$

where N_i is the set of classes that are in the same package as class i, $CS(C_i, C_j)$ is the connection strength between classes C_i and C_j. The PC_{coh} (M) value ranges between zero, when all classes are in separate packages, and the sum of all connection strength, when all classes are in a single, fully-connected package. For many software applications, high-quality software package will have high connection strengths within packages (and thus high intra-package cohesiveness). The package cohesiveness index is as follows:

$$PCI_{coh} = \frac{PC_{coh}(M)}{PC_{\max}(M)} = \frac{\sum_{i=1}^{n} \sum_{j \in N_i} CS(C_i, C_j)}{\sum_{i=1}^{n} \sum_{j=1 \wedge i \neq j}^{n} CS(C_i, C_j)} \tag{4}$$

The value of the PCI evaluates to zero when all classes are in separate packages (i.e., number of package equal to number of classes), because every class is "perfectly" connected to every other class in each package. Conversely, it becomes one when all classes are in one package. Hence, the PCI value needs to be maximized for better software package structure.

4.5.2. Package connectedness index (PCI_{con})

The package cohesiveness index, incorporates the within-package connection strengths, but does not consider across package boundaries. If a package boundary cuts through low- connection strength, the effect of package to remaining other packages would be insignificant. To measure the connection strength between packages we define the package connectedness $PC_{con}(M)$, which can be calculated as follows:

$$PC_{con}(M) = \sum_{i=1}^{n} \sum_{j \notin N_i} CS(C_i, C_j) \tag{5}$$

The package connectedness index (PCI_{con}) measures the extent to which classes in different packages are connected to one other. The PCI_{con} for a given modularization is evaluated as follows:

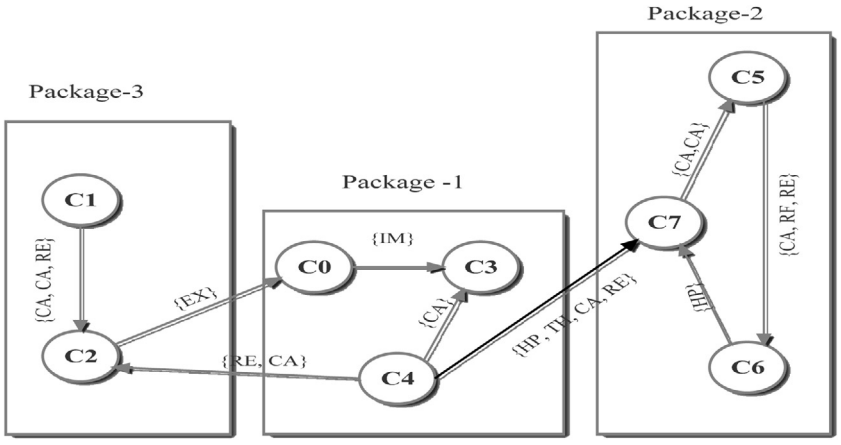

Figure 3 WCCG of a hypothetical object-oriented software system.

$$PCI_{con} = \frac{PC_{con}(M)}{PC_{\max}(M)} = \frac{\sum_{i=1}^{n}\sum_{j\notin N_i} CS(C_i, C_j)}{\sum_{i=1}^{n}\sum_{j=1\wedge i\neq j}^{n} CS(C_i, C_j)} \quad (6)$$

Unlike PCI_{coh}, the PCI_{con} value for any modularization M, is evaluated to zero when all classes are in the same package, because there are no connections to other packages. For a modularization where all classes are in different packages, the PCI_{con} is evaluated as one. Hence, for better software package structure, PCI_{con} value needs to be minimized for better software package structure.

4.5.3. Intra-package connection density (ICD)

The package modularization solution, where all classes are packaged into a single module, exhibits a best package connectedness, however it is not the best possible package cohesiveness. Hence, the goal of *ICD* is to limit the excessive package connectedness, but not to eliminate package connectedness altogether. The *ICD* is trade-off between coupling and cohesion. For each package k, ICD_k is calculated as the ratio between internal and external relation weights, which has to be maximized. Finally, the ICD of the overall package modular structure is calculated as the average of every ICD_k.

$$ICD_k = \frac{CI_k^{in}}{CI_k^{in} + CI_k^{out}} = \frac{\sum_{i=1\wedge i\in N_k}^{n}\sum_{j\in N_k} CS(C_i, C_j)}{\sum_{i=1\wedge i\in N_k}^{n}\sum_{j\in N_k} CS(C_i, C_j) + (\sum_{i=1\wedge i\in N_k}^{n}\sum_{j\notin N_k}^{n} CS(C_i, C_j) + \sum_{i=1\wedge N_k}^{n}\sum_{j\notin N_k}^{n} CS(C_j, C_i))} \quad (7)$$

$$ICD = \sum_{k=1}^{p}\frac{ICD_k}{p} \quad (8)$$

where CI_k^{in} is the connection strength between classes allocated in the same package? CI_k^{out} represents the connection strengths between package k and others. For better modularization solution, *ICD* needs to be maximized.

4.5.4. Package size index (PCI)

The PCI_{coh}, PCI_{coup} and *ICD* quality criteria may not lead to the best package structure since its definition does not take into account the class distribution in the package. This objective function is used to avoid the extremely skewed distribution of classes in packages (e.g., $n-1$ packages with just one class and all remaining classes in an nth package). To handle such situation we use the *PCI* which is defined as follows:

$$PCI = \frac{P_{\min}}{P_{\max}} \quad (9)$$

where P_{min} and P_{max} represent the minimum and maximum number of classes in the packages respectively. If the size of large package increases and small one decreases, then the value of *PCI* becomes smaller. Hence, the optimization process must avoid reduction in *PCI*.

4.5.5. Fitness function

To evaluate the aggregate quality (or fitness F) of a software modularization solution, we use the additive aggregate fitness function, calculated as the weighted sum of the four modularization quality measure above.

$$F = w1 * PCI_{coh} - w2 * PCI_{con} + w3 * ICD + w4 * PCI \quad (10)$$

where $\{w1, w2, w3, w4\}$ are scalar values that define the relative importance of PCI_{coh}, PCI_{coup}, *ICD* and *PCI* respectively. But in this paper we use the value of $w1 = w2 = w3 = w4 = 1$.

5. Experimental setup

We have conducted a series of experiments on different real world and random software systems with our proposed multi-objective optimization approach. The goal of the experiment is to investigate the quality of produced software modularization of our weighted relation approach and compare it to the quality of modularization of un-weighted relation. The multi-objective formulation of our approach is also compared with single-objective optimization approaches.

5.1. Algorithms and parameters

To solve our proposed multi-objective optimization approach, we use the Non-dominated Sorting Genetic Algorithms (NSGA-II) (Deb et al., 2002). It is a meta-heuristic genetic algorithm based on the non-domination sorting concepts of multi-objective optimization technique. It generates a set of non-dominated solutions that is known as the Pareto set. The primary reason for choosing NSGA-II is that it has been reported in the recent literature that it performs well in similar problems (Barros, 2012; Abdeen et al., 2013). Moreover, NSGA-II's performance has been compared with two other applicable algorithms Simulated Annealing (SA) algorithm (Kirkpatrick et al., 1983) & Hill-Climbing (HC) algorithms of Bunch API (Mancoridis et al., 1999) and our results clearly indicate that NSGA-II outperforms other two algorithms in the situations for which our approach has been proposed. This paper uses the same operator configuration for NSGA-II algorithms as given in literatures (Barros, 2012; Abdeen et al., 2013). There are three main operators in NSGA-II i.e., crossover, mutation and selection operator. They used the single point crossover operator with 80% crossover probability for problem that has less than 100 classes and 100% for problem having classes greater than 100. For mutation operator, the uniform mutation operator with $0.04 * \log_2(N)$ mutation probability, where N is the number of classes has been used. The population size is set to 10 times the number of classes and the number of generation is set as 200 times the square of the number of classes. For SA algorithm the parameter setting is same as Abdeen et al. (2009) and for HC algorithm the parameter setting is same as given in Bunch API (Mancoridis et al., 1999).

5.2. Problem instance selection

We used a set of well-known four real-world software systems including JavaCC5.0, JUnit, Java Servlet API and XML API DOM and two random object-oriented software system instances with different size and characteristics. The

Table 2 Characteristics of problem instances.

Problem Instances	Version	Abbreviation	#Connections	#Package	#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
Random problems					
Random50	NA	R1	218	7	50
Random100	NA	R2	342	12	100

real-worlds software systems instances are based on the java programming language and are open-source or free-software projects. The details about the selected object-oriented software systems instances are given in the following Table 2.

To apply the proposed approach, the selected object-oriented software systems are modeled in terms of weighted class connectivity graph (WCCG). The software systems are considered as weighted relation WCCG, if the relation weights are assigned using the Eq. (1), and un-weighted relation WCCG, if the relation weights are assigned binary value.

5.3. Collecting results from experiment

As the NSGA-II, SA and HC algorithms produce probabilistic results in each run, hence to collect the results for analysis, they are executed 30 times for all problem instances. The SA and HC algorithms produce a single solution at each running cycle, and the mean of fitness function F of the solutions produced by SA and HC algorithms of each 30 runs can be calculated directly. However the NSGA-II produces a set of solutions called Pareto set for each running cycle. The each running cycle for each problem instance contains a set of Pareto front and Pareto set in objective and solution space. After running all cycles for a given problem instance, the Pareto front of each of 30 running cycles is collected. Next to select a representative solution from each Pareto set, we use the method proposed by the Abdeen et al. (2013). The method first calculates the arithmetic mean value of each objective function f (PCI_{coh}, PCI_{coup}, ICD and PCI) for all Pareto set solutions. Then it selects a single representative solution from each running cycle using the following equation.

$$S_{\text{chosen}} \Longleftrightarrow \min_{i=1}^{|\text{Pareto Front}|} \left(\sqrt{\sum_{f_j \in F} \left(f_j(s_i) - \bar{f}_j\right)^2} \right) \tag{11}$$

where f_j (s_i) represent the jth objective function value of ith Pareto set solution. The $\bar{f}_i$ represent the arithmetic mean of jth objective function values and S_{chosen} represent the selected solution of a cycle corresponding to that software metric.

5.4. Evaluation metrics

In order to evaluate the modularization solution produced by our proposed software remodularization approach different quality measures have been adapted. These include Rate per Refactoring of Achieved Improvement ($RRAI$) (Abdeen et al., 2013), modularization merit factor (MMF) (Abreu and Goulao, 2001), Modularization Quality (MQ) (Mancoridis et al., 1999). The detailed explanation of these measures is given as follows.

5.4.1. Rate per refactoring of achieved improvement (RRAI)

To assess the degree of modification in the produced package structures with regard to original package structure, we use the Rate per Refactoring of Achieved Improvement ($RRAI$) measurement criteria proposed by Abdeen et al., 2013. The $RRAI$ is defined as follows:

$$RRAI(SM) = \frac{RPMC(SM)}{RPC(SM)} : SM \in \{\text{Coupling}, \text{Cohesion}, MQ, \ldots\} \tag{12}$$

Here we calculate $RRAI$ with respect to solution corresponding to MQ value. The RPC (MQ) represents rate per class of MQ measurement and it is computed as follows: RPC (MQ) = ${}^{MQor}/|C|$ where MQ or is the value of MQ of original software module 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: $RPMC$ (MQ) = ${}^{\delta}{}_{MQ}/MD$, where ${}^{\delta}{}_{MQ}$ is increased value of MQ in new modularization and MD is the count of the classes, which changes their package in new modularization. The larger the value of $RRAI$ (MQ), the smaller is the modification with respect to the MQ measurement.

5.4.2. Modularization merit factor (MMF)

The modularization merit factor measures the intra-package connection density with constant number of packages and classes. If value of MMF improves for the new solution, it indicates an increase in the cohesion value and decrease in the coupling value. The detailed description of MMF is given as follows:

$$MMF = \left(\frac{\sum_{i=1}^{n} \sum_{j \in N_i} \sum_{k=EX}^{TH} N_k(C_i, C_j)}{\sum_{i=1}^{n} \sum_{j \in N_i} \sum_{k=EX}^{TH} N_k(C_i, C_j) + \sum_{i=1}^{n} \sum_{j \notin N_i} \sum_{k=EX}^{TH} N_k(C_i, C_j)} \right) \times \frac{N_p}{N_c} \tag{13}$$

The N_P and N_C represent the total number of packages and total number of classes of the system. We evaluate the improvement in the normality of the MMF as follows:

$$MMF_{IMPROV} = \frac{MMF_{PROP} - MMF_{ORIG}}{MMF_{ORIG}} \tag{14}$$

where MMF_{PROP} indicates MMF for the proposed solution, MMF_{ORIG} shows the MMF for the original solution and MMF_{IMPROV} represents relative improvement of MMF.

5.5.3. MQ metrics

The modularity quality is measured in terms of class connection strength between the packages and within the package. Mancoridis et al. (1999) first time formulated it as the sum of Modularization Factor (MF_l) where the MF_l is the ratio of total connection strength within package k and total connection strength to rest of l. The MF_l for package k can be determined as follows:

$$MF_l = \frac{\sum_{i=1}^{m}\sum_{j=1\wedge j\neq i}^{m}\sum_{k=EX}^{TH} N_k(C_i, C_j)}{\sum_{i=1}^{m}\sum_{j=1\wedge j\neq i}^{m}\sum_{k=EX}^{TH} N_k(C_i, C_j) + \frac{1}{2}\left(\sum_{i=1}^{m}\sum_{j\notin l}^{n}\sum_{k=EX}^{TH} N_k(C_i, C_j) + \sum_{i=1}^{m}\sum_{j\notin l}^{n}\sum_{k=EX}^{TH} N_k(C_j, C_i)\right)} \tag{15}$$

where m is the total number of classes within a particular package l and n is the total number of classes in the software system. MQ can be determined in terms of MF as:

$$MQ = \sum_{l=1}^{p} MF_k \tag{16}$$

Eq. (7) evaluates the modularity quality MQ, where p is the total number of packages. The MQ measure shows a tradeoff between package connectedness and package cohesiveness.

6. Results and analysis

This section presents the experimentation results obtained from the proposed weighted relation based software remodularization approach. In order to demonstrate that the modularization solution produced through weighted relations is more useful, we also obtained the modularization solution from the un-weighted relation. The results have been computed over six different software for the proposed approach and evaluated through three evaluation criteria namely $RRAI$/moved-classes, MMF and MQ, as explained in Section 5.4 above and their discussions are mentioned in the following three subsections. The mean and standard deviations of percentage improvement in each evaluation metrics are collected using each algorithm for each type of problem instance (weighted and un-weighted relation) over 30 execution cycles. The mean differences between algorithms (NSGA-II and SA, NSGA-II and HC) were obtained with two-tailed Wilcoxon Tests. The asterisk (*) denotes the statistically significant mean differences at $\alpha = 0.05$. The mean differences with bold-face indicate that the NSGA-II algorithm performs better than others.

6.1. Moved classes

Table 3 presents the percentage of classes that change their original packages in modularization solution obtained through NSGA-II, SA and HC algorithms over both weighted and un-weighted relation of OO systems. The results of un-weighted relation indicate that a large number of classes change their packages. For example, in NSGA-II algorithm the maximum percentage of movement of classes is 89.12% (in case of Random 50). It is important to note that even the minimum percentage of movement of classes in such situations is very high i.e., 65.36% (in case of Random 100). If we compare the NSGA-II with SA and HC for un-weighted relations, the results show that the NSGA-II performs better in some problems, while SA/HC algorithm performs better in others. Hence it is difficult to conclude which algorithm is better for un-weighted relations. Further, if we see the $RRAI$ (MQ) values given in Table 4 for un-weighted relations, $RRAI$ (MQ) values for all problems are strictly lesser than the baseline value (which is 1), which indicates that there are very large number of classes changing their original packages. Hence, both results obtained through un-weighted relations clearly indicate that large numbers of classes change their packages in final modularization solutions, which is hardly acceptable by the maintainers.

Weighted relations compared to un-weighted relations

The modularization results obtained through our proposed approach (i.e., the weighted relations) given in Tables 3 and 4 show that there are very small percentages of classes changing their original package. For example, the lowest movement of classes is only 8.86% in case of Java Servlet API for NSGA-II algorithm; moreover, the highest value of the movement of classes is 17.51% (for Java CC) which is very low compared to un-weighted relations. It may be observed that the $RRAI$ (MQ) values given in Table 4 for all weighted relations are strictly greater than the baseline value (which is 1) clearly showing that these results are better and more acceptable. If we compare the movement of classes obtained by the NSGA-II with SA and HC, the results show that the NSGA-II algorithm has the significantly small number of class movement compared to SA and HC algorithm. On the basis of these empirical evidences, it can be stated that the proposed approach performs small modifications in the original software package structure, as desired by the maintainers.

6.2. Relative improvement of the MMF

Table 5 presents descriptive statistics of MMF_{IMPROV} improvement of all weighted and un-weighted problem instances with NSGA-II, SA and HC algorithms. The results obtained through un-weighted and weighted relations indicate that the value of MMF_{IMPROV} improves in all algorithms. For example, in weighted relation with NSGA-II algorithms the minimum improvement is observed in XML API DOM problem instance i.e., 18.6% and maximum improvement is observed in JUnit problem instance i.e., 58.8%. Table 5 also shows that the NSGA-II algorithm performs significantly

Table 3 Percentage of Moved classes from their packages.

	Systems	NSGA-II		SA		HC		Mean differences	
		Mean	STD	Mean	STD	Mean	STD	NSGA-SA	NSGA-HC
Un-weighted relation	JavaCC	77.65	10.59	81.32	15.25	78.52	12.04	**−3.67**	**−0.87**
	JUnit	88.34	9.58	85.65	11.85	89.28	10.68	+2.69	**−0.94**
	Java Servlet API	68.87	13.25	71.29	13.58	76.63	13.84	**−2.42**	**−7.76**
	XML API DOM	76.34	11.59	78.61	12.45	76.95	11.48	**−2.27**	**−0.61**
	Random50	89.12	14.12	87.35	13.49	85.16	9.56	+1.77	+3.96
	Random100	65.36	11.27	74.12	10.24	79.58	8.87	**−8.76***	**−14.22***
Weighted relation	JavaCC	17.51	3.17	28.67	2.66	32.53	3.27	**−11.16***	**−15.02***
	JUnit	13.34	2.56	22.54	2.26	29.81	2.52	**−9.2***	**−16.47***
	Java Servlet API	8.86	1.45	17.34	1.54	22.41	1.76	**−8.48***	**−13.55***
	XML API DOM	14.62	2.39	23.56	2.39	31.54	2.16	**−8.94***	**−16.92***
	Random50	6.17	1.43	12.45	2.13	26.31	1.69	**−6.28***	**−20.14***
	Random100	11.64	1.87	19.34	1.89	28.48	2.15	**−7.70***	**−16.84***

Table 4 *RRAI* (*MQ*) evaluation results.

	Systems	NSGA-II		SA		HC		Mean differences	
		Mean	STD	Mean	STD	Mean	STD	NSGA-SA	NSGA-HC
Un-weighted relation	JavaCC	0.445	0.004	0.328	0.021	0.256	0.005	**+0.117***	**+0.189***
	JUnit	0.451	0.012	0.543	0.007	0.412	0.007	−0.092	**+0.039***
	Java Servlet API	0.365	0.006	0.363	0.003	0.364	0.009	**+0.002**	**+0.001**
	XML API DOM	0.548	0.008	0.545	0.007	0.546	0.007	**+0.003**	**+0.002**
	Random50	0.326	0.011	0.418	0.009	0.485	0.004	−0.092	−0.159
	Random100	0.542	0.007	0.371	0.005	0.279	0.006	**+0.171***	**+0.263***
Weighted relation	JavaCC	2.842	0.012	1.231	0.011	2.125	0.047	**+1.611***	**+0.717***
	JUnit	4.690	0.054	2.387	0.032	1.984	0.015	**+2.303***	**+2.706***
	Java Servlet API	3.217	0.052	1.177	0.054	2.164	0.053	**+2.040***	**+1.053***
	XML API DOM	2.764	0.086	1.652	0.076	1.326	0.044	**+1.112***	**+1.438***
	Random50	3.361	0.031	1.760	0.051	1.276	0.065	**+1.601***	**+2.085***
	Random100	6.563	0.039	2.435	0.043	2.896	0.038	**+4.128***	**+3.667***

Table 5 Percentage improvement in *MMF*.

	Systems	NSGA-II		SA		HC		Mean differences	
		Mean	STD	Mean	STD	Mean	STD	NSGA-SA	NSGA-HC
Un-weighted relation	JavaCC	0.431	0.014	0.416	0.025	0.422	0.024	**+0.015***	**+0.009**
	JUnit	0.588	0.032	0.542	0.021	0.576	0.031	**+0.046***	**+0.012**
	Java Servlet API	0.214	0.016	0.193	0.018	0.193	0.026	**+0.017***	**+0.021***
	XML API DOM	0.186	0.047	0.161	0.016	0.156	0.019	**+0.025***	**+0.030***
	Random50	0.778	0.017	0.761	0.012	0.752	0.013	**+0.017***	**+0.026***
	Random100	0.614	0.013	0.589	0.015	0.577	0.028	**+0.025***	**+0.033***
Weighted relation	JavaCC	0.426	0.012	0.413	0.028	0.419	0.023	**+0.013***	**+0.007**
	JUnit	0.586	0.071	0.538	0.068	0.563	0.073	**+0.048***	**+0.023***
	Java Servlet API	0.213	0.033	0.187	0.035	0.181	0.038	**+0.026***	**+0.032***
	XML API DOM	0.184	0.062	0.151	0.058	0.153	0.055	**+0.033***	**+0.031***
	Random50	0.773	0.017	0.756	0.023	0.747	0.043	**+0.017***	**+0.026***
	Random100	0.611	0.008	0.587	0.013	0.574	0.024	**+0.024***	**+0.037***

better in comparison with SA and HC algorithms for both weighted and un-weighted relations.

Weighted relations compared to un-weighted relations

As reported in Table 5 above, values of *MMF* improve for the weighted relations also. For example, the minimum improvement is recorded as 18.4% for XML API software system; however maximum improvement is recorded as 77.3% for random50 in NSGA-II algorithm. If we compare MMF_{IMPROV} values of weighted relations with un-weighted relations, improvement in MMF_{IMPROV} values is slightly lesser that corresponding un-weighted relation based MMF_{IMPROV} values. For example, in NSGA-II algorithm, the MMF_{IMPROV} values

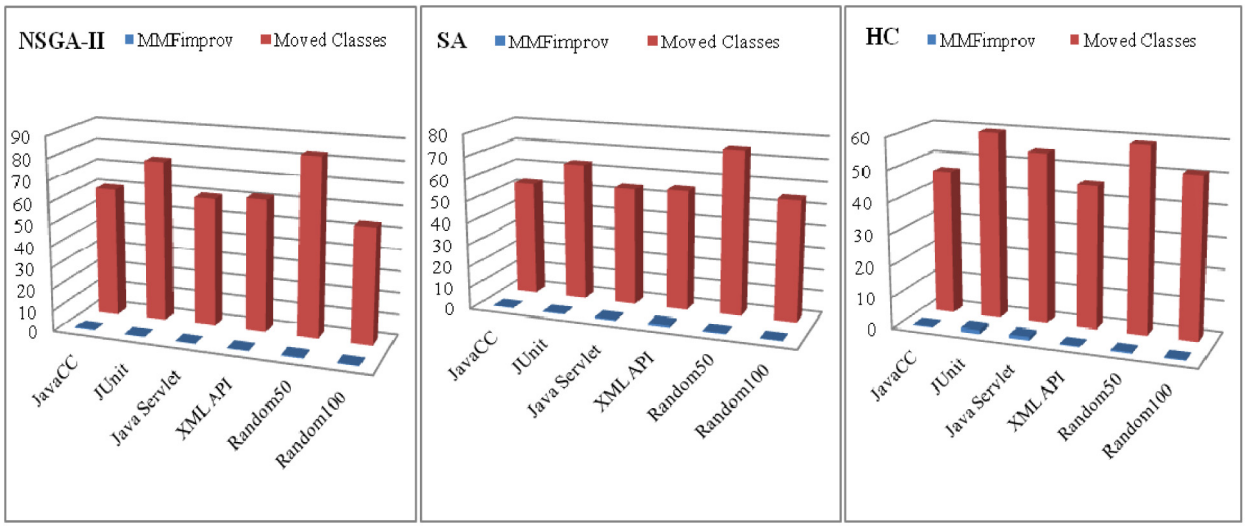

Figure 4 Percentage of reduction in MMF_{IMPROV} vs. percentage of reduction in moved classes.

for un-weighted relation JavaCC problem instance are 43.1%, while the MMF_{IMPROV} values for weighted relation of counterpart of the same problem are 42.6% (i.e., 0.5% reduction). This fractional reduction happens due to additional constraints imposed to keep the solution closer to original structure. On the other hand, if we compare the NSGA-II with SA and HC algorithms, the result indicates that the NSGA-II algorithm performs significantly better than both algorithms in both un-weighted relation and weighted relations. So it can be concluded that NSGA-II algorithm is better than other two algorithms. Further weighed relation based approach is able to keep the solution much closer to original modular-structure and yet is able to improve the *MMF* almost equivalent to un-weighted relations.

6.3. MMF_{IMPROV} vs. moved classes

From the Section 6.2, it is clear that the MMF_{IMPROV} values obtained through weighted relations improve almost equivalent to un-weighted relations. However, Section 6.1 shows that the movement of classes from their original packages in weighted relation is very small compared to their un-weighted counterpart. It indicates that we can stop a large number of class movements by compromising a little with quality. The results can be better understood through a graph based representation, as given below in Fig. 4. Instead of looking at absolute values of movement and *MMF*, Fig. 4 plots the percentage of movement and change in *MMF* i.e. difference of values of un-weighted relations from their weighted counterpart, for all six software for all three algorithms (NSGA-II, SA and HC). From the figure, it can be clearly observed that in all algorithms, the weighted relations are able to reduce movement of a very large percentage of classes from their original packages on the cost of very small compromise in MMF_{IMPROV} values.

6.4. MQ metrics

Table 6 presents descriptive statistics of *MQ* improvement of all weighted and un-weighted problem instances with NSGA-II, SA and HC algorithms. The results obtained through un-weighted and weighted relations indicate that the

Table 6 Percentage improvement in *MQ*.

	Systems	NSGA-II		SA		HC		Mean differences	
		Mean	STD	Mean	STD	Mean	STD	NSGA-SA	NSGA-HC
Un-weighted relation	JavaCC	23.14	3.18	20.05	3.28	20.58	3.56	**+3.09***	**+2.56***
	JUnit	61.16	3.78	56.47	3.88	57.14	3.87	**+4.69***	**+4.02***
	Java Servlet API	28.42	2.68	26.56	3.23	24.56	3.18	**+1.86***	**+3.86***
	XML API DOM	33.01	2.66	31.13	3.02	29.98	3.01	**+1.88***	**+3.03***
	Random50	40.78	3.58	34.16	2.88	38.47	3.55	**+6.62***	**+2.31***
	Random100	65.17	4.18	60.87	3.54	60.88	4.42	**+4.30***	**+4.29***
Weighted relation	JavaCC	20.34	3.13	15.66	3.45	15.23	3.26	**+4.68***	**+5.11***
	JUnit	56.31	4.18	52.34	4.23	53.65	4.25	**+3.97***	**+2.66***
	Java Servlet API	24.62	2.65	22.21	3.14	21.87	3.14	**+2.41***	**+2.75***
	XML API DOM	30.23	2.51	27.37	2.76	26.56	2.78	**+2.86***	**+3.67***
	Random50	33.61	3.28	30.12	3.56	31.76	3.79	**+3.49***	**+1.85***
	Random100	59.23	4.37	56.45	3.67	54.87	4.65	**+2.78***	**+4.36***

value of *MQ* improves in all algorithms clearly meaning that quality improves after re-modularization. The improvement in *MQ* is significant in all cases. It gets verified from the fact that the minimum improvement is 20.34% (in JavaCC problem instance for weighted relation with NSGA-II algorithms), and maximum improvement is 65.17% (in Random100 problem instance for NSGA-II with un-weighted relation). Table 6 also indicates that the NSGA-II algorithm performs significantly better in comparison with SA and HC algorithms for both weighted and un-weighted relations.

6.4.1. Weighted relation compared to un-weighted relation

As reported in Table 6 above, values of *MQ* improves for the weighted relations also. For example, the minimum improvement is recorded as 20.34% for JavaCC software system; however maximum improvement is recorded as 59.23% for random100 in NSGA-II algorithm. If we compare *MQ* values of weighted relations with un-weighted relations, improvement in *MQ* values is slightly lesser than the corresponding un-weighted relation based *MQ* values. For example, in NSGA-II algorithm, the *MQ* values for un-weighted relation JUnit problem instance is 61.16%, while the *MQ* values for weighted relation of counterpart of the same problem is 56.31% (i.e., 4.85% reduction). This small drop in the value of *MQ* is expected, as we are targeting to keep our solution closer to initial design-structure. Comparison of results of NSGA-II with SA and HC algorithms clearly demonstrates the superiority of NSGA-II algorithm for both un-weighted relation and weighted relations. Hence we can state that NSGA-II algorithm is better than other two algorithms and the proposed approach is able to adequately improve the quality of modularization within the specified constraints.

6.5. MQ values vs. moved classes

Similar to Section 6.3 above, in Fig. 5 below, Red bars show, in percentage form, the difference of movement of classes for un-weighted relation and weighted relations. Blue colored bars show in percentage, the difference in *MQ* values for un-weighted and weighted relations. The figure shows bar-charts for all six problem instances for all three algorithms (NSGA-II, SA and HC). The bar-charts clearly show that in all algorithms, the weighted relations result into a reduction of movement of a very large percentage of classes from their original packages, with a very small dip in *MQ* values.

From the above results of *RRAI*/moved-classes, *MMF* & *MQ*, it is clearly evident that weighted relation based approach is able to adequately improve the different quality parameters of the software and at the same time reduces significantly the movement of classes from their original packages. Hence it can be stated that the proposed approach is very useful for remodularization whenever maintainers need that, but cannot afford to have complete overhauling of the modular-structure and when the design structure of the original software is also to be kept in consideration.

7. Discussions

In this section, we discuss the main contributions and limitations of our software remodularization approach and differentiate it from existing approaches on the software remodularization problem.

The contribution of this paper with respect to the existing approaches on software remodularization of existing package organization (e.g., Abdeen et al., 2009, 2013; Bavota et al., 2014), is that this paper proposes a new remodularization approach to improve the quality of existing package organization using multi-objective evolutionary algorithm i.e., NSGA-II. Our approach uses the design rules of original package structure to control the optimization process instead of penalizing the objective function. For this, we assign the weights to relations on the basis of their intra and inter-package locality in the original package organization. Further, the connection strength between the classes is calculated in terms of weighted relations. The approach improves different package quality criteria such as coupling, cohesion, modularization quality and at the same time it minimizes the movement of classes among existing packages and also partially preserves the original package design principle.

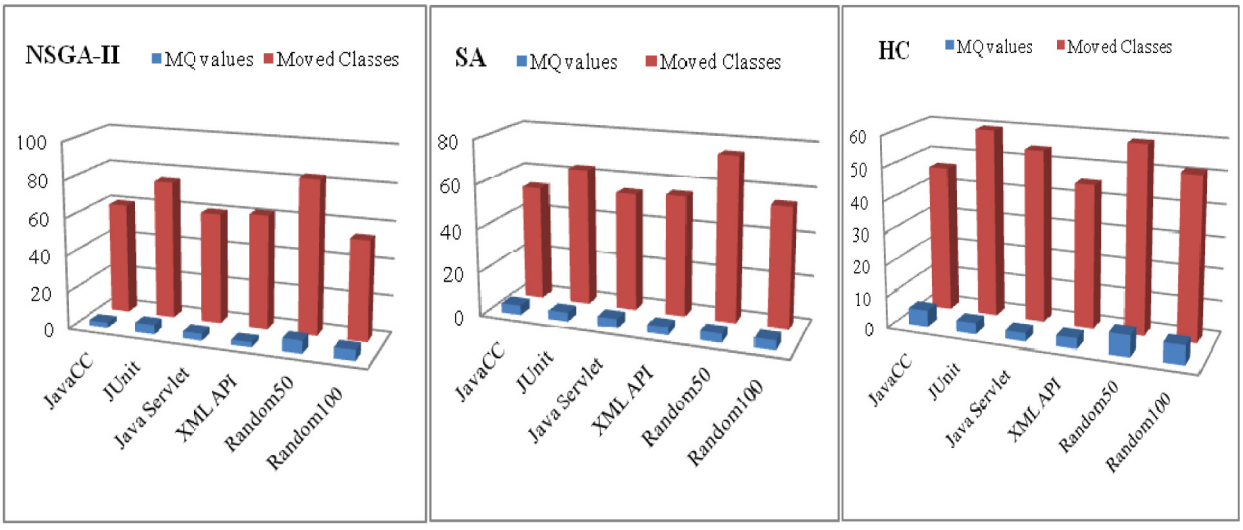

Figure 5 Percentage of reduction in *MQ* values vs. percentage of reduction in moved classes.

The main limitations of our proposed work are: it does not address the problem of decomposing/modularizing software systems from scratch (e.g., multi-objective software module clustering (Praditwong et al., 2011)). Rather, it aims at helping system maintainers in the task of improving the structure of existing software package organization by making as less as possible perturbations in it. Our approach does not consider the hierarchical software package structure and it does not claim that remodularization approach automatically improves existing software package structure with regard to every package design factor.

8. Conclusion and future work

We have presented a multi-objective optimization approach to improve the package structure of an object-oriented software system within existing package organization. The proposed approach takes into consideration eight different types of relations existing with appropriate weights between classes. These relation weights are used to calculate overall connection strength between classes, which is being attempted for the first time in the literature. Another contribution of this paper is that the accurately computed connection strength is used to drive the optimization process toward better quality package organization as well as controlling the excessive movement of classes from originally defined packages. The proposed multi-objective optimization approach has been experimentally evaluated over four real-world and two random software applications. Our approach has been found to perform well and results indicate that the new multi-objective optimization methodology performs better in terms of the software package quality parameters such as modularization merit factors (MMF) and modularization quality (MQ). Future work includes the use of dynamic and semantic relations for connection strength calculation and proposing additional objective functions that can lead the optimization process toward better quality with preservation of the original design to the maximum extent possible.

References

Abdeen, H., Ducasse, S., Sahraoui, H., Alloui, I., 2009. Automatic Package Coupling and Cycle Minimization. In: 16th Working Conference on Reverse Engineering. WCRE '09. pp. 103–112.

Abdeen, H., Sahraoui, H., Shata, O., Anquetil, N., Ducasse, S., 2013. Towards automatically improving package structure while respecting original design decisions. In: 20th Working Conference on Reverse Engineering (WCRE). pp. 212–221.

Abreu, F.B., Goulao, M., 2001. Coupling and cohesion as modularization drivers: are we being over-persuaded. In: Fifth European Conference on Software Maintenance and Reengineering. pp. 47-57.

Anquetil, N., Lethbridge, T.C., 1999. Experiments with clustering as a software modularization method. In: Working Conference on Reverse Engineering. IEEE CS Press, pp. 235–255.

Anquetil, N., Lethbridge, T.C., 2002. Approaches to clustering for program comprehension and remodularization. In: Advances in Software Engineering. Springer, New York.

Barros, M., 2012. An analysis of the effects of composite objectives in multiobjective software module clustering. In: Proceedings of the Fourteenth International Conference on Genetic and Evolutionary GECCC-12. pp. 1205–1212.

Barros, M., Farzat, F.A., Travassos, G.H., 2014. Learning from optimization: a case study with Apache Ant. Inf. Softw. Technol. 57, 684–704.

Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., Lucica, A., 2014. Improving software modularization via automated analysis of latent topics and dependencies. ACM Trans. Software Eng. Method. 23 (1), 213–427.

Bittencourt, R.A., Guerrero, D.D.S., 2009. Comparison of graph clustering algorithms for recovering software architecture module views. In: Software Maintenance and Reengineering. IEEE CS Press, pp. 251–254.

Briand, L., Devanbu, P., Melo, W., 1997. An Investigation into Coupling Measures for C++. In: Proc. 19th Int'l Conf. Software Eng. pp. 412–421.

Cui, J.F., Chae, H.S., 2011. Applying agglomerative hierarchical clustering algorithms to component identification for legacy systems. Inf. Software Technol. 53 (6), 601–614.

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comp. 6 (2), 182–197.

Doval, D., Mancoridis, S., Mitchell, B., 1999. Automatic Clustering of Software Systems using a Genetic Algorithm. In: Proc. Of the International Conference on Software Tools and Engineering Practice (STEP'99).

Eder, J., Kappel, G., Schrefl, M., 1994. Coupling and Cohesion in ObjectOriented Systems. Technical Report, Univ. of Klagenfurt, 1994.

Erdemir, U., Buzluca, F., 2014. A learning-based module extraction method for object-oriented systems. J. Syst. Software 97, 156–177.

Farrugia, A., 2004. Vertex-partitioning into fixed additive induced hereditary properties is np-hard. Electron. J. Comb. 11, 2004.

Forestier, G., Arski, P.G., Wemmert, C., 2000. Collaborative clustering with background knowledge. Data Knowl. Eng. 69 (2), 211–228.

Gui, G., Scott, P.D., 2006. Coupling and cohesion measures for evaluation of component reusability. In: Proceedings of the international workshop on Mining software repositories. ACM, pp. 18–21.

Gupta, V., Chhabra, J.K., 2009. Package coupling measurement in object-oriented software. J. Comput. Sci. Technol. 24 (2), 273–283.

Harman, M., Jones, B.F., 2001. Search based software engineering. Inf. Software Technol. 43 (14), 833–839.

Harman, M., Hierons, R., Proctor, M., 2002. A new representation and crossover operator for search-based optimization of software modularization. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, 9-13. Morgan Kaufmann Publishers, pp. 1351–1358.

Harman, M., Swift, S., Mahdavi, K., 2005. An Empirical Study of the Robustness of two Module Clustering Fitness Functions. In: Proceedings of GECCO'05, Washington DC, USA.

Harris, J., Hirst, J., Mossinghoff, M., 2000. Combinatorics and Graph Theory. Springer, New York, pp. 212–237.

Hitz, Montazeri, 2005. Measuring coupling and cohesion in object-oriented systems. Proc. Int'l Symp. Appl. Corporate 1 (4).

Kashef, R., Kamel, M.S., 2000. Cooperative clustering. J. Pattern Recognit. 43 (6), 2315–2329.

Khan, B., Sohail, S., Javed, M.Y., 2008. Evolution strategy based automated software clustering approach. Adv. Software Eng. Appl. 27, 13–15.

Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P., 1983. Optimization by simulated annealing. Science 220 (4598), 671–680.

Kuhn, A., Ducasse, S., Gírba, T., 2007. Semantic clustering: identifying topics in sourcecode. Inf. Software Technol. 49 (3), 230–243.

Maffort, M., Valente, M.T., Terra, R., Bigonha, M., Anquetil, N., Hora, A., 2015. Mining architectural violations from version history. In: Empirical Software Engineering. Springer, pp. 1–42.

Mahadavi, K., Harman, M., Hierons, R.M., 2003. A multiple hill climbing approach to software module clustering. Proc. IEEE Int'l Conf. Software Maintenance, 315–324.

Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y.F., Gansner, E.R., 1999. Bunch: Recovery and Maintenance of Software System Structures. In: Proceedings of the IEEE International Conference on Software Maintenance. pp. 50–59.

Maqbool, O., Babri, H.A., 2007. Hierarchical clustering for software architecture recovery. IEEE Trans. Software Eng. 33 (11), 759–780.

Márcio, B., Farzat, F.A., Travassos, G.H., 2014. Learning from optimization: a case study with Apache Ant. Inform. Software, Technology.

Mitchell, B.S., 2002. A heuristic approach to solving the software clustering problem (Ph.D. Dissertation). Drexel University, USA.

Mitchell, B.S., Mancoridis, S., 2006. On the automatic modularization of software systems using the bunch tool. IEEE Trans. Software Eng. 32 (3), 193–208.

Patel, C., Hamou-Lhadj, A., Rilling, J., 2009. Software clustering using dynamic analysis and static dependencies. In: Proceedings of the Software Maintenance and Reengineering (CSMR '09), IEEE Computer Society, Kaiserslautern, Germany. pp. 27–36.

Praditwong, K., Harman, M., Yao, X., 2011. Software module clustering as a multi-objective search problem. IEEE Trans. Software Eng. 37 (2), 264–282.

Tonella, P., 2001. Concept analysis for module restructuring. IEEE Trans. Software Eng. 27 (4), 351–363.

Tucker, A., Swift, S., Liu, X., 2001. Grouping multivariate time series via correlation. IEEE Trans. Syst. Man Cybern. B Cybern. 31 (2), 235–245.

Tzerpos, V., Holt, R.C., 1999. MoJo: a distance metric for software clustering. In: Proceedings of the 6th Working Conference on Reverse Engineering, Atlanta, GA, USA. pp. 187–193.

Tzerpos, V., Holt, R.C., 2000. On the stability of software clustering algorithms, International Workshop on Program Comprehension (IWPC 2000), IEEE. pp. 211–218

Wiggerts, T.A., 1997. Using clustering algorithms in legacy systems re-modularization. In: Working Conference on Reverse Engineering, IEEE. pp. 33–43.

Wu, J., Hassan, A.E., Holt, R.C., 2005. Comparison of clustering algorithms in the context of software evolution. In: Proceedings of the 21st IEEE International Conference on Software Maintenance.