Mahouachi R. Search-based cost-effective software remodularization. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(6): 1320–1336 Nov. 2018. DOI 10.1007/s11390-018-1892-6

Search-Based Cost-Effective Software Remodularization

Rim Mahouachi

Complex Outstanding Systems Modeling Optimization and Supervision Laboratory
National School of Computer Science, University of Manouba, 2010 Manouba, Tunisia

E-mail: rim.mahouachi@ensi-uma.tn

Received June 20, 2017; revised August 23, 2018.

Abstract Software modularization is a technique used to divide a software system into independent modules (packages) that are expected to be cohesive and loosely coupled. However, as software systems evolve over time to meet new requirements, their modularizations become complex and gradually loose their quality. Thus, it is challenging to automatically optimize the classes' distribution in packages, also known as remodularization. To alleviate this issue, we introduce a new approach to optimize software modularization by moving classes to more suitable packages. In addition to improving design quality and preserving semantic coherence, our approach takes into consideration the refactoring effort as an objective in itself while optimizing software modularization. We adapt the Elitist Non-dominated Sorting Genetic Algorithm (NSGA-II) of Deb *et al.* to find the best sequence of refactorings that 1) maximize structural quality, 2) maximize semantic cohesiveness of packages (evaluated by a semantic measure based on WordNet), and 3) minimize the refactoring effort. We report the results of an evaluation of our approach using open-source projects, and we show that our proposal is able to produce a coherent and useful sequence of recommended refactorings both in terms of quality metrics and from the developer's points of view.

Keywords remodularization, search-based software engineering, refactoring effort, multi-objective optimization, semantics dependency

1 Introduction

In industry, most object-oriented software systems are large and complex because they contain hundreds of inter-dependent classes. To cope with this complexity, packages are used for organizing a software system into subsystems. We can define packages as software modules used to separate the functionality of a program and to allow developers to group classes together. The original software modularization may be well designed. However, software systems evolve by adding new functions and modifying existing functionalities over time. Through this evolution process, system artifacts (classes, methods, etc.) are added, modified, and removed[1,2]. Therefore, some classes may not be placed in appropriate packages resulting in a bad modularization that is characterized by a large number of inter-dependencies (high coupling) and a small number of intra-dependencies (low cohesion)[3]. Thus, the optimization of class organization into packages to improve software modularization, widely known as remodularization, is required[4]. There has been much research devoted to optimizing software modularization[3,5−18]. Most of them[5−8,16,19] aimed to find the best system decomposition in terms of clusters rather than to improve the existing modularization. Thus, they did not take into account the original modularization and propose a whole new organization of classes in packages (better than the original one according to various coupling and cohesion metrics) which can be difficult to interpret by software developers. In addition, proposing a completely new modularization is useful only when the original system modularization is too degraded to be restructured[4]. In another category of the existing work[9−14], researchers considered the initial structure of a system and optimized the current organization of classes by moving them into other more appropriate packages.

The proposed approach can be classified in the second category of approaches since our objective is to restructure software design by recommending Move Class refactoring operations. In general, the majority of ex-

Regular Paper

isting contributions have formulated the remodularization problem as an optimization problem (mono or multi/many objective) where the aim is to support the principle "packages are designed to be loosely coupled and cohesive to a certain extent"[4]. However, a recent empirical study[20] shows that minimizing coupling and maximizing cohesion are not enough to suggest meaningful remodularization solutions and so other important quality factors (e.g., modules complexity, size) should be taken into account during the optimization procedure. Also, the semantic coherence of the design is not considered by most of the existing work. In fact, the definition of semantic dependency is only limited to the cohesion measures without any consideration of semantic relationships (e.g., the names of packages, classes, methods, attributes and variables). Consequently, the restructured program could improve structural metrics but it becomes semantically incoherent and difficult to understand since classes are moved to other packages based on structural metrics (measuring mainly intra- and inter-package connections) but without any consideration of semantic dependencies. It is important to consider semantic coherence to respect/produce a coherent modularization after applying suggested refactorings. For this, some techniques have been proposed in this context to optimize the distribution of classes using both structural and semantic measures[11,13,21]. Another important issue is the inconsideration of the refactoring effort by the majority of the existing approaches. In fact, in a real-world scenario, developers would have preferred remodularization solutions with small refactoring effort to tools producing "Big-Bang" remodularization (i.e., the number of required changes is in the order of thousands of lines of code)[22]. Since a big-bang remodularization is not a viable solution, it is important to minimize code changes when we optimize software modularization. Indeed, a recent study has shown that "in some specific cases, quite a few refactoring operations could provide a sensible improvement of the modularization quality"[20]. Currently, only few existent approaches take the refactoring effort as an objective while optimizing software modularization. In particular, as stated by a recent empirical study[17], only seven over 31 publications in search based modularization between 1998 and 2017 treated this problem and only five of them[13,17,18,23,24] considered the effort (or disruption) as an objective in their proposed multiobjective optimization techniques.

In this article, we consider automated modularization improvement as a multi-objective optimization problem where the objectives are 1) maximizing the modularization quality, 2) maximizing the semantic cohesion of packages, and 3) minimizing the refactoring effort. The aim of our work is to find the compromise between the mentioned objectives in order to recommend optimal refactoring solutions that maximize/minimize each of them. To find a good modularization, we select and use, from the existing heuristic search algorithms, the NSGA-II algorithm[25]. We decide to use NSGA-II since it is one of the most efficient genetic algorithms proposed in the literature and it has been used successfully in many search-based software remodularization approaches[13,23,26]. The primary contributions of the paper can be summarized as follows.

• We introduce a novel approach for identifying the best refactoring operations sequence that optimizes the original modularization based on NSGA-II. The approach uses cohesion (package internal dependencies) and coupling (package external dependencies) measures to estimate the quality improvement.

• In addition to structural dependencies, we propose to exploit a semantic measurement to identify conceptual dependencies between elements of code.

• We consider the refactoring effort estimated by the rate per refactoring of achieved improvement (RRAI) measurement[23] as an objective to minimize. We note that only few existent approaches take this factor into consideration, and that in the most of them, the number of code changes has been used as an indicator of a refactoring effort.

• We report the results of an evaluation of our approach using 21 releases of four software systems to verify if the refactoring suggestions are able to improve packages' quality and if they are meaningful.

The rest of the paper is organized as follows. Section 2 is dedicated to the background and challenges. Section 3 describes the details of our approach. Section 4 explains the experimental method and its results. Section 5 introduces related work. Section 6 exposes the threats to validity, and the paper is concluded in Section 7.

2 Background and Challenges

2.1 Background

We define the software modularization M as a distribution of the set of object-oriented software classes $C = \{c_i, i \in [1..|C|]\}$ into a set of packages $P = \{p_i, i \in [1..|P|]\}$. We distinguish two main types of dependencies between packages:

• package external dependencies (inter-edges dependencies) related to classes belonging to different packages: $ED_p(p_i, p_j) = \sum_{l=1}^{|p_i|} \sum_{s=1}^{|p_j|} ED_p(c_l, c_s)$, where $c_l \in p_i$, $c_s \in p_j$, and $p_i, p_j \in P$;

• package internal dependencies (intra-edges dependencies) related to classes in the same package: $ID_p(p_i) = \sum_{l=1}^{|p_i|} \sum_{s=1}^{|p_i|} ID_p(c_l, c_s)$, where c_l, $c_s \in p_i$, and $p_i \in P$.

A dependency can be a method, a field or a class access. Fig.1 shows an example of software modularization. It consists of the distribution of six classes over three packages. We define a package size by the number of its classes.

As showed in Fig.1, there are four external dependencies ($ED_p(c_1, c_4)$, $ED_p(c_4, c_5)$, $ED_p(c_5, c_3)$, and $ED_p(c_2, c_5)$), and five internal dependencies ($ID_p(c_1, c_2)$, $ID_p(c_2, c_3)$, $ID_p(c_3, c_1)$, $ID_p(c_3, c_2)$, and $ID_p(c_6, c_5)$).

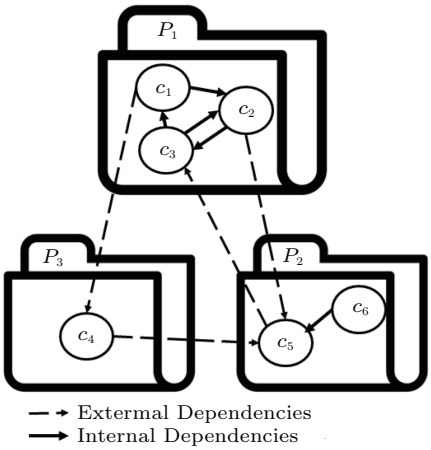

Fig.1. Modularization example.

2.2 Challenges in Optimizing Software Modularization

In this subsection, we emphasize on the specific problems that are addressed by our approach. We can summarize the principal remodularization issues as follows.

• Classes are not well distributed into packages. In addition, the coupling between classes from different packages increases the coupling between these packages. Therefore, most of the packages are dependent on some dominant ones (i.e., packages that contain a big number of classes).

• Optimizing a quality modularisation can produce semantic incoherencies: it is necessary to consider both structural and semantic dependencies[11,13,21].

• The vast majority of existing studies ignore the refactoring effort and produce a big-bang remodularization that implies developers to change thousands of lines of code, even for modest systems. In fact, Hall *et al.*[22] estimated that developers would have to change up to 10% of their code when they adapted solutions proposed by automated remodularization approaches.

To address these challenges, we propose a multi-objective approach in order to optimize the software modularization quality with respect to semantic constraints. The proposed approach also aims to minimize the modification degree in produced solutions with regard to achieved improvements. We describe in Section 3 how to consider the optimization of modularization as an optimization problem using a quality model evaluation.

3 Multi-Objective Optimization Approach for Software Remodularization

This section shows how the above-mentioned issues can be treated and describes the principles that underlie the proposed method to improve the software modularization quality. Therefore, we first present an overview of the approach and subsequently provide the details of our adaptation of NSGA-II to suggest refactorings.

3.1 Approach Overview

In this paper, an automated approach is proposed to optimize the original software modularization based on some quality attributes. The general structure of our approach is introduced in Fig.2. The approach takes as input a code to re-modularize. It generates as output the best combination of refactoring operations (limited to Move Class operations) that improves software modularization quality (evaluated by quality evaluator (B)), preserves semantic domain (evaluated by semantic evaluator (A)), and minimizes refactoring effort (evaluated by RRAI (C)). In this case, a solution is defined as the sequence of refactoring operations that find a compromise between the three objectives. We use Soot[27] (a Java framework to analyze, instrument, optimize, and visualize Java applications) in order to parse and extract from the original source code the relevant code elements (i.e., packages, classes, methods, attributes, etc.) and the existing relationships among them. This tool generates a parsed code in a specific representation and a call graph that will be used for calculating semantic constraints and quality metrics.

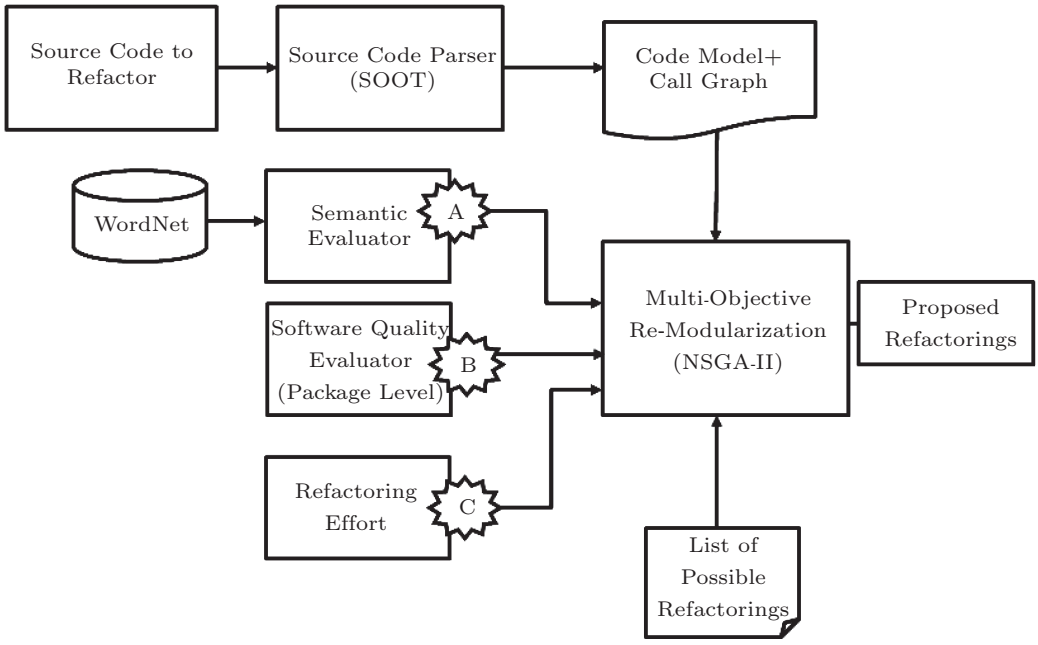

Fig.2. Overview of proposed approach.

According to Mitchell and Mancoridis[6], the software remodularization problem is a graph partitioning problem, which is known to be an NP-hard problem[28]. Due to the large number of possible refactoring combinations, an enumerative approach is infeasible; thus a heuristic method should be used to explore the space of possible solutions. To this end, we adapt and use the Elitist Non-dominated Sorting Genetic Algorithm (NSGA-II)[25] to perform a global heuristic search. This algorithm and its adaptation to the remodularization problem are described in Subsection 3.2.

3.1.1 Semantic Evaluator (A)

1) *Main Idea.* In order to quantify semantic cohesiveness, we consider classes, methods, and fields' names as class/package vocabulary, and we calculate the semantic similarity between two vocabularies using a similarity function based on a variety of Jiang and Conrath distance[29] relying on WordNet information content (IC). In fact, two software packages can be semantically related if they implement a similar/common vocabulary (names of methods, fields, variables, parameters, declaration types, etc.). Indeed, the name of a variable, method, or class, reveals the developer intent and it corresponds on internal indicators for the meaning of a program[30]. Many authors focused their attention on source code identifiers and showed their importance for various tasks in software comprehension[31−37].

Thus, when a class has to be moved from one package to another, using the refactorings "Move Class", the target package should be appropriate to contain the selected class[4]. Consequently, the refactoring may make sense if both target package and source class implement a similar vocabulary.

2) *Identifier Extracting and Splitting.* We extract the data required to compute the semantic similarity in three steps.

• *Data Extracting.* First, we extract the identifiers (names) found in classes (e.g., names of attributes and methods, user defined types) and packages (e.g., classes, names of attributes and methods) concerned by the refactoring operation. Extracted identifiers (e.g., causesIllegalSymlinkLoop) are composed of terms (e.g., causes, Illegal, Symlink, and Loop).

• *Identifier Splitting.* This step aims at splitting identifiers into their composing terms. We use a CamelCase splitter to build the term dictionary. The output of this phase is the lists of terms composing each name. For example, the identifier addDefaultExcludes is split into {add, Default, and Excludes}.

• *Term Filtering.* This step consists in removing common terms (e.g., words shorter than three characters and stop words in Java).

3) *Semantic Similarity Evaluation.* For our proposal, we use the similarity measure of Jiang and Conrath[29] to estimate the semantic similarity between code elements (in our case classes and packages). We

choose, in particular, the similarity measure of Jiang and Conrath[29] redefined by Seco *et al.*[38] to calculate the semantic similarity of refactorings in order to evaluate the given solution. Indeed, according to several similarity measures evaluations[29,39,40], the measure of Jiang and Conrath[29] obtained the best results correlation into human judgments. This measure is based on information content (IC) defined by Resnik[41] and calculated by combining knowledge of a hierarchical structure like WordNet with statistics on a large corpus. Seco *et al.*[38] expressed the IC value using only WordNet without the need for external corpora. They reported that similarity measures using this IC correlate more closely with human assessments than classic measures of IC based on corpus analysis. In this work, we use the similarity function of Jiang and Conrath[29] transformed by Seco *et al.*[38] defined as follows.

$$sim_{jcn}(c_1, c_2) = 1 - \left(\frac{ic_{wn}(c_1) + ic_{wn}(c_2) - 2 \times sim_{res'}(c_1, c_2)}{2}\right), \quad (1)$$

where:

• c_1 and c_2 are two concepts (i.e., senses or synsets in WordNet),

• $sim_{res'}(c_1, c_2) = \max_{c \in S(c_1, c_2)} ic_{wn}(c)$, where $S(c_1, c_2)$ are the set of concepts that subsume c_1 and c_2, and

• $ic_{wn}(c)$ is the information content (IC) in WordNet defined by Seco *et al.*[38] as a function of the hyponyms where the concept c has:

$$ic_{wn}(c) = \frac{\log(\frac{hypo(c)+1}{max_{wn}})}{\log(\frac{1}{max_{wn}})} = 1 - \frac{\log(hypo(c)+1)}{\log(max_{wn})}.$$

The function *hypo* returns the number of hyponyms of a given concept, and max_{wn} is a constant that is set to the maximum number of concepts that exist in the taxonomy[38]. Note that a hyponym is a word whose semantic field is included within that of another word. For example the word car is the hyponym of the word vehicle.

4) *Illustrated Example.* To illustrate the use of similarity measurement in our refactoring context, let us take the example of "Move Class". This refactoring has to be applied when a package is more appropriate to contain a class than the one containing it.

For instance, let us take an example of a Move Class refactoring candidate in JHotdraw 5.2.

RO_i: **MoveClass**
Class to move: PaletteButton
From: CH.ifa.draw.util
To: CH.ifa.draw.framework

The output of the textual information (terms) extracting and splitting phases is the following three vocabularies having sizes n, m and p respectively:

• $Vocabulary1 = \{term"_j, j \in [1, n]\}$ corresponding to *CH.ifa.draw.util* vocabulary,

• $Vocabulary2 = \{term'_k, k \in [1, m]\}$ corresponding to *CH.ifa.draw.framework* vocbulary, and

• $Vocabulary3 = \{term_l, l \in [1, p]\}$ corresponding to *PaletteButton* vocabulary.

As showed in Fig.3, considering the source code of *PaletteButton*, identifiers are extracted to compose *Vocabulary3*. On the other side, *Vocabulary2* contains the set of terms {*Tool, DrawingView, setTool, selectionZOrdered, selectionCount, addToSelection, DrawingEditor, ...*}.

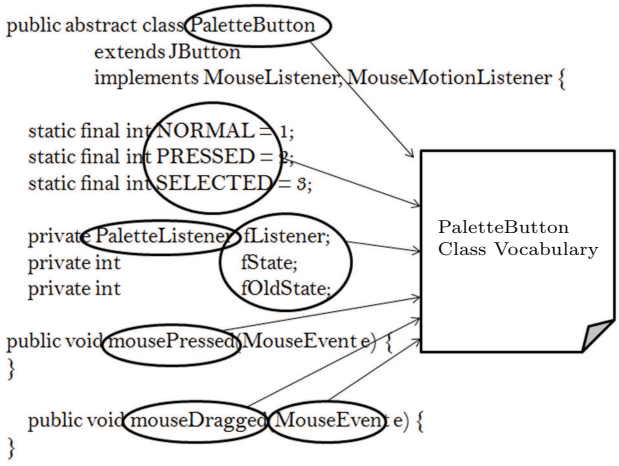

Fig.3. Source code snippet example.

The target package (*CH.ifa.draw.framework*) is considered as more appropriate to contain the class under analysis (*PaletteButton*) if the target package is closer related to it conceptually (i.e., if *Vocabulary3* indicates similar domain semantics) than its original package (*CH.ifa.draw.util*).

We calculate the semantic similarity $sem\_sim(RO_i)$ as follows.

$$sem\_sim(RO_i) = \frac{1}{p}\sum_{l=1}^{p}\left(\frac{1}{m}\sum_{k=1}^{m} sim(term_l, term'_k) - \frac{1}{n}\sum_{j=1}^{n} sim(term_l, term"_j)\right),$$

where:

$$sim(term_1, term_2) = \begin{cases} 1, & \text{if } term_1 = term_2, \\ sim_{jcn}(term_1, term_2), & \text{otherwise.} \end{cases}$$

For instance, the similarity between the two terms $\{term_1 = mousePressed,\ term_2 = setTool\}$ is equal to 0.518 according to the values obtained in Table 1.

Table 1. Semantic Similarity Calculation Example

Term	Set	Tool
Mouse	0.367	0.496
Press	0.644	0.610

3.1.2 Quality Evaluator (B)

Modularization quality is measured using cohesion and coupling metrics. The package cohesion consists of the relatedness among classes of a package (i.e., the number of intra-edges dependencies of a package). The package coupling consists of the number of classes (from other packages) that a class is directly related to.

3.1.3 Refactoring Effort Evaluator (C)

In order to assess refactoring effort, we choose to use the RRAI (rate of achieved improvement) measurement calculating the modification degree related to achieved improvements and defined by Abdeen *et al.*[23] as follows.

$RRAI(m) = \frac{RPMC(m)}{RPC(m)}$, and in our context $m \in$ {package coupling, package cohesion}.

- $RPMC(m) = \frac{\delta m}{|C_{\text{move}}|}$, where δm is the increased (decreased) value of m in the new resultant modularization and $|C_{\text{move}}|$ is the number of classes to move. The $RPMC(m)$ value consists of the average contribution of each moved class to the achieved improvement to m.
- $RPC(m) = \frac{m_{\text{or}}}{|C|}$, where m_{or} is the value of m in the original modularization, and $|C|$ is the number of all classes in the system. The $RPC(m)$ value consists of the average contribution of all classes to the value of m in the original modularization.

For example, if the proposed solution recommends to move 20 classes and improve the structural cohesion by 0.5 (the value is normalized), we obtain $RPMC(cohesion) = \frac{\delta cohesion}{|C_{\text{move}}|} = \frac{0.5}{20} = 0.025$. Let us suppose further that $RPC(cohesion) = \frac{coh_{\text{or}}}{|C|} = 0.01$. In this case, $RRAI(cohesion) = \frac{RPMC(cohesion)}{RPC(cohesion)} = 2.5$ which is bigger than 1. In other words, we can say that $RPMC(cohesion) > RPC(cohesion)$ which means that if they are moved to the determined packages, the 20 selected classes have an average contribution to the cohesion improvement in the resultant modularization better than the average contribution of all classes to the value of cohesion in the original modularization.

3.2 NSGA-II Algorithm Adaptation

NSGA-II (Elitist Non-dominated Sorting Genetic Algorithm)[25] is an algorithm designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A feasible solution is non-dominated when there is no other feasible solution better than the current one in terms of some objective functions without worsening other objective functions.

A high level view of NSGA-II adapted to the remodularization problem using structural and semantic information is described in this subsection. The algorithm takes as input a code to re-modularize. It starts by creating a random population P_0 of individuals. Then, a child population Q_0 is generated from the population of parents P_0 using selection and change operators (crossover and mutation). Both populations are merged into an initial population R_0 of size Max_size, and a subset of individuals is selected, based on the dominance principle (RRAI, structural and semantic modularization quality improvement are the objectives) and crowding distance (for solutions having the same dominance) to create the next generation. This process will be repeated it_Max iterations. The output of the algorithm is the best combination of refactorings that improve the software modularization quality (evaluated by structural and semantic metrics) with regard to their effort.

The following three subsections describe more precisely our adaptation of NSGA-II to the remodularization problem.

3.2.1 Solution Representation

We consider the potential solution as a vector of refactoring operations, in other words, a set of ordered operations. The order of applying a combination of refactorings composing a potential solution corresponds to their vector position. In this work, only the refactoring "Move Class (source class, source package, target package)" is considered, but other types of refactorings can be supported in future work. This refactoring consists of moving a source class from the source package to a more appropriate target package. A possible 3-sized solution could be, for example:

1) Move Class (org.w3c.HTMLDocument, org.w3c, simpletype),

2) Move Class (org.apache.xerces.ASModel, org.apache.xerces, xni),

3) Move Class (org.w3c.dom.ls.DOMBuilder, org.w3c.dom.ls, sax).

The execution of these refactorings must conform to certain semantics pre- and post-conditions (to avoid conflicts and incoherence semantic errors). For example, let us consider the following remodularization operation.

RO :	**MoveClass**
Class to move :	*XSGroupDecl*
From :	*xs*
To :	*xpath*

To apply this refactoring, a number of necessary preconditions should be satisfied:

• *xs* and *xpath* should exist and should be packages,

• *XSGroupDecl* should exist and should be a class, and

• the class *XSGroupDecl* should be implemented in the package *xs*.

Postconditions are as follows:

• *XSGroupDecl*, *xs* and *xpath* should exist,

• *XSGroupDecl* class should be in the package *xpath* and should not exist any longer in the package *xs*.

To generate an initial population, we start by fixing the minimum and the maximum vector length including the number of refactorings. Thus, the individuals have different vector lengths (structures) since lower and upper bounds of the chromosome length depend on the studied system and are fixed experimentally. Then, for each individual we randomly assign one refactoring, with its parameters, to each dimension.

After applying the proposed refactorings to the initial remodularization, we obtain a new modularization that will be evaluated using the fitness functions.

3.2.2 Fitness Functions

The objectives of our optimization approach are 1) maximizing modularization quality, 2) maximizing packages semantic cohesion, and 3) minimizing refactoring effort.

• *Quality Objective.* To evaluate the impact of applying the refactorings on the modularization quality of the refactored system, we define the quality gain QG ($\in [0..1]$) as follows.

$$QG = \frac{1}{2}(v(i)_{\text{after refactoring}} - v(i)_{\text{before refactoring}}),$$

where $v(i)$ is the normalized value of quality attribute i and $i \in \{\text{Cohesion, Coupling}\}$.

After applying a generated solution (the proposed refactorings) to the initial software system, this fitness function calculates the quality of the resultant modularization.

• *Semantic Coherence Objective.* In our case, we consider the semantic similarity of a solution ($\in [-1..1]$) as the average of semantic similarity scores of refactoring operations that constitute it.

$$sem\_sim(sol) = \frac{1}{\mid sol \mid} \sum_{i=1}^{|sol|} (sem\_sim(RO_i)).$$

• *Refactoring Effort.* To assess the refactoring effort we consider the arithmetic mean, ($\overline{RRAI}$), of RRAI values (for every quality attribute improvement) in NSGA-II solutions. When the rate average of the achieved improvement $\overline{RRAI}$ is larger than 1, it indicates that the suggested modifications have a considerable impact on quality improvement. Therefore, the objective is to maximize $\overline{RRAI}$.

• *Normalization.* Since objective functions have different scales, we use the normalization procedure proposed by Deb and Jain[42] to normalize them. For that, the minimal and the maximal values for each objective function value are recorded and then used by the normalization procedure.

3.2.3 Operators

Selection. To guide the selection process, NSGA-II sorts the population using the dominance rank and the crowding distance rank[25] to select individuals of the new population P_{t+1}. Then, to generate an offspring population Q_{t+1}, the selection of individuals to which genetic operators (crossover and mutation) are applied is based on the tournament selection operator.

Crossover. We use the one-point crossover operator. For each crossover, two parents are selected randomly. The crossover operator allows creating two offspring P'1 and P'2 from the two selected parents P1 and P2. First, a random position k is selected, then the first k refactorings of P1 becomes the first k elements of P'1, and the first k refactorings of P2 become the first k refactorings of P'2.

Mutation. The mutation operator consists in randomly changing one or more dimensions (refactoring) in the solution. Given a selected individual, the mutation operator first randomly selects some dimensions in the vector representation of the individual. Then the

selected dimensions are replaced by other refactorings generated randomly.

4 Validation

To evaluate the feasibility of our approach, we conduct an experiment with different open-source projects. We start by presenting our research questions and our settings. Then, we describe and discuss the obtained results.

4.1 Goals and Objectives

Our study involves five research questions, which we define here, explaining how our experiments are designed to address them. The goal of the study is to evaluate the efficiency of our approach for the refactoring recommendation from the perspective of a software maintainer conducting a quality audit. We present the results of the experiment that aims at answering the following research questions.

• $RQ1$. To what extent can the proposed approach improve the structural quality of packages in the studied software systems?

• $RQ2$. To what extent can the proposed approach minimize the refactoring effort?

• $RQ3$. To what extent can the proposed approach preserve the semantics while improving the packages structure?

• $RQ4$. How does the proposed approach perform compared with other existing search-based remodularization approaches?

• $RQ5$. To what extent can the consideration of refactoring effort improve the effectiveness of the proposed refactorings?

To answer RQ1, we evaluate the structural quality improvements of systems after applying the best solution using a set of structural metrics:

• $QG = \frac{1}{2}(v(i)_{\text{after refactoring}} - v(i)_{\text{before refactoring}})$,

• total number of external dependencies ($TED_p = \sum_{i=1}^{|P|} ED_p(p_i)$),

• total number of internal dependencies ($TID_p = \sum_{i=1}^{|P|} ID_p(p_i)$),

• the structural coupling between packages:

$$StC = StructuralCoupling(p_i, p_j) = \frac{\sum_{l=1}^{|p_i|}\sum_{s=1}^{|p_j|} MPC(c_l, c_s)}{|p_i| \times |p_j|},$$

where $c_l \in p_i, c_s \in p_j$, and $MPC(c_l, c_s)$ is the message passing coupling between c_l and c_s which is directly correlated with the maintenance effort[12].

To answer RQ2, we analyzed the $\overline{RRAI}$ results obtained for each system. To evaluate the effort of each recommended remodularization solution, we used also the *MoJoFM* distance[43]. This metric calculates the number of "move" and "join" operations, needed to transform one partition A into another one B. In this paper, a Move operation represents moving a class from its original package to another package, when the partition A represents the original modularization, and B the obtained modularization after applying the set of recommended refactorings. Therefore, the *MoJoFM* formula is:

$$MoJoFM(A, B) = (1 - \frac{mno(A, B)}{\max(mno(\forall A, B)}) \times 100\%,$$

where:

• $mno(A, B)$ is the minimal number of "move" and "join" operations needed to transform A in B, and

• $\max(mno(\forall A, B))$ is the maximum number of possible "move" and "join" operations to transform any partition A into B.

To answer RQ3, six Ph.D. students verified manually the feasibility and meaningfulness (having semantic sense) of the proposed refactoring solutions. All students have significant experience in Java programming (ranging from 5 to 10 years). They are members of the SBSE (Search-Based Software Engineering) Lab and students at the University of Michigan (USA), University of Montreal (Canada), and Missouri University of Science and Technology (USA). The students are unaware of the proposed technique (but of course know that they are going to evaluate the semantic coherence of refactoring operations) in order to guarantee that there will be no bias in their judgment.

Each subject received 1) the source code of four selected releases of studied object systems, and 2) the questionnaire. For each recommended refactoring operation, participants had to answer to the question: "would you apply the proposed refactoring?" The answer is one of these three possibilities: yes, no, and maybe. Since the application of remodularization solutions is a subjective process, it is normal that not all the programmers have the same opinion. In our case, we consider the majority of votes to determine if suggested solutions are correct or not.

We report the percentage of proposed refactorings that were semantically correct (ratio of the number of

coherent refactoring operations over the number of evaluated refactoring operations) which we note the refactoring semantic precision ($\in [0..1]$) and define as follows:

$$RP = \frac{\text{number of coherent refactorings}}{\text{number of proposed refactorings}}.$$

To answer RQ4, we compare our results with those produced by two other search-based engineering approaches: the one of Abdeen *et al.*[23] whose objective is to automatically reduce package coupling and cycles by moving classes over the existing packages, and that of Mkaouer *et al.*[13], a many-objective search-based approach using NSGA-III to find refactorings that improve packages structure, the semantic coherence of the restructured program, and minimize the number of changes.

To answer RQ5, we compare our results of with and without considering refactoring effort as an objective in our algorithm.

4.2 Systems Studied

We used a corpus of 21 releases of four open source Java projects, namely Apache Ant①, JHotDraw②, JFreeChart③, and Xerces-J④. Apache Ant is a tool and library specifically conceived for Java applications. Xerces-J is a family of software packages for parsing and manipulating XML, and implements a number of standard APIs for XML parsing. JHotdraw is a framework used to build graphic editors. JFreeChart is a powerful and flexible Java library for generating charts. Table 2 reports characteristics of the analyzed systems. We selected Xerces-J and Apache Ant because they are medium-sized open-source projects and are analyzed in related work. The initial versions of Apache Ant are known to be of poor quality, which has led to major revised versions.

Table 2. Systems Studied

System	Number of Releases	Number of Classes [Min, Max]	Number of Packages [Min, Max]
Xerces-J	14	[560, 969]	[34, 57]
Apache Ant	1	[384, 384]	[18, 18]
JHotdraw	3	[173, 585]	[11, 24]
JFreeChart	3	[646, 696]	[30, 32]

4.3 Algorithm Settings

Parameters tuning influences significantly the performance of a search algorithm on a particular problem. However, there is no generic best configuration which can be adopted in all situations; in contrast, it is different from one search problem to another. For example, a larger number of operations in a solution do not necessarily mean that the results will be better. For this reason, for each system, we performed a set of experiments using several configurations (varying population size, and the maximum of chromosome length) and we selected the best one (i.e., producing the best results). Ideally, a small number of operations should be sufficient to provide a good trade-off between the objectives. In a solution, the maximum number of refactorings can also be specified by the user. In this case, the user can select a smaller number of changes, if he/she wants to make minor revisions or a larger number of changes, or if he/she wants to make major revisions in his/her code. In our experiments, we chose these setting parameters: population size = 100, termination criterion = 10 000, crossover probability = 0.7, mutation probability = 0.3, and individual size = 50. Another important point to clarify is that our approach is stochastic by nature which means that two different executions on the same system and with the same configuration generally lead to different sets of suggested refactorings. Therefore, in order to confirm the validity of the obtained results, our experimental study is performed based on 31 independent simulation runs for each problem instance. In this paper, we consider software remodularization as a multiobjective optimization problem and we adopt NSGA-II. Hence, the output is a set of non-dominated solutions. The developers can choose one solution from the set of Pareto depending on their preferences in terms of compromise. For example, a developer can prefer a solution producing the best score of quality improvement but needing much more maintenance effort than the others. However, for the validation of our approach, we need to select only one solution from the set of Pareto. Since the ideal solution consists in the one with the best value of quality improvement in terms of cohesion and coupling (equal to 1), design semantic coherence (equal to 1), and refactoring effort (normalized value equal to 1), we selected the

①http://ant.apache.org, Aug. 2018.

②http://www.jhotdraw.org, Aug. 2018.

③http://www.jfree.org/jfreechart/, Aug. 2018.

④http://xerces.apache.org, Aug. 2018.

nearest solution to the ideal one in terms of Euclidian distance.

4.4 Results and Discussions

In this subsection, we present the answer to each research question in turn, indicating how the results answer them.

4.4.1 Results for RQ1

Table 3 shows the changes in terms of the structural coupling (StC) achieved while applying Move Class operations suggested by NSGA-II. We also report the deviation (delta with the initial design) median values of internal and external packages dependencies obtained after applying refactoring operations suggested by NSGA-II.

Table 3. δStC, δTED_p, and δTID_p Median Values of 31 Independent Runs of NSGA-II

System	Release	δStC (%)	δTED_p	δTID_p
JFreeChart	0.9.12	−6	−12	+7
	0.9.13	−13	−4	+22
	1.0.9	−5	−9	+18
Apache Ant	1.8.4	−19	−21	+143
JHotDraw	5.1	−13	−8	+69
	5.2	−21	−18	+149
	6.1	−24	−7	+131
Xerces	2.9.0	−17	−21	+100
	2.8.1	−29	−37	+101
	2.8.0	−25	−25	+97
	2.7.1	−22	−33	+126
	2.7.0	−18	−26	+174
	2.6.2	−28	−12	+180
	2.6.1	−32	−14	+183
	2.6.0	−18	−10	+44
	2.5.0	−15	−5	+141
	2.4.0	−21	−52	+159
	2.3.0	−11	−17	+100
	2.2.1	−29	−14	+148
	1.2.1	−60	−4	+102
	1.2.0	−60	−13	+68

As shown in Table 3, we note that our approach improves all evaluated structural metrics. In Xerces v2.4.0, for example, the refactored solution has a cohesion value (TID_p) of 268 and a coupling value (TED_p) of 202 while the original developers' implementation has a cohesion value of 427 and a coupling value of 254, which corresponds to an improvement of +159 and −52 respectively.

Also, both structural cohesion (TID_p) and coupling (TED_p) of packages for all studied systems are strongly improved. Our approach succeeds to minimize the maintenance effort required by developers ($\delta StructuralCoupling_{avg}$) by 24% for all systems. In conclusion, the obtained results are very convincing, and the resultant modularizations are clearly better than the original ones according to obtained quality metrics values. Therefore, we can conclude that the proposed approach can find solutions that improve the modularity of the software system, as assessed by cohesion and coupling.

4.4.2 Results for RQ2

To answer RQ2, we evaluated the effort needed to apply the suggested remodularization solutions. We believe that optimizing the number of required changes is a difficult and very important objective to reach. In our approach, we considered RRAI, the modification degree related to achieved improvements, in order to minimize this number. Considered as an objective to maximize, this measure means that a set of refactorings is preferred rather than another one if the number of classes to move and the improvement achieved are better. For example, a solution that recommends moving 20 classes and improves the structure quality by 10% is better than another one moving 30 classes to improve the quality by 12%. Table 4 presents the ($\overline{RRAI}$) obtained on different systems. We find that our approach succeeds in suggesting solutions that improve considerably structural packages quality and do not require high code changes to achieve it.

In fact, NSGA-II moved in average 30 classes (which corresponds to less than 1% of total classes' number) for JfreeChart achieving a consistent improvement in terms of quality with an average QG score of 0.59. For JHotdraw 5.3, moving only 16 classes (less than 6% of the total number of classes) allowed to achieve an average of QG score of 0.68 in terms of quality gain.

In addition, our experiments show that the arithmetic mean, ($\overline{RRAI}$), of RRAI values in NSGA-II solutions is larger than 1 for all studied systems. This indicates that the suggested modifications have a considerable impact on quality improvement. For all systems, the rate average of the achieved improvement values is about 21.62.

In addition, we report in Table 4 the obtained *MoJoFM* values. For instance, for Xerces v2.9.0, it is possible to have a modularization improvement (36%) even considering a lower modularization effort (3.31%). We

Table 4. Quality Improvement vs Needed Modifications

System	Release	QG	Number of Moved Classes	$\overline{RRAI}$	$MoJoFM(\%)$
JFreeChart	0.9.12	0.49	27	1.59	95.33
	0.9.13	0.44	28	21.17	95.63
	1.0.9	0.59	30	40.74	95.39
Apache Ant	1.8.4	0.43	25	41.87	93.17
JHotDraw	5.1	0.39	12	13.34	91.98
	5.2	0.48	13	8.77	92.78
	6.1	0.68	16	112.69	94.27
Xerces	2.9.0	0.36	45	10.39	96.69
	2.8.1	0.53	43	23.48	97.63
	2.8.0	0.29	44	10.04	97.73
	2.7.1	0.57	48	19.80	94.67
	2.7.0	0.68	46	32.90	96.95
	2.6.2	0.59	49	19.25	96.65
	2.6.1	0.59	45	22.63	96.10
	2.6.0	0.24	46	6.54	94.21
	2.5.0	0.11	47	4.24	96.95
	2.4.0	0.25	44	12.54	95.21
	2.3.0	0.34	46	13.60	95.78
	2.2.1	0.36	47	14.86	94.88
	1.2.1	0.39	31	20.31	94.01
	1.2.0	0.26	31	21.98	94.22

can cite also the example of JfreeChart v0.9.12, where it is possible to improve the structural quality by 49% by changing only 4.67% of the original modularization.

To conclude, for all considered systems, *MoJoFM* is about 95.2%, which means that the developers need to change, in average, 4.8% of the original implementation if they consider the remodularization solution. We think this is a very interesting result since the main goal of this work is to improve the structure of packages while minimizing the effort needed to apply the suggested remodularization solutions. Thus, we believe that our proposal can be used by developers at different moments during the software lifecycle when they are planning a major release of the software system as well as a minor or bug-fixing release. According to their needs, they have to choose, from the set of Pareto, the modularization solution which best fits their expectations.

4.4.3 Results for RQ3

To answer RQ3, we report results of the manual validation elaborated by a group of potential users of our remodularization tool. The ratio of meaningful operations, in terms of semantic coherence from the suggested ones, is described in Table 5. For Xerces, 77% of the set of the evaluated refactorings has been appreciated by participants (29 out of 48 answered yes), 17% (8 out of 48) has been considered as probably coherent, and 23% (11 out of 48) has been rejected. For JfreeChart, 19 refactoring operations have been considered to be absolutely or probably coherent when only eight out of 27 have been completely rejected by most of subjects. To conclude, most of the suggested operations are feasible and make sense semantically from the participants' point of view.

Table 5. Collected Results for Refactoring Meaningfulness

Studied System	Release	Yes (%)	No (%)	Maybe (%)	RP (%)
Xerces-J	v2.7.0	60	23	17	77
Apache Ant	v1.8.4	48	20	16	64
JHotDraw	v6.1	50	25	20	70
JFreeChart	v1.0.9	63	28	9	72

To conclude, almost 71% of the proposed refactorings are semantically feasible and do not affect the semantic coherence of the refactored program from the point of view of potential users.

4.4.4 Results for RQ4

Since in this paper we are limited to Move Class refactoring while Mkaouer *et al.*[13] considered eight other refactoring types, we do not compare the number of code changes suggested by each approach. We limit our comparison to the structural cohesion and coupling obtained by the best proposed remodularization solutions for three different systems. As described in Fig.4, we find that our proposal provides, in average, similar structural improvements to the other techniques

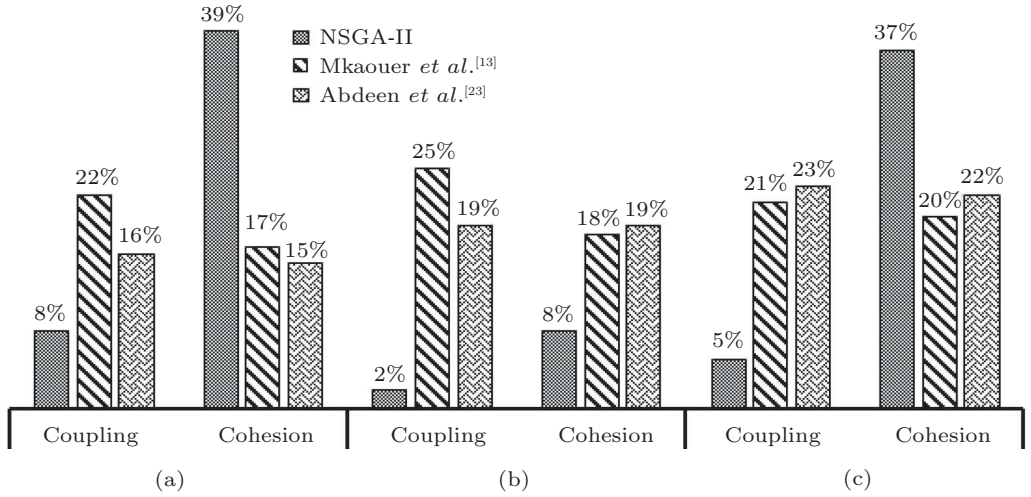

Fig.4. Coupling and cohesion improvement median values of NSGA-II, NSGA-III by Mkaouer *et al.*[13] and NSGA-II by Abdeen *et al.*[23] over 31 independent simulation runs. (a) Xerces v2.7.0. (b) JFreeChart v1.0.9. (c) JHotDraw v6.1.

(Mkaouer *et al.*[13] and Abdeen *et al.*[23]) in terms of modularization cohesion and coupling. We can consider that the NSGA-II performance in terms of improving the distribution of classes into packages is similar to that of the existing approaches.

4.4.5 *Results for RQ5*

To investigate the effect of the consideration of the refactoring effort, we compared the obtained results of with and without considering RRAI as an objective in NSGA-II. According to Fig.5, we note that in most of the systems when the refactoring effort is combined with other objectives, our approach succeeds in recommending remodularization solutions that maximize the rate average of achieved improvement. In fact, approximately, the same number of the refactoring operations is proposed but with a relatively stable and better quality improvement than the refactorings proposed by the algorithm without considering the effort. For example, for Apache Ant, δTID_p is $+48$ and δTED_p is -2 when only quality and semantics are considered; however, when the refactoring effort is included, δTID_p is improved to $+143$ and δTED_p is -21 with approximately the same number of proposed refactorings (about 25 refactoring operations).

5 Related Work

There are several studies that have recently focused on optimizing modularization in software using different techniques. These techniques range from clustering to evolutionary or search-based algorithms. We can distinguish two categories of approaches: those considering the original modularization and existing packages, and those not considering them.

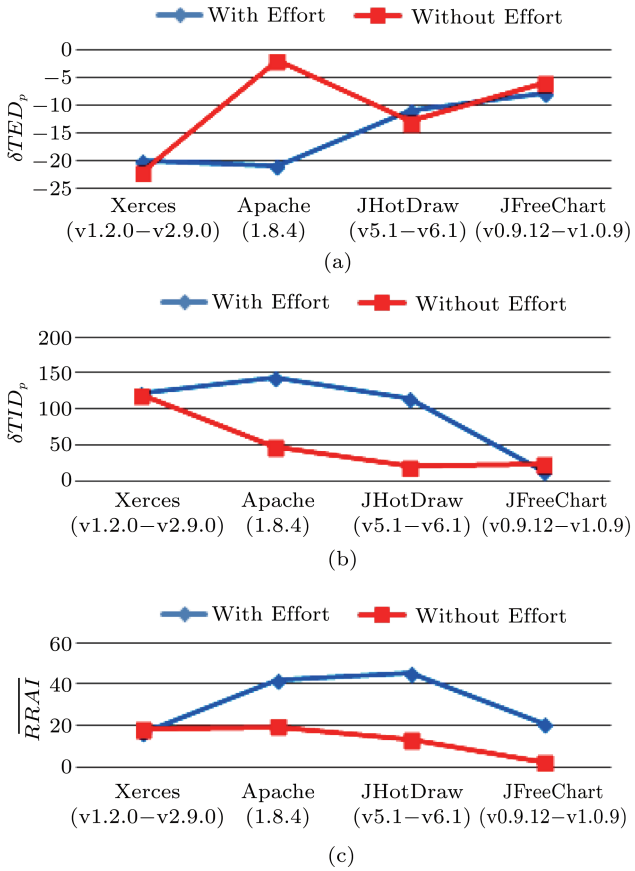

Fig.5. Remodularization results of NSGA-II with and without effort in terms of (a) coupling improvement (δTED_p), (b) cohesion improvement (δTID_p), and (c) $\overline{RRAI}$.

5.1 Remodularization Approaches Without Consideration of the Original Modularization

In this category of approaches, the original modularization is not considered, and the aim is to find the best modularization according to the cohesion and coupling principles. These approaches produce a new modularization completely different from the original with new classes' distribution into clusters (new packages). Mitchell and Mancoridis[6] proposed the first search-based approach to address the problem of software modularization using a single-objective function. Their approach consists on a search-based clustering technique using hill-climbing to optimize clusters organization. The input of the clustering algorithm is a system that is represented by MDG as a (weighted or unweighted) graph where nodes represent entities to be clustered and edges the relationships among them. The output is a partition of this graph that is composed of clusters of nodes having high cohesion and low coupling. The algorithm aims to increase cluster internal dependencies by moving randomly classes to new clusters and calculating the modularization quality metric. In another work, Seng *et al.*[7] proposed a search-based technique using a genetic algorithm to partition software classes into subsystems. They used a set of refactoring operations to modify the population of modularizations. The fitness function is based on weighted metrics measuring coupling between program components. Some other work[21,44] exploited semantic dependencies between classes in their used clustering techniques in order to divide a system into new clusters containing semantically related classes. All approaches cited above produce a whole new decomposition of classes in packages that is useful only when the system structure is too degraded and needs a deep restructuration. However, it is important to note that to minimize this issue, some approaches were proposed to solve specific design problems such as those of Bavota *et al.*[8] and Palomba *et al.*[15] aiming to remove Promiscuous Packages by decomposing them in new more cohesive ones (using Split Package refactoring).

5.2 Remodularization Approaches Optimizing Original Software Modularization

In this category of approaches, the original distribution of classes is considered and its aim is to optimize the existing modularization. Sahraoui *et al.*[45] used some metrics as indicators for automatically detecting situations where the system can be refactored to improve its quality. They analyzed the impact of various transformations on these metrics using the quality estimation models. In a previous work[46], we proposed an automated bad-smell correction approach to refactoring based on the structural similarity between reference examples and the code to correct. In another work, Harman and Tratt[9] introduced the concept of Pareto optimality to combine two metrics into a fitness function in order to optimize many aspects of the system. They proved how Pareto can usefully be applied to search-based refactoring. Doval *et al.*[16] used a combination of quality metrics (coupling, cohesion, and complexity) to improve the subsystem decomposition. In their approach, the structure of a software system is expressed as a module dependency graph (MDG). An automatic clustering GA is applied to find a "good" partition of MDG. Related modules are regrouped into clusters. Bavota *et al.*[10] used Interactive Genetic Algorithms to integrate the developer's knowledge (decisions to group together (or not) some components) in a remodularization task. The main limitation of this work is that the user's feedback is requested in each iteration of the remodularization process. Therefore, the applicability of this approach is still limited to small/medium size software projects. In our approach, we do not consider the decision maker feedback but we think it can be an interesting perspective to make our approach interactive. Abdeen *et al.*[14] used a search-based algorithm to automatically reduce package coupling and cycles by moving classes over the existing packages. In another work, Abdeen *et al.*[23] adapted NSGA-II to optimize software modularization. Four objectives are considered in their approach: 1) maximizing package cohesion (i.e., intra-package connections), 2) minimizing packages' coupling (i.e., inter-package connections), 3) minimizing cyclic connectivity of packages, and 4) minimizing modification number. The algorithm takes also as input some preferences specified by maintainers like the maximum number of classes that may move to other packages. However, both proposed approaches do not consider semantic cohesiveness within packages.

The first attempt that addresses this issue was by Bavota *et al.*[11] who proposed a mono-objective approach to split an existing package into more cohesive ones. In their approach, the structural and semantic relationships between classes in a package are analyzed in order to identify the set of strongly related ones to split together. In another work, Bavota *et al.*[12] introduced a software remodularization ap-

proach based on semantic and structural information to recommend Move Class refactorings. The proposed approach uses RTM (Relational Topic Models) to analyze natural language topics in classes and packages in order to identify their responsibilities. Besides semantic information, their algorithm considers structural relationships (calls and dependencies) between classes. Both structural and semantic information are used to identify a set of highly similar classes and then to move each class to the package containing the larger number of similar classes. Another approach called MethodBook[47] shares some similarities with this approach[12] in terms of used techniques. But the MethodBook approach[47] differs from the approach using RTM[12] by supporting moveMethod refactoring. Tsantalis and Chatzigeorgion[48] also proposed a technique for suggesting Move Method refactoring opportunities that purely rely on structural analysis. It is important to note that Bavota *et al.*'s[47], Tsantalis and Chatzigeorgion's[48] and our approach work at different granularities (method vs class). Also, none of them consider the refactoring effort when proposing remodularization solutions. Other approaches have been proposed to ensure the semantic of recommended refactorings by exploiting semantic information from the source code[13,49,50]. Mkaouer *et al.*[13] proposed a many-objective search-based approach using NSGA-III to find the remodularization solution that minimizes the number of changes and improves packages structure (by optimizing some metrics such as coupling and cohesion), and the semantic coherence of the restructured program (using vocabulary similarity and dependency-based similarity). Several metrics have been used to quantify the module's quality. Nevertheless, only a few related studies consider the refactoring effort (only 5[13,17,18,23,24] over 31 publications in Search Based Modularization between 1998 and 2017) while optimizing software modularization. In [13] and [23], the effort is measured by the number of the required modifications/changes. Hence, the estimated modularization effort of operations at different granularity levels (class/package), such as extract method and extract class, is considered as the same while they have a different impact in the systems modularity. In another work, Ouni *et al.*[18] presented a Multi-Criteria Code Refactoring approach aiming to minimize the number of design defects, and code changes required to fix them, to preserve design semantics, and to maximize the consistency with the historic of changes. In this work, the authors propose a based-code changes score model that classifies the refactoring operations into low and high level. Furthermore, this model weights them according to their code fragment complexity, and their change impact. Paixao *et al.*[17] introduced a multiobjective evolutionary approach that minimizes disruption while maximizing cohesion and coupling improvement. They used DisMoJo, a disruption metric based on *MoJoFM*, to assess the percentage of code changes in the original modularization required by developers when adopting proposed modularization solutions. However, the proposed approach does not ensure the design coherence of the refactored program.

6 Threats to Validity

6.1 Construct Validity

In our experiments, construct validity threats concern the quantitative measures used in our results. Concerning the structural metrics, we used the cohesion and coupling quality attributes. For textual metrics, we extracted lexical information from source code identifiers. To validate our proposal, we evaluated our results using QG and other independent structural metrics in order to have a better vision in structural improvement of packages quality. To mitigate this threat, we also inspected manually and validated the remodularization solutions by a set of experts. Also, the used semantic measure, as currently defined, does not take into account terms undefined in WordNet. In fact, each term can be a dictionary word or an abbreviation, or an acronym (e.g., teacher, Max, and XML), and only dictionary words are indexed by WordNet. In our case, the similarity between two terms is 1 if they are equals and 0 if one/both of them is/are not indexed. In addition, developers use sometimes identifiers that semantically do not make any sense. One of the existing solutions in the research literature consists in extending the conceptual measure to include the similarity with prior changes[13]. We are planning to incorporate this solution in our future work.

6.2 Internal Validity

The internal threat to validity is related to the use of the stochastic algorithm. We took into consideration the influence of the NSGA-II randomness since our experimental study was performed on the base of 31 independent simulation runs for each problem instance. However, the parameter tuning of the optimization algorithm (namely NSGA-II) used in our experiments creates another internal threat that we need to evaluate

in our future work. In fact, in our work, we tested some possible configurations (varying the size of population, the solution dimension, and the number of iterations), and we selected for each system the best configuration (i.e., achieving the best quality improvement ratio).

6.3 External Validity

In this study, we limited our experiments to 21 releases of four different open-source systems, Java-written and medium-sized. However, we cannot assert that our results can be generalized to industrial applications or systems exploiting specific architectures (e.g., JEE, .NET) or written in other programming languages (e.g., C++, C♯). Future replications of this study are necessary to confirm the generalization of our findings. In addition, our experiment has been conducted with external developers (students), but these students were not the original developers of the subject systems which represent the main threat. However, we recalled that the least experienced participants involved in our study have five years experience in Java programming since subjects are composed of Master and Ph.D. students in software engineering. Indeed, in the future work, we need to involve the original developers of the studied systems. Another limitation of our results is the selection of the best solution from the Pareto front (which consists in selecting the closest solution to the ideal point in terms of Euclidian distance).

7 Conclusions

In this paper, we introduced a novel search-based approach to automate software remodularization. To this end, we used NSGA-II to evolve better refactoring solutions which find a compromise between three objectives: improving modularization quality criteria, optimizing the semantic coherence of the restructured program, and minimizing the refactoring effort task. We evaluated our approach on 21 releases of four real-world, open source software applications. We reported the results on the efficiency of our approach based on a quantitative and qualitative evaluation, and we showed that our approach improves the modularization quality by in average 43% with less than 5% of needed effort to apply the suggested remodularization solutions. We also showed that more than 70% of the recommended refactorings are considered to be meaningful from the developer's point of view. Compared with the existing work, we could conclude that the obtained results are very promising.

As future work, we will investigate other semantic measurements inspired from the information search field, in order to improve the meaningfulness of the recommended refactorings. Finally, we plan to test our approach in larger open source systems and to involve original developers as expert participants during the qualitative evaluation process.

References

[1] Lehman M M. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1984, 1: 213-221.

[2] Eick S G, Graves T L, Karr A F, Marron J S, Mockus A. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 2001, 27(1): 1-12.

[3] Lanza M, Marinescu R. Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer-Verlag Berlin Heidelberg, 2006.

[4] Fowler M, Beck K, Brant J, Opdyke W, Roberts D. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

[5] Harman M, Hierons R M, Proctor M. A new representation and crossover operator for search-based optimization of software modularization. In *Proc. the 4th Annual Conference on Genetic and Evolutionary Computation*, July 2002, pp.1351-1358.

[6] Mitchell B S, Mancoridis S. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 2006, 32(3): 193-208.

[7] Seng O, Bauer M, Biehl M, Pache G. Search-based improvement of subsystem decompositions. In *Proc. the 7th Annual Conference on Genetic and Evolutionary Computation*, June 2005, pp.1045-1051.

[8] Bavota G, de Lucia A, Marcus A, Oliveto R. Software remodularization based on structural and semantic metrics. In *Proc. the 17th Working Conference on Reverse Engineering*, October 2010, pp.195-204.

[9] Harman M, Tratt L. Pareto optimal search based refactoring at the design level. In *Proc. the 9th Annual Conference on Genetic and Evolutionary Computation*, July 2007, pp.1106-1113.

[10] Bavota G, Carnevale F, de Lucia A, di Penta M, Oliveto R. Putting the developer in-the-loop: An interactive GA for software re-modularization. In P*roc. the 4th International Symposium on Search Based Software Engineering*, September 2012, pp.75-89.

[11] Bavota G, de Lucia A, Marcus A, Oliveto R. Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, 2013, 18(5): 901-932.

[12] Bavota G, Gethers M, Oliveto R, Poshyvanyk D, de Lucia A. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology*, 2014, 23(1): Article No. 4.

[13] Mkaouer M W, Kessentini M, Shaout A, Koligheu P, Bechikh S, Deb K, Ouni A. Many-objective software remodularization using NSGA-III. *ACM Trans. Softw. Eng. Methodol.*, 2015, 24(3): Article No. 17.

[14] Abdeen H, Ducasse S, Sahraoui H, Alloui I. Automatic package coupling and cycle minimization. In *Proc. the 16th Working Conference on Reverse Engineering*, October 2009, pp.103-112.

[15] Palomba F, Tufano M, Bavota G, Oliveto R, Marcus A, Poshyvanyk D, de Lucia A. Extract package refactoring in ARIES. In *Proc. the 37th IEEE/ACM International Conference on Software Engineering, Volume 2*, May 2015, pp.669-672.

[16] Doval D, Mancoridis S, Mitchell B S. Automatic clustering of software systems using a genetic algorithm. In *Proc. the 9th International Workshop on Software Technology and Engineering Practice*, September 1999, pp.73-81.

[17] Paixao M, Harman M, Zhang Y, Yu Y. An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE Transactions on Evolutionary Computation*, 2018, 22(3): 394-414.

[18] Ouni A, Kessentini M, Sahraoui H, Inoue K, Deb K. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology*, 2016, 25(3): Article No. 23.

[19] Maqbool O, Babri H. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 2007, 33(11): 759-780.

[20] Candela I, Bavota G, Russo B, Oliveto R. Using cohesion and coupling for software remodularization: Is it enough? *ACM Transactions on Software Engineering and Methodology*, 2016, 25(3): Article No. 24.

[21] Corazza A, di Martino S, Maggio V, Scanniello G. Investigating the use of lexical information for software system clustering. In *Proc. the 15th European Conference on Software Maintenance and Reengineering*, March 2011, pp.35-44.

[22] Hall M, Khojaye M A, Walkinshaw N, McMinn P. Establishing the source code disruption caused by automated remodularisation tools. In *Proc. the IEEE International Conference on Software Maintenance and Evolution*, September 2014, pp.466-470.

[23] Abdeen H, Sahraoui H, Shata O, Anquetil N, Ducasse S. Towards automatically improving package structure while respecting original design decisions. In *Proc. the 20th Working Conference on Reverse Engineering*, October 2013, pp.212-221.

[24] Ouni A, Kessentini M, Sahraoui H, Boukadoum M. Maintainability defects detection and correction: A multi-objective approach. *Automated Software Engineering*, 2013, 20(1): 47-79.

[25] Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 2002, 6(2): 182-197.

[26] Praditwong K, Harman M, Yao X. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 2011, 37(2): 264-282.

[27] Vallée-Rai R, Gagnon E, Hendren L, Lam P, Pominville P, Sundaresan V. Optimizing Java bytecode using the soot framework: Is it feasible? In *Proc. the 9th International Conference on Compiler Construction*, March 2000, pp.18-34.

[28] Farrugia A. Vertex-partitioning into fixed additive induced-hereditary properties is NP-hard. *The Electronic Journal of Combinatorics*, 2004, 11(1): R46.

[29] Jiang J J, Conrath D W. Semantic similarity based on corpus statistics and lexical taxonomy. In *Proc. the 10th International Conference Research on Computational Linguistics*, March 1997, pp.19-33.

[30] Brooks R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 1983, 18(6): 543-554.

[31] Merlo E, McAdam I, de Mori R. Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance: Research and Practice*, 2003, 15(4): 205-244.

[32] Caprile C, Tonella P. Nomen est omen: Analyzing the language of function identifiers. In *Proc. the 6th Working Conference on Reverse Engineering*, October 1999, pp.112-122.

[33] Lawrie D, Morrell C, Feild H, Binkley D. What's in a name? A study of identifiers. In *Proc. the 14th IEEE International Conference on Program Comprehension*, June 2006, pp.3-12.

[34] Poshyvanyk D, Marcus A. The conceptual coupling metrics for object-oriented systems. In *Proc. the 22nd IEEE International Conference on Software Maintenance*, September 2006, pp.469-478.

[35] Gethers M, Poshyvanyk D. Using relational topic models to capture coupling among classes in object-oriented software systems. In *Proc. the 26th IEEE International Conference on Software Maintenance*, September 2010, pp.1-10.

[36] Arnaoudova V, Eshkevari L M, di Penta M, Oliveto R, Antoniol G, Guéhéneuc Y G. REPENT: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 2014, 40(5): 502-532.

[37] Arnaoudova V, di Penta M, Antoniol G. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 2016, 21(1): 104-158.

[38] Seco N, Veale T, Hayes J. An intrinsic information content metric for semantic similarity in WordNet. In *Proc. the 16th European Conference on Artificial Intelligence*, August 2004, pp.1089-1090.

[39] Budanitsky A, Hirst G. Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures. In *Proc. Workshop on WordNet and Other Lexical Resources, Second Meeting of the North American Chapter of the Association for Computational Linguistics, Volume* 2, June 2001, pp.24-29.

[40] Lin D. An information-theoretic definition of similarity. In *Proc. the 15th International Conference on Machine Learning*, July 1998, pp.296-304.

[41] Resnik P. Using information content to evaluate semantic similarity in a taxonomy. In *Proc. the 14th International Joint Conference on Artificial Intelligence*, August 1995, pp.448-453.

[42] Deb K, Jain H. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Trans. Evolutionary Computation*, 2014, 18(4): 577-601.

[43] Wen Z, Tzerpos V. An effectiveness measure for software clustering algorithms. In *Proc. the 12th IEEE International Workshop on Program Comprehension*, July 2004, pp.194-203.

[44] Kuhn A, Ducasse S, Gîrba T. Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 2007, 49(3): 230-243.

[45] Sahraoui H A, Godin R, Miceli T. Can metrics help to bridge the gap between the improvement of OO design quality and its automation? In *Proc. the 8th International Conference on Software Maintenance*, October 2000, pp.154-162.

[46] Kessentini M, Mahaouachi R, Ghedira K. What you like in design use to correct bad-smells. *Software Quality Journal*, 2013, 21(4): 551-571.

[47] Bavota G, Oliveto R, Gethers M, Poshyvanyk D, de Lucia A. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 2014, 40(7): 671-694.

[48] Tsantalis N, Chatzigeorgiou A. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering, 2009*, 35(3): 347-367.

[49] Oliveto R, Gethers M, Bavota G, Poshyvanyk D, de Lucia A. Identifying method friendships to remove the feature envy bad smell: NIER track. In *Proc. the 33rd International Conference on Software Engineering*, May 2011, pp.820-823.

[50] Lee J, Lee D, Kim D K, Park S. A semantic-based approach for detecting and decomposing god classes. arXiv: 1204.1967, 2012. https://arxiv.org/pdf/1204.1967.pdf, Sept. 2018.

Rim Mahouachi is an associate professor in computer science at the Faculty of Sciences of Bizerte, University of Carthage, Tunis, and a researcher in National School of Computer Science, University of Manouba, Manouba. She received her Ph.D. degree in computer science in 2017 from the National School Of Computer Sciences (ENSI), Manouba. She received her Master Degree Diploma (M.Sc.) in automatic and signal processing in 2010, and her Engineering degree in computer science in 2009, from the National Engineering School of Tunis (ENIT), Manouba. Her research interests include the application of computational intelligence techniques, such as machine learning and search-based ones to solve problems in software engineering.