Towards Automatically Improving Package Structure While Respecting Original Design Decisions

Hani Abdeen*, Houari Sahraoui†, Osama Shata*, Nicolas Anquetil‡ and Stéphane Ducasse‡
* Department of Computer Science Engineering, Qatar University, Qatar
hani.abdeen@qu.edu.qa – sosama@qu.edu.qa
† DIRO, Université de Montréal, Montréal(QC), Canada
sahraoui@iro.umontreal.ca
‡ RMod, Inria Lille-Nord Europe, France
Fname.Lname@inria.fr

***Abstract*—Recently, there has been an important progress in applying search-based optimization techniques to the problem of software re-modularization. Yet, a major part of the existing body of work addresses the problem of modularizing software systems from scratch, regardless of the existing packages structure. This paper presents a novel multi-objective optimization approach for improving existing packages structure. The optimization approach aims at increasing the cohesion and reducing the coupling and cyclic connectivity of packages, by modifying as less as possible the existing packages organization. Moreover, maintainers can specify several constraints to guide the optimization process with regard to extra design factors. To this contribution, we use the Non-Dominated Sorting Genetic Algorithm (NSGA-II). We evaluate the optimization approach through an experiment covering four real-world software systems. The results promise the effectiveness of our optimization approach for improving existing packages structure by doing very small modifications.**

***Index Terms*—Software Modularization; Cohesion and Coupling Principles; Multi-Objective Optimization**

I. Introduction

To help maintainers improve software modularization, there is much interest in automatic re-modularization approaches. Major contributions on the problem of software re-modularization, such as [1] and [2], address the problem of automatic software decomposition (i.e., module clustering), rather than the problem of improving existing software decomposition. Their approaches aim at maximizing/minimizing package cohesion/coupling as much as possible [3], [4], regardless of other design factors that are involved in the software existing modularization [5], [6]. Consequently, in such approaches, it can be difficult to understand the resulting structure and/or to validate it [6]. In literature, few articles address the problem of optimizing packages structure within existing modularizations [7], [8]. Yet, this body of work has formulated the remodularization as a single-objective optimization problem. However, because the conflict between cohesion and coupling properties [3], [4], single-objective remodularization approaches might produce sub-optimal solutions [2].

To address the limitations explained above, we present a multi-objective optimization approach that improves packages structure with regard to the well-known design principles, *cohesion* and *coupling* [3], by modifying as less as possible the existing packages organization. Moreover, the optimization approach avoids increasing the size of large packages and/or merging small packages into larger ones. Finally, the optimization approach allows maintainers to guide and control the optimization process by defining a variety of constraints.

To this contribution, we used and adapted the multi-objective Non-Dominated Sorting Genetic Algorithm (NSGA-II) [9]. We chose this algorithm, in particular, because it has been shown to perform well in similar problems such as the automated detection and correction of class design defects [10]. To evaluate our optimization approach we perform a comparative study with the single-objective optimization approach defined by Abdeen *et al.* in [8]. The results promise the effectiveness of our optimization approach for improving the quality of packages structure by doing very small modifications, and without merging small packages into larger ones.

The rest of this paper is organized as follows: Section II presents the used terminology. Section III overviews main challenges for optimizing packages structure. Section IV positions our approach with relevant existing works on software remodularization. Section V details our multi-objective approach of optimizing existing packages structure. Section VI describes the research questions and sets up the empirical study that aims to evaluate the effectiveness of our multi-objective approach, using a variety of object-oriented applications. Section VII analyzes the results of the empirical study and answers the research questions. Section VIII discusses the contributions and limitations of our approach with regard to existing works on software re-modularization, then considers threats to validity, before concluding in Section IX.

II. Background

This section introduces the terminology used in this paper. We define an OO software *Modularization* $\mathcal{M}$ as a decomposition of the set of software classes $\mathcal{C}$ into a set of packages $\mathcal{P}$, where each package p represents a container of classes. We define package size, $size(p)$, by the number of its classes.

Inter-class dependencies: we consider three types of dependencies: *method call*, *inheritance* and *class reference* (i.e., where the name of a class is explicitly used within other classes). The pair (ci, cj) denotes a dependency directed from a class ci to another class cj. We denote the set of all Inter-Class Dependencies within a given modularization by ICD.

Internal vs. External package dependencies: a dependency is *intra*-package (*internal*) if it relates two classes belonging to the same package. Otherwise, the dependency is *inter*-package (*external*). In Modularization1 in Figure 1, the dependency ($c1$,$c2$) is internal to $p1$, while ($c4$,$c1$) is an external dependency relating $p2$ to $p1$. We denote the set of all Inter-Package Dependencies within a modularization by IPD.

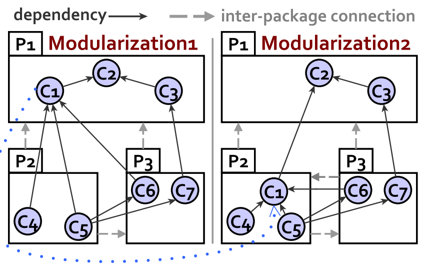

Figure 1. Modularization presentation: Package Dependency Graph. Different organizations of the set of classes $[c1..c7]$ into 3 packages $[p1, p2, p3]$.

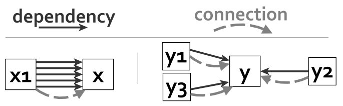

Figure 2. Explanation for Package Connections vs. Dependencies [4].

Package connectivity: a *Connection* from a package x to another one y exists if there is/are some dependency/ies from x pointing to y. Modularization1 in Figure 1 shows that there are two connections from $p2$ pointing to $p1$ and $p3$. IPC denotes the set of all Inter-Package Connections within a given modularization. To explain the difference between inter-package connections and dependencies, Figure 2 shows that although the number of inter-package dependencies among the packages $\{x, x1\}$ is larger than that among the packages $\{y, y1, y2, y3\}$, the number of inter-package connections among $\{x, x1\}$ is 3 times smaller than among $\{y, y1, y2, y3\}$.

Package cyclic dependencies and connections: Figure 1 shows that in Modularization$_2$ the packages $p2$ and $p3$ are involved in a cycle. Hence, the connections, and dependencies, between $p2$ and $p3$, in Modularization$_2$, are cyclic. We denote the set of all Inter-Package Cyclic Dependencies/Connections within a given modularization by IPCD/IPCC.

Evolution Representation: we associate each modularization $\mathcal{M}$ to a refactoring list, namely Class Moving List (CML), representing the modifications that were applied on the original modularization to obtain $\mathcal{M}$. Only one refactoring type is considered, which is moving classes among packages. During the evolution process, it may happen that a class c moves out of its package and later returns to it. In such a case, the class-move element of c will be removed from the CML since c, in the end, did not change package.

III. Issues in Improving Packages Structure

In this section, we discuss the main issues in automated optimization of existing packages structure.

A. Conflict between Package Quality Properties

The organization of classes into packages must follow the principles of cohesion and coupling [4], [3], [2]. Martin discussed principles of package design, addressing package cohesion and coupling [3]:

The Common Closure Principle (CCP): the CCP aims at minimizing the number of packages that would be changed in any given release cycle. To achieve this goal, classes that change together should be packaged together. We relate the probability of classes being changed together to the number of dependencies among those classes. Hence, optimizing package *common closure* property requires reducing inter-package dependencies.

The Common Reuse Principle (CRP): according to the CRP, classes that are reused together should be packaged together. The rational behind CRP is that changing any class within a considered package p will have the same impact-propagation if another class is changed within p. As a consequence, this may reduce the number of packages to upgrade and re-validate, when a package changes. For example, changing the package x in Figure 2 may impact only one package ($x1$), whilst changing the package y may impact three packages ($y1$, $y2$ and $y3$). Hence, optimizing package *common reuse* property requires reducing inter-package connections.

The Acyclic Dependencies Principle (ADP): to achieve the desired goals of CCP and CRP packages should not be involved into cyclic dependencies [3] –since cyclic dependencies between packages may involve a recursive impact propagation of package changes. Therefore, to improve packages structure, we need first of all to reduce cyclic dependencies, as well as cyclic connections, between packages.

CCP and CRP are conflicting package properties. Abdeen *et al.* [4] showed that reducing inter-package dependencies (i.e., improving the CCP) may increase inter-package connections (i.e., degrading the CRP). Furthermore, Martin [3] states that the CCP and CRP principles are mutually exclusive and cannot be simultaneously satisfied.

B. Developers' Decisions and Original Packages Structure

The distribution of classes over packages might depend on other factors than package cohesion and coupling [11], [12], [13], [5], [3], [14]. Unfortunately, up to date, relevant previous work on software re-modularization (e.g., [2], [7], [1]) addresses the problem of maximizing cohesion and/or minimizing coupling of packages/modules, regardless of the original software structure and other design factors. Despite its success, this body of work raises the following uncomfortable questions:

- *What if software maintainers want to control the optimization process and guide it?*
- *How to improve package cohesion and coupling by performing minimal modifications on existing packages organization?*

We believe that the existing modularization is an important map that reflects the decisions of the domain experts. Additionally, we claim that the optimization approach of packages structure should allow its end users to control the optimization process and guide it, so that to produce acceptable solutions from developer's perspective.

C. Package Size and Class Distribution

Studies agree that package size should not exceed a given limit [15]. However, it is unfortunately difficult, or unfeasible, to determine automatically that limit and/or an ideal package size [5]. Furthermore, in real applications, packages do not have similar sizes. Let us take the example of JBoss and ArgoUML applications (see VI-A for more information about studied applications). Both applications contain large packages beside many small packages (e.g., 106 packages in JBoss contain no more than a couple of classes).

In summary, *the existing modulrization is an important map that reflects acceptable package sizes from developer's perspective, however Blob packages should be avoided.*

IV. Relevant Related Work

Praditwong *et al.* have recently formulated the problem of software clustering as a multi-objective problem [2]. Their work presents two multi-objective clustering approaches. Their approaches use the following objectives: (1) maximizing the Modularization Quality (MQ) measurement [16]; (2) reducing inter-package dependencies; (3) increasing intra-package dependencies; (4) maximizing the number of clusters, which aims at producing clusters of similar sizes; and (5) minimizing the number of isolated clusters. What is surprising in (2) and (3) objectives is that they are formally identical. In fact, in their approach, as in ours, a dependency is either intra-package or inter-package, but never both or none. Thus, reducing inter-package dependencies by a δ value surely leads to increase intra-package ones by the same δ value. As for the 4th objective, it raises the following uncomfortable question: *what is the relationship between the produced clusters and the design factors of the original packages structure?*. Praditwong *et al.* said clearly that the goal of their approaches is to produce cohesive clusters, as much as possible, regardless of the package original design.

Similar to the aforementioned approach of Praditwong *et al.*, Bavota *et al.* [11] proposed an *interactive* multi-objective remodularization approach. In their approach, *in each generation* of the re-modularization process, the solution with the highest MQ value in the Pareto set is selected and suggested to end users. Then, end users should analyze the suggested solution, class-by-class/package-by-package, and provide their feedback. User feedback can be either about classes which should stay together, or not, and/or about small/isolated clusters. After that, the remodularization process restarts with the suggested solution, but keeps user feedback penalizing the fitness of solutions which do not satisfy user feedback.

In literature, few prior articles address the problem of optimizing packages structure within existing modularizations [7], [8]. Recently, Bavota *et al.* [7] proposed an approach that tackles the problem of package size and cohesion improvement. They proposed a single-objective, automatic, approach to split an existing package into smaller but more cohesive ones, — from the perspective of structural and semantic relationships between package classes. The main drawback of their approach is that it does not consider the relations between different packages (i.e., package coupling/connections). Furthermore, their package decomposition approach cannot propose appropriate refactorings for small packages (e.g., packages which contain no more than a couple of classes: the case of 106 packages in JBoss).

Abdeen *et al.* [8] defined single-objective optimization approach based on the Simulated Annealing technique (SA). The SA optimization approach aims at improving package cohesion and reducing package coupling and cycles by moving classes among existing packages. However, as a single objective optimization approach, SA takes the risk of optimizing some property at cost of others. For instance, Abdeen *et al.* state that their approach has a strong trend to improve package coupling at cost of class distribution and package size property. Abdeen *et al.* state that in the SA's produced solutions a large set of packages, especially small ones, are removed since their classes are merged to larger packages.

Like the SA optimization approach proposed in [8, Abdeen *et al.*], this paper addresses the problem of optimizing existing modularizations rather than producing new ones (e.g., software clustering: [11, Bavota *et al.*] and [2, Praditwong *et al.*]). We define an approach that considers the original modularization as an important map that reflecting developers' decisions. Indeed, our optimization approach considers explicitly the objective of minimizing the size of proposed refactorings. However, unlike the SA optimization approach, this paper proposes a multi-objective optimization approach that avoids improving a property of package design at cost of other properties, and avoids merging the classes of small package into larger ones.

Inspired by the interactive approach by Bavota *et al.* [11], to improve packages structure in an iterative and interactive manner, our approach allows maintainers to control and guide the whole process of optimization, using generic constraints. However, unlike the approach by Bavota *et al.*, the constraints used by our approach are generic enough to cover a large variety of contexts, which would facilitate the end user tasks. For instance, maintainers can specify a limited amount of authorized modifications, classes which should not change their packages, packages which should not be changed, and specify the locality of classes that may change their packages. Moreover, our approach does not violate the maintainer constraints along the optimization process, rather than requesting the maintainer feedback in every generation.

V. Multi-Objective Optimization Approach

This paper proposes a multi-objective approach that aims at improving all the package properties CCP, CRP and ADP, within existing modularizations. This is by perturbing as less as possible the existing packages organization, and without merging small packages into larger ones. To this contribution, we selected and adapted a multi-objective evolutionary algorithm: the Non-dominated Sorting Genetic Algorithm (NSGA-II) [9]. The aim of NSGA-II is to find in a single run a set of Pareto optimal solutions that are non-dominated with respect to each other. By definition, a solution s_1 *dominates* another one s_2 with regard to a list of objectives $\mathcal{O}$ ($s_1 \succ_{\mathcal{O}} s_2$), if s_1 is better than s_2 for at least one objective, while s_2 is not better than s_1 regarding all the objectives in $\mathcal{O}$ [9]:

$$\forall o_i \in \mathcal{O}.\ o_i(s_1) \geq o_i(s_2)\ \wedge\ \exists o_j \in \mathcal{O}.\ o_j(s_1) > o_j(s_2) \qquad (1)$$

The NSGA-II explores the search space by making and evolving a population of candidate solutions using selection and genetic operators. The output of the algorithm is a set of the fittest solutions produced along all generations. To apply NSGA-II to a particular problem, the following elements have to be defined for its implementation: the optimization objectives, domination function and the Genetic operators used to explore the search space. The next sections explain our design and adaptation of these elements to the problem of optimizing packages structure.

A. Optimization Objectives

The objectives of our optimization approach are:

Maximizing package cohesion by transforming inter-package dependencies (IPD) to *intra* package ones.

Minimizing package coupling by reducing inter-package connections (IPC).

Minimizing package cycles by reducing inter-package cyclic dependencies and connections (i.e., IPCD and IPCC).

Avoiding Blob packages by avoiding merging the classes of small packages into larger ones.

Minimizing the modification of original modularizations, so that the modification of existing packages organization is *near-minimal "with respect to the achieved improvements to package cohesion, coupling and cyclic connectivity"*.

However, Ishibuchi *et al.* [17] reported that the convergence of NSGA-II to the Pareto front considerably slows down when the number of objectives exceeds 4 objectives. To adapt NSGA-II for many objective problems and improve its convergence to the Pareto front, Sato *et al.* [18] proposed to introduce minor changes to NSGA-II by modifying the objective functions as follows: $co_i(s) = o_i + \beta \times \sum o_j(s) \quad , \forall o_i \in \mathcal{O}$. Where $\mathcal{O}$ is the set of objectives, β is a prespecified constraint and co_i is the modified objective corresponding to the i^{th} objective (o_i) in $\mathcal{O}$. Ishibuchi *et al.* [17] reported that this modification of the objective functions noticeably improves the convergence property of NSGA-II. Basing on the aforementioned studies of Sato *et al.* and Ishibuchi *et al.*, we carefully define our objectives using a formula similar to that one above-mentioned. Since our optimization process does not change the dependencies between classes (ICD), we use ICD to normalize some of our metrics in the interval [0..1].

High Cohesion Objective: we define the Quality of Common Closure Property ($QoCCP$) and the Quality of Acyclic Dependencies Property ($QoADP$) measurements by using respectively the normalized ratio of IPD and IPCD to ICD: $QoCCP = 1-({}^{IPD}/_{ICD})$; $QoADP = 1-({}^{IPCD}/_{ICD})$. Maximizing $QoCCP$ and $QoADP$ will reduce (increase) inter (intra) package dependencies and reduce inter-package cyclic dependencies, and this will optimize the common closure and acyclic dependencies properties. Hence, we formulate the objective of high Cohesion (Coh) as follows:

$$Coh = \lambda \times (QoCCP + \beta \times QoADP) \qquad \textbf{O}\text{bj.1}$$

The rational behind the occurrence of the term $\beta \times QoADP$ in the above equation, rather than merely $QoADP$, is to give weight to cyclic-dependencies ($\beta \geq 1$). In this paper we set the value of β to 2, so that cyclic-dependencies have double weight than acyclic ones. The rational behind the occurrence of the λ factor is that optimizing $QoCCP$ and $QoADP$ could be done by merging some packages into other ones: i.e., reducing the number of packages. In order to avoid such solutions, we use the λ factor which we define as follows:

$$\lambda = \frac{\mid notEmptyPackages \mid}{\mid allPackages \mid} \qquad (2)$$

The λ factor takes its value in]0..1]. When the number of empty packages increases then the λ value decreases and penalizes the Coh value. Our hypothesis is that optimizing Coh will increase (reduce) intra- (inter-) package dependencies, and particularly reduce inter-package cyclic dependencies, and this is not at cost of removing some packages.

Low Coupling Objective: similarly to the objective of high cohesion (Coh) –Obj. 1, we formulate the objective of low Coupling Cop as follows:

$$Cop = \lambda \times (QoCRP + \beta \times QoACP) \qquad \textbf{O}\text{bj.2}$$

Where $QoCRP$ and $QoACP$ measures respectively the Quality of package Common Reuse and package Acyclic Connection properties, and defined similarly to $QoCCP$ and $QoADP$: $QoCRP = 1 - ({}^{IPC}/_{ICD})$; $QoACP = 1 - ({}^{IPCC}/_{ICD})$. Our hypothesis is that optimizing Cop will reduce inter-package connections, particularly cyclic ones, and this is not at cost of removing some packages.

Low Modification Degree Objective: to evaluate the modifications applied to the original modularization $\mathcal{M}_{original}$ in order to obtain a new one $\mathcal{M}_{new}$, we define the Modification Degree (MD) in $\mathcal{M}_{new}$ as the number of moved classes in $\mathcal{M}_{new}$. The smaller the MD value, the smaller the perturbation of original packages organization. *However, our objective is not to minimize the MD in an absolute way. Rather, it is to ensure that the achieved improvement of packages structure cannot be done with a smaller modification degree*. For this purpose, we formulate the objective of Low Modification Degree, LMD, as follows:

$$LMD = Coh + Cop - (\Delta \times MD) \qquad \textbf{O}\text{bj.3}$$

The rational of LMD is to ensure that an improvement of Coh and/or Cop should not be accepted unless it is larger than the applied modification. Moreover, with the LMD objective, if two solutions have the same cohesion and coupling quality, then the solution with the smallest modification degree will be selected. As for the Δ factor in the above equation, we define it as follows: ${}^{1}/_{ICD}$ –since Coh and Cop are defined relatively to the number of inter-class dependencies (ICD: see the aforementioned definitions of $QoCCP$, $QoADP$, $QoCRP$ and $QoACP$).

Quality of Class Distribution: the idea is that the optimization process should avoid moving classes from small packages to large ones. This paper defines the border which specifies the size-range of small packages and that of large ones as the Average Package Size ($APS = {}^{\mid \mathcal{C} \mid}/_{\mid \mathcal{P} \mid}$). ASP represents the size of every package if classes are distributed equally over packages. Let *size_LargePackages* (*size_SmallPackages*) denote the number of classes packaged in large (small) packages. We define the Quality of Class Distribution $QoCD$ as the ratio of *size_SmallPackages* on *size_LargePackages*.

$$QoCD = \frac{size\_SmallPackages}{size\_LargePackages} \qquad \textbf{O}\text{bj.4}$$

If large packages grow and small ones shrink, then the value of $QoCD$ becomes smaller. Hence, the optimization process must avoid degrading $QoCD$. However, in this paper we do not assume that classes should have an uniform distribution across packages (see our discussion in Section III-C). Therefore, our approach does not use the objective $QoCD$ for maximizing, but rather as a constraint for avoiding God packages. Next section presents our adaptation of the domination function and usage of the $QoCD$ objective.

B. The Domination Function

We define our domination function using the metrics introduced in Section V-A: $\mathcal{O} = \{Coh, Cop, LMD, QoCD\}$. On the one hand, among those objectives, our optimization approach aims at maximizing the following subset: $\mathcal{O}_{opt} = \{Coh, Cop, LMD\}$. On the other hand, the optimization process must avoid solutions that degrade the quality of class distribution $QoCD$. For this purpose, we introduce the predicate $QoCD^{\succeq}(\mathcal{M})$ which returns *true* if, and only if, the quality of the class distribution is *not worse* in $\mathcal{M}$ than in the original modularization $\mathcal{M}_{original}$. We adapt the domination function, so as we say that $\mathcal{M}_1$ dominates $\mathcal{M}_2$ if $\mathcal{M}_1$ dominates $\mathcal{M}_2$ with regard to $\mathcal{O}_{opt}$ (see Equation (1)) and $QoCD^{\succeq}(\mathcal{M}_1)$ is true. Hence, solutions with degraded $QoCD$ cannot dominate other solutions.

C. Modularization Evolution with Maintainer Constraints

The NSGA-II uses crossover and mutation operators to explore the search-space. In the following, we explain our adaption of the crossover and mutation operators.

The crossover operator considers every modularization $\mathcal{M}$ through its class-moving list (CML) (Section II). When applying crossover on two parents, $\mathcal{M}_1$ and $\mathcal{M}_2$, one crossover point will be randomly selected in the CMLs of the parents, and two new CMLs, representing two children ($\mathcal{M}'_1$, $\mathcal{M}'_2$), will be generated by crossing the parent lists. Applying the mutation operator on a modularization will produce a new one where some class c, namely the modification actor c_{actor}, is moved from its current package to another one. However, maintainers may specify the following constraints:

C1. Constrain the Refactoring Space: maintainers can specify some classes and/or packages as *frozen*. As a consequence, a *frozen* package will never be changed and a *frozen* class will never change its package.

This is an important constraint since maintainers may agree that some packages, even though they are not “cohesive”, are well designed and should not be changed. Furthermore, maintainers may be interested in limiting the refactoring space to a given subsystem. In such a context, the advantage of our approach is that it will search for a refactoring list limited to that subsystem but takes into account the improvement of package cohesion, coupling and cycles of the whole system.

C2. Guide the Refactoring Process: specify to which packages a class is likely to move.

The particular importance of this constraint is that maintainers can enforce some rule on class locality. For example, in a layered application context, the maintainer might need to control the optimization algorithm, so that the classes of each layer move only among the packages of that layer. For this purpose, we introduce the class-package friendship concept: a class c can move only to its *friend* packages. We use $Fr(c)$ to denote the set of packages that are friends to c.

C3. Limit the Refactoring Size: specify the maximal number of classes that may move.

Although our approach considers the objective of minimizing the modification amount (see Obj.3), this constraint is important when maintainers look for local solutions that are limited to a pre-specified amount of modification. We use d_{max} to denote the maximal number of classes that can change their packages. To satisfy this constraint, a class c in a modularization $\mathcal{M}$ can move if: (1) the number of classes changed their packages in $\mathcal{M}$ is smaller than the d_{max}; or (2) c has already changed its original package, thus moving it will not increase the number of classes changed their packages.

It is worth noting that utilization of the above-mentioned constraints could lead to local-optimal solutions. Our optimization approach take this risk since it is intended to satisfy the following goals:

Avoid bad refactoring we want to avoid moving a class to packages that are not related, at all, to that class, therefore classes may move only to their friend packages.

Support further constraints we want our optimization approach to be *extensible* for different restrictions to the refactoring space. The maintainer may define the *friends* set differently for different classes: e.g., some classes can move to their client and provider packages, while other classes can move only among UI packages.

VI. Experiment Definition

This section describes the experiment conducted to evaluate our optimization approach.

A. Context, Goal and Perspective

The *context* of the study consists of four object-oriented applications: JHotDraw$_{v7.1}$, is a Java GUI framework for technical Graphics; JBoss$_{v6.0.0}$, is a widely used Java application server; ArgoUML$_{v0.28.1}$, is an UML editing tool; and Hibernate$_{v4.1.4}$, is a Relational persistence for idiomatic Java. As illustrated in Table I, we chose those applications since they differ in terms of size, number of classes ($|\ \mathcal{C}\ |$); number of packages ($|\ \mathcal{P}\ |$); number of inter-package dependencies and connections (IPD and IPC), and cyclic ones (IPCD and IPCC). For instance, the number of inter package cyclic dependencies and connections (IPCD and IPCC) in ArgoUML and Hibernate is much larger than that in JBoss (and JHotDraw). Furthermore, in JBoss and Hibernate many packages have a very small size: 106 packages in JBoss and 72 packages in Hibernate contain no more than 2 classes.

The *goal* of the study is to investigate the quality of the produced solutions of our approach and compare it to the quality of original modularizations and of produced solutions of another optimization approach. This is from the *perspective* of researchers and with respect to package's cohesion, coupling, cycles and size, and with respect to the amount of modifications in produced solutions.

Baseline Optimization Approach: following our discussion in Section IV, we believe that the SA optimization approach proposed by Abdeen *et al.* [8] is the most appropriate baseline approach for our comparative study. Like our approach (that we refer to by NSGA), SA aims at improving existing modularizations rather than producing new ones, and it considers package coupling (connections) and cycles.

B. Hypotheses and Assessment Criterion

Hypothesis on “Package Cohesion”: *“NSGA's solutions are characterized with a better package cohesion than SA's solutions and original modularizations.”*

★ *Assessment:* in produced solutions, IPD and IPCD should be reduced, and the values of the Modularization Quality measurement (MQ) [16] should be increased. Although MQ does not belong to the objectives of our approach NSGA neither to the fitness function of the SA approach, we use it since it is traditionally used in module clustering approaches (e.g., [1] and [2]). A drawback of MQ is that its values may be *‘arbitrarily’* large, depending on the number of packages in the concerned modularization [2]. Fortunately, both approaches NSGA and SA do not increase the number of packages since they are limited to move classes among existing packages. In fact, we consider this drawback of MQ as an advantage for comparing between the cohesion of NSGA's solutions and SA's solutions: if the optimization approach reduces IPD at cost of merging some packages into larger ones this will lead to a reduced number of packages, and as a consequence will negatively impact on the MQ value.

Hypothesis on “Package Coupling”: *“NSGA's solutions are characterized with a lower package coupling than SA's solutions and original modularizations.”*

★ *Assessment:* the number of inter-package connections (IPC), as well as that of cyclic ones (IPCC), should be reduced in produced solutions.

Table I
INFORMATION ABOUT THE USED SOFTWARE APPLICATIONS.

	Package Cohesion			Package Coupling		Package Size		Basic information		
	IPD	IPCD	**MQ**	IPC	IPCC	$\mid \mathcal{P} \mid$	QoCD	$\mid \mathcal{C} \mid$	APS	ICD
JHotDraw	1322	312	10.80	146	10	38	0.429	516	13	2196
JBoss AS	1741	461	221.73	374	19	343	0.406	2820	8	4303
ArgoUML	7289	1249	34.46	717	47	119	0.280	2365	20	9938
Hibernate	6956	1846	78.80	1557	123	259	0.371	3126	12	10555

Hypothesis on "Package Size": *"NSGA does not degrade the package size property, so that this property is more respected by NSGA than by SA."*

★ *Assessment:* to assess the package size property in produced solutions, we use two measurements: QoCD Obj.4 (Section V-A) and the number of packages that their classes have been merged to other ones ($\mathcal{P}_\phi$). In produced solutions, the set of empty packages $\mathcal{P}_\phi$ should be as small as possible. Ideally, produced solutions should not include empty packages ($\mid \mathcal{P}_\phi \mid = 0$). Moreover, the values of QoCD should not be degraded (decreased) in the produced solutions.

Hypothesis on "Modifications versus Optimization": *"The modification degree in the NSGA's solutions is smaller than that in the SA's solutions, this is with regard to the achieved improvement of packages structure."*
Generally speaking, the smallest the modification degree (the MD value) is, the smallest the perturbation of original packages organization is. However, to assess the modification degree in produced solutions *with regard to achieved improvements*, we define the Rate per Refactoring of Achieved Improvement (RRAI) measurement. The RRAI aims at comparing the Rate-Per-Class (RPC) of each quality measurement (i.e., IPD, IPCD, MQ, IPC and IPCC) to the Rate-Per-Moved Class (RPMC) of the achieved improvement to that measurement. The RPC of a given measurement m is computed as follows: $RPC(m) = {}^{m_{or}}/_{\mid \mathcal{C} \mid}$, where m_{or} is the value of m in the original modularization and $\mathcal{C}$ is the set of all classes. Hence, moving a class should decrease/increase, on average, the m value by the RPC value. The RPMC of the achieved improvement to m in a produced solution $\mathcal{M}_{new}$ is computed as follows: $RPMC(m) = {}^{\delta m}/_{MD}$, where δm is the decreased (increased) value of m in $\mathcal{M}_{new}$, if m is to be minimized (maximized), and MD is the number of classes that changed their packages in $\mathcal{M}_{new}$. Hence, the RPMC value represents the average contribution of every moved class to the achieved improvement to m. Our hypothesis is: the average contribution of moved classes to the achieved improvement of m, i.e., RPMC(m), should be larger than the average contribution of all classes to the original value of m, i.e., RPC(m). We define the RRAI as the ratio of RPMC on RPC:

$$RRAI(m) = \frac{RPMC(m)}{RPC(m)} : m \in \{\text{IPD}, \text{IPCD}, \text{MQ}, ...\} \quad (3)$$

★ *Assessment:* the larger the value of RRAI(m), the smaller the modification amount with regard to the achieved improvement to m. Ideally, RRAI values would always be larger than 1 (i.e., RPMC(m) > RPC(m)). However, since we want to assess the quality of produced solutions with regard to different measurements, we expect that the arithmetic mean of RRAI values with regard to those measurements, $\overline{\text{RRAI}}$, should be larger than 1.

C. Experiment Scenarios

Global Optimization: in this scenario we do not specify any constraint on the optimization process of NSGA and SA, so that: all classes can change their packages, classes can move to any packages and all packages can be changed.

Controlled Optimization: in this scenario we control and limit the optimization process using the following specifications. We limit the locality of a class c to the c' friend packages (C2), which we define as the set of provider and client packages of c. Hence, c can move only to packages that contain classes directly related to c via explicit dependencies. Furthermore, we want to test the optimization process when the refactoring space is constrained (C1) and limited (C3). For this purpose, we limit the maximal authorized modification degree, d_{max}, to 5% (i.e., C3: only 5% of the application classes can change their packages). Finally, based on the benefits of isolating abstractions and details from each other [19], we specify packages that contain only interfaces and/or abstract classes as frozen packages (C1). This is in order to keep those packages, with the interfaces/abstracts they include, isolated from the implementation classes. Hence, the number of packages/classes that will be frozen in the studied applications is as follows: none in JHotDraw, 8 packages and 45 classes in JBoss, 2 packages and 23 classes in ArgoUML, 26 packages and 71 classes in Hibernate.

D. Algorithmic Parameters

Regarding the algorithmic parameters of our NSGA approach, we set the population size and the number of generations to 100 and 200, receptively. As for the SA (Simulated Annealing) optimization approach, we set its algorithmic parameters as follows: the start and stop temperatures are respectively set to 22.8 and 1 (using a geometric cooling scheme: $T_{next} = 0.9975 \times T_{current}$ [8]), and the number of local search iterations is set to 15; so that the number of candidate solutions evaluated during the evolution process of both approaches, NSGA and SA, is the same. Regarding the parameters of the genetic operators, we use values similar to those used by Praditwong *et al.* [2]: the probability of crossover and mutation are respectively 1.0 and $0.04 \times \log_2(\mid \mathcal{C} \mid)$, where $\mid \mathcal{C} \mid$ is the number of classes in the concerned modularization.

E. Solution Selection and Results Collection

The output of our multi-objective approach NSGA is usually a set of solutions (Pareto set solutions), but the output of the single-objective approach SA is a single solution. To fully automate our comparative study, in each run of the NSGA we calculate the arithmetic mean of the values of each objective function $\overline{o}$ ($\overline{Coh}$, $\overline{Cop}$, $\overline{QoCD}$ and $\overline{LMD}$) for all Pareto set solutions. Then, the median solution in the Pareto set is selected as follows:

$$s_{chosen} \Leftrightarrow \min_{i=1}^{|Pareto\ set|} \left(\sqrt{\sum_{o_j \in O} (o_j(s_i) - \bar{o}_j)^2} \right) \quad (4)$$

Since both optimization approaches, NSGA and SA, are probabilistic by nature, it is essential to use statistical tests to support the conclusions that we derive from the approach's results. For this purpose, with each optimization approach we performed 30 runs on each application and collected the representative solutions produced by every runs. Then we computed the assessment measurements for all collected solutions (Section VI-B) and compared between the NSGA's produced solutions and the SA's ones using two-tailed Wilcoxon tests.

VII. Results Analysis

This section analyzes the results of the experimental study to address the research hypotheses outlined in Section VI-B.

A. Package Cohesion

NSGA. Table II shows that our optimization approach (NSGA) succeeded in improving package cohesion, as measured by IPD, IPCD and MQ, for all studied applications and in both experiments: *Global* optimization and *Controlled* one. However, Table II shows that the improvement of the Acyclic Dependencies property (i.e., the reduced value/percent of the IPCD original value in Table I) is significantly larger than the improvement of other cohesion properties. This is explained by our definition of the high cohesion objective (Obj.1), where reducing inter-package cyclic dependencies has more importance than reducing acyclic ones. Taking for example the case of Hibernate application in the global optimization experiment, the NSGA approach improved package cohesion on average as follows: transformed 333 inter-package dependencies ($\approx 5\%$ of IPD in the original modularization) to intra-package ones; removed 707 inter-package cyclic dependencies ($\approx 38\%$); and increased the MQ value by 14 ($\approx 18\%$).

NSGA compared to SA. Reading the delta values of cohesion measurements in Table II we observe the following. With regard to the MQ measurement, the NSGA performed better than SA in almost all cases and both experiments. A notable exception is the case of JBoss in the controlled optimization experiment, where the average MQ value is higher in SA's solutions than in NSGA's solutions (MQ$_{NSGA}$ - MQ$_{SA}$ = -0.55). However, the delta value in this case is not statistically significant. We also observe that NSGA succeeds in reducing inter-package cyclic dependencies (IPCD) much more than SA. Keep in mind that we set the same weight to cyclic connections and dependencies in both approaches NSGA and SA. Besides, we observe that SA performed better than NSGA in reducing (increasing) inter- (intra-) package dependencies (see δIPD values): five cases in favor of SA (ArgoUML and Hibernate in both experiments, and JBoss in the second experiment) against 3 cases in favor of NSGA (JHotDraw in both experiments, and JBoss in the first experiment).

As a summary, the NSGA approach can well improve the cohesion of existing packages, even though maintainers constrain the refactoring's space and/or size. Indeed, the NSGA approach significantly outperforms the SA approach in reducing inter-package cyclic dependencies and improving the modularization quality as measured by MQ. However, we believe that the contradictory results with regard to the IPD and MQ measurements are mainly due to changes in the package size property. We return back to this point later in this section.

B. Package Coupling

NSGA. Same as package cohesion, Table II shows that our approach succeeded in well reducing package coupling and cycles in all studied applications and both experiments. This is even true for case-studies that have tightly coupled packages, such as ArgoUML and Hibernate (see Table I and Section VI-A). Taking for example the case of ArgoUML in the global optimization experiment, NSGA reduced, on average, inter-package connections (IPC) and cyclic ones (IPCC) by 144 ($\approx 20\%$) and 19 ($\approx 40\%$), respectively. Another example is the case of Hibernate in the controlled optimization experiment, where NSGA reduced IPC and IPCC by $\approx 10\%$ and $\approx 39\%$, respectively. This is an empirical evidence that the NSGA optimization approach successes in reducing and optimizing package connectivity in existing modularizations, regardless of their size and/or complexity.

NSGA compared to SA. By comparing the values of IPC and IPCC measurements between NSGA's solutions and SA's solutions (see the delta values of coupling measurements in Table II) we observe the following. SA performed better than NSGA in reducing package connections, whilst NSGA performed better than SA in reducing package cycles. Taking for example the case of Hibernate in the global optimization experiment, where SA reduced, on average, 100 connections ($\approx 6\%$) more than NSGA; whilst NSGA reduced, on average, 10 cyclic connections ($\approx 8\%$) more than SA. We believe that this different results can be explained as follows. NSGA as a multi-objective optimization approach, unlike SA, aims at improving the package design properties mutually (not at cost of each others). Furthermore, it is worth noting that merging some packages into others would automatically reduce inter-package connections (and dependencies), but not necessarily package cycles. Hence, we need to check the package size property in solutions produced by NSGA and SA before deriving our conclusions about package coupling.

C. Package Size and Class Distribution

NSGA. Table III shows that for all case-studies, and in both experiments, the general shape of package size and class distribution in NSGA's solutions is very similar to that shape in original modularizations. Actually, in NSGA's solutions for all case studies no package is removed ($| \mathcal{P}_\phi | = 0$). Moreover, we observe that the quality of class distribution (QoCD) in NSGA's solutions is not degraded. Rather, the QoCD is improved in some cases: e.g., in JBoss, which contains a large set of very small packages, the QoCD is increased by 0.01 ($\approx 2\%$ of the QoCD original value).

NSGA compared to SA. As for the SA's solutions, we observe that a relatively large set of packages are merged into other ones, and the QoCD is considerably degraded. That is for all case-studies, and in both experiments. Taking for example the case of Hibernate in the first experiment, where on average 63 packages ($\approx 24\%$ of packages in the Hibernate's original modularization) are merged to other packages. In this case, the QoCD is decreased (degraded) by 0.09 ($\approx 24\%$ of QoCD original value). Another example is the case of JHotDraw in both experiments, where on average 7 packages ($\approx 18\%$ of packages in the JHotDraw's original modularization) are merged to other packages, and the QoCD is decreased (degraded) by $\approx 23\%$.

Table II
Cohesion and Coupling improvements in the solutions produced by NSGA and SA.
The means (μ) of achieved improvements to cohesion and coupling measurements, and the delta values (Δ : NSGA - SA) obtained with two-tailed Wilcoxon tests ($\alpha = 0.05$).

	Cohesion									Coupling					
	δIPD			δIPCD			δMQ			δIPC			δIPCC		
	μ NSGA	μ SA	Δ	μ NSGA	μ SA	Δ	μ NSGA	μ SA	Δ	μ NSGA	μ SA	Δ	μ NSGA	μ SA	Δ
Global Opt.															
JHotDraw	-123.87	-81.10	**-49.00***	-249.27	-62.80	**-189.00***	+4.51	+3.41	**+1.16***	-37.27	-37.20	**-1.00**	-6.07	-3.80	**-2.00***
JBoss	-274.70	-233.66	**-44.00***	-161.60	-38.43	**-123.00***	+32.52	+30.83	**+1.93***	-105.53	-95.03	**-12.00***	-10.20	-5.40	**-5.00***
ArgoUML	-252.50	-378.70	*+125.85**	-604.23	-85.60	**-515.00***	+3.57	+3.52	**+0.06**	-144.27	-177.50	*+36.00**	-19.17	-4.90	**-15.00***
Hibernate	-333.90	-491.80	*+162.97**	-707.23	-367.10	**-346.34***	+14.00	+11.40	**+3.00***	-155.90	-255.10	*+100.00**	-43.40	-32.60	**-10.00***
Controlled Opt.															
JHotDraw	-148.23	-68.30	**-81.00***	-179.50	-37.60	**-142.00***	+3.92	+3.04	**+0.89***	-27.50	-38.20	*+12.00**	-5.73	-3.00	**-3.00***
JBoss	-156.93	-209.40	*+52.00**	-92.27	-28.70	**-62.00***	+28.16	+28.64	*-0.55*	-79.87	-87.50	*+9.00**	-9.43	-4.10	**-5.00***
ArgoUML	-242.10	-250.70	*+9.00*	-544.47	-32.30	**-516.00***	+4.80	+3.57	**+1.00***	-122.37	-146.20	*+23.81**	-14.37	-4.00	**-10.00***
Hibernate	-206.97	-257.80	*+55.00**	-777.93	-134.30	**-664.30***	+7.33	+7.22	**+0.06**	-148.56	-178.80	*+32.00**	-48.50	-17.70	**-31.00***

* denotes statistically significant deltas at $\alpha = 0.05$. Delta values that are in **bold-face** (*italic-face*) denote that **NSGA performed better than SA** (*SA performed better than NSGA*), with regard to the corresponding measurement/property.

Table III
Package Size and Class Distribution property in the solutions produced by NSGA and SA.
The means (μ) of improvement/degradation to package size measurements, and the delta values (Δ : NSGA - SA) obtained with two-tailed Wilcoxon tests ($\alpha = 0.05$).

	Empty Packages ($\mathcal{P}_\phi$)			δQoCD		
	μ NSGA	μ SA	Δ	μ NSGA	μ SA	Δ
Global Opt.						
JHotDraw	00.00	6.70	**-7.00***	+0.04	-0.10	**+0.12***
JBoss	00.00	36.50	**-37.00***	+0.01	-0.04	**+0.05***
ArgoUML	00.00	32.10	**-32.00***	0.00	-0.06	**+0.06***
Hibernate	00.00	63.00	**-62.50***	0.00	-0.09	**+0.10***
Controlled Opt.						
JHotDraw	00.00	6.80	**-7.00***	+0.03	-0.08	**+0.11***
JBoss	00.00	35.50	**-36.00***	+0.01	-0.04	**+0.04***
ArgoUML	00.00	27.10	**-27.00***	0.00	-0.05	**+0.05***
Hibernate	00.00	46.20	**-46.00***	0.00	-0.06	**+0.06***

* denotes statistically significant deltas at $\alpha = 0.05$. Delta values that are in **bold-face** denote that **NSGA performed better than SA**.

Table IV
Moved Classes and the Rate per Refactoring of Achieved Improvement (RRAI) in the solutions produced by NSGA and SA
The means (μ) of modification amount measurements in produced solutions, and the delta values (Δ : NSGA - SA) obtained with two-tailed Wilcoxon tests ($\alpha = 0.05$).

	Moved Classes			RRAI		
	μ NSGA	μ SA	Δ	μ NSGA	μ SA	Δ
Global Opt.						
JHotDraw	29.87	36.90	**-5.00***	7.53	3.45	**+4.13***
JBoss	361.50	162.53	*+201.00**	2.30	3.11	*-0.85**
ArgoUML	100.93	186.00	**-80.00***	6.00	1.47	**+4.03***
Hibernate	189.43	336.90	**-151.63***	3.52	1.56	**+1.94***
Controlled Opt.						
JHotDraw	24.43	26.00	**-1.00***	7.66	4.03	**+3.63***
JBoss	139.93	141.00	**-9.05***	4.54	3.05	**+1.48***
ArgoUML	78.57	119.00	**-46.00***	6.63	1.80	**+4.83***
Hibernate	150.90	157.00	**-2.00***	4.30	1.83	**+2.48***

* denotes statistically significant deltas at $\alpha = 0.05$. Delta values that are in **bold-face** denote that **NSGA performed better than SA**.

This degradation of package size and class distribution property in SA's solutions explains why the SA approach succeeded in reducing IPC and IPD more than the NSGA approach in some cases. Additionally, these results show that our approach succeeds in optimizing the quality of existing packages without merging small packages into larger ones.

D. Achieved Optimization vs. Applied Modifications

NSGA. Table IV shows that a relatively small number of classes changed their packages in NSGA's solutions, compared to the number of reduced dependencies, connections and cycles among packages (Table II). For example, in the global optimization experiment, where 100% of applications' classes can change their packages, NSGA moved on average the following numbers (percentages) of classes: $\approx$ 30 ($\approx$ 5.8%) for JHotDraw, $\approx$ 362 ($\approx$ 12.8%) for JBoss, $\approx$ 101 ($\approx$ 4.3%) for ArgoUML, $\approx$ 190 ($\approx$ 6.1%) for Hibernate. Moreover, Table IV shows that the mean of RRAI values in NSGA's solutions is, in the worst case (JBoss case in the first experiment), strictly twice as large than the baseline value (which is 1). Hence, for all case-studies the average contribution of a moved class to the improvement of package cohesion and coupling is considerably important. This is an empirical evidence that NSGA does relatively small modifications in original modularizations.

NSGA compared to SA. By comparing the number of moved classes and the RRAI values in SA's solutions to those in NSGA's solutions, we observe the following. For all case-studies, except for JBoss in the first experiment: (1) the number of moved classes with NSGA is smaller than that with SA, and (2) the rate per refactoring of achieved improvement to package cohesion and coupling is considerably more important with NSGA than with SA. We believe that the exception of JBoss case in the first experiment is mainly due to the following: NSGA preferred to move a larger number of classes than SA, rather than merging a large set of small packages into larger ones (see the results about package size property in Section VII-C). However, in the controlled optimization experiment, where the refactoring size is limited to only 5% of the application classes, NSGA performed better than SA even in the case of JBoss application.

Results Summary

The results show significant evidence that the presented multi-objective approach, NSGA, outperforms the existing single-objective approach, SA, in improving exiting packages structure. Indeed, the empirical results show clearly that our optimization approach, NSGA, can well improve package cohesion and reduce package coupling and cycles by doing very small modifications in existing modularizations. That is without merging small packages into larger ones and without degrading the quality of class distribution, and finally, despite the extra constraints on refactoring size and on class locality.

VIII. Discussion

This section discusses the contributions and limitations of our optimization approach, and contrasts it from existing approaches on the software re-modularization problem.

A. Contributions

The contribution of this paper with respect to the existing work on automated optimization of packages structure (e.g., [11], [2], [7], [8]), is that this paper proposes a rich, open, iterative and extensible multi-objective optimization approach.

a) Rich: our approach uses a rich model for evaluating the quality of packages structure. It can improve all of package cohesion, coupling and cycles, and this not at cost of each other or at cost of package size and class distribution. Moreover, it attempts to minimize the size of proposed refactorings with regard to the achieved improvements. Hence, our approach attempts to minimize the perturbations of the design factors of original modularizations. This, in its turn, should reduce the effort for understanding and validating produced solutions. To the best of our knowledge, up to date there is no existing remodularization/optimization approach that considers all those objectives (see Section IV).

b) Open: thanks to the openness of our approach to different kinds of constraints, maintainers can empower the optimization process to produce acceptable packages structure, with regard to different design factors (e.g., see the setup of the conducted experiment in Section VI-C). The contribution of our optimization approach over existing remodularization approaches that allow end-users to interact with the remodularization process (e.g., [7]) is the following. First of all, our approach is an optimization approach rather than remodularization one. Additionally, the constraints used by our approach are generic enough to cover a large variety of contexts. Unlike the interactive approach in [7] which requests feedback of end-users in every step of the evolution process, end users should setup their constraints only once before starting the optimization process of our approach. We believe that this would facilitate the end user tasks. In this paper, we tested our approach in a context where classes can move only to packages that are directly related to them via explicit dependencies. However, depending on the preference of decision makers, the class-friends rule can be specified differently with different groups of classes. For example, inspired by the optimization approach of [11], the class-friends rule can be aligned to the semantic similarity between classes. In this way, our approach is expected to increase the structural cohesion of packages whilst it takes into account the semantic similarity between classes belonging to the same package. Nevertheless, further investigations for assisting maintainers in defining the class-friends rule, as well as further validations of our approach with different rules of class-friends, are desired.

c) Iterative: moreover, maintainers can use the approach to optimize existing packages structure in an iterative way. Maintainers can set the d_{max} value to a very small number/percent of the application classes (e.g., $d_{max} = 1\%$) and/or limit the refactoring space to some subsystem(s) –by freezing all packages outside that (or those) subsystem(s). In this way, the optimization process will try to optimize the cohesion and coupling properties of packages inside and outside that subsystem –since the evaluation model still takes into account the quality of the whole modularization. At the end of the optimization process, maintainers can investigate the produced solutions and select one of them. After that, maintainers can redefine their constraints and restart the optimization process on the selected solution, and so on. However, even though maintainers should investigate produced solutions at the end of each optimization step, this task would be relatively easy thanks to the limited/controlled size/space of refactorings. Still, as a future work, we plan to investigate our aforementioned claim by evaluating our approach with developers.

d) Extensible: thanks to the multi-objective formulation of the optimization approach, our approach can be easily extended with different objectives. For example, the objective for assessing the quality of class distribution (Obj.4) can be replaced by another objective for assessing the semantic cohesion of packages. This way, the optimization approach will be expected to improve the quality of packages structure without degrading the semantic cohesion of packages. Furthermore, our approach can also be easily extended with additional objectives (i.e., new metrics for assessing further properties of packages). However, further investigations will be desired to assess the performance of the optimization approach with different/additional objectives.

B. Main limitations

Now that we have presented the contributions of our approach, the following discusses its main limitations.

e) It is an optimization approach: our approach does not address the problem of modularizing/decomposing software systems from scratch (e.g., module clustering [2]). Moreover, it is not intended to be used for a radical restructuring of packages (e.g., decomposing large packages [7]). Rather, it aims at assisting software maintainers in the task of improving the quality of existing packages structure by doing as less as possible of modifications in it.

f) It is limited to moving classes among packages: our approach does not take into account the design quality at the class level. In fact, in some cases, high coupling between packages and/or low cohesion of some packages can be due to design defects at class level [20]: e.g., a *God class* that points a large number of dependencies to *data classes* packaged in different packages may be the cause of a high coupling between packages; and the existence of a large number of *data classes* may be the cause of low cohesion of their packages. In such cases, it would be better to detect and correct the class design defects (e.g., [10]) before using our optimization approach. However, to the best of the authors' knowledge, up to date there is no research effort on the problem of improving the quality of software structure at both levels (classes and packages) at the same time.

g) It is a semi-automated optimization approach: it is important to note that we do *not* claim that the optimization approach *automatically* improves packages structure with regard to every design factor. As outlined in earlier sections, this work addresses the problem of improving packages structure with regard to specific design principles, which are CCP, CRP and ADP (Section III). However, the optimization approach attempts to minimize the perturbation of other design factors that are involved in original modularizations. In our approach, software maintainers should specify, according to their preferences, their own constraints on the other design factors. As a consequence, in addition to increasing cohesion and decreasing coupling, our approach can attempt to propose solutions that are meaningful from a developer's point of view. Still, as a future work, we plan to perform further experiments with qualitative analysis of proposed solutions, aiming at investigating the suggested refactorings.

C. Threats to Validity

As a matter of fact, the external validity concerns arise from the use of a limited set of software projects. However, the study reported in this paper is concerned with an abstract representation of software systems (their Package Dependency Graphs: Figure 1). Since many software systems could have the same package dependency graph, the results reported in this paper can thus be automatically generalized to a wider range of software systems: all systems that have similar package dependency graphs to that of the case-studies. As a consequence, this might considerably mitigate the external threats to validity of the reported experiments.

The threats to internal validity of our experiments concern the used independent variables (e.g., IPD, IPCD, MQ, etc.). In these experiments, the values of all used measurements are computed by a static analysis tool. Due to programming-language dynamic mechanisms such as polymorphism and dynamic class loading, the derived values of the independent variables could be under or over-estimated. In fact, for the construct validity also, the package dependency graphs of the considered applications can differ from one analysis tool to another. This can be due to the analysis approach of the used tool (statistic or dynamic) and/or the hypotheses made when extracting the dependencies between software classes.

IX. Conclusion and Future Work

This paper presents a multi-objective optimization approach for improving existing packages structure with regard to well-known cohesion and coupling principles of package design. To limit the perturbation of the other design factors that are involved in the original packages organization, the optimization process minimizes the modification amount on original modulaizations with regard to the achieved improvement of package cohesion and coupling. Furthermore, the optimization process avoids improving of package cohesion and coupling at cost of merging small packages into larger ones and/or increasing the size of large packages. It also considers different types of constraints that maintainers can define to guide the optimization process with regard to other design factors than cohesion and coupling.

To evaluate the optimization approach we performed a comparative study with the Simulated Annealing optimization approach proposed by Abdeen *et al.* [8], covering four large object-oriented applications that have radically different modularizations. The results provide an empirical evidence on the efficiency of our optimization process to improve existing packages structure by moving very small number of classes from their original packages. These results are important since they were obtained without degrading the quality of class distribution over packages and without merging small packages into larger ones. Furthermore, they were obtained with the consideration of different constraints that were specified with regard to other design factors different than cohesion, coupling and cycles.

As future work, we intend to extend the optimization approach with additional objectives. For example, future work can consider semantic cohesiveness within packages. We also plan to set up a new study to validate empirically the improvements to the resulting source code structures.

Acknowledgment

This publication was made possible by NPRP grant #09-1205-2-470 from the Qatar National Research Fund (a member of Qatar Foundation).

References

[1] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Trans. on Soft. Eng.*, vol. 32, no. 3, pp. 193–208, 2006.

[2] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. on Soft. Eng.*, vol. 37, no. 2, pp. 264–282, Mar. 2011.

[3] R. C. Martin, *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

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

[5] H. Melton and E. Tempero, "The crss metric for package design quality," in *Proceedings of ACSC' 2007*. Australian Computer Society, Inc., 2007, pp. 201–210.

[6] F. B. e Abreu and M. Goulao, "Coupling and cohesion as modularization drivers: are we being over-persuaded?" in *Proceedings of CSMR' 2001*, Mar. 2001, pp. 47–57.

[7] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Software re-modularization based on structural and semantic metrics," in *Proceedings of WCRE' 2010*, 2010, pp. 195–204.

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

[9] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.

[10] A. Ouni, M. Kessentini, H. A. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 47–79, 2013.

[11] G. Bavota, F. Carnevale, A. D. Lucia, M. D. Penta, and R. Oliveto, "Putting the developer in-the-loop: An interactive ga for software re-modularization," in *Proceedings of SSBSE' 2012*, 2012, pp. 75–89.

[12] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *Proceedings of ICSM' 2012*, 2012, pp. 472–481.

[13] N. Anquetil and J. Laval, "Legacy software restructuring: Analyzing a concrete case," in *Proceedings of CSMR' 2011*. IEEE Computer Society Press, 2011, pp. 279–286.

[14] H. Abdeen, S. Ducasse, D. Pollet, and I. Alloui, "Package fingerprints: A visual summary of package interface usage," *Information and Software Technology*, vol. 52, no. 12, pp. 1312–1330, Dec. 2010.

[15] B. Meyer, *Object success: a manager's guide to object orientation, its impact on the corporation, and its use for reengineering the software process*. Prentice-Hall, Inc., 1995.

[16] S. Mancoridis and B. S. Mitchell, "Using Automatic Clustering to produce High-Level System Organizations of Source Code," in *Proceedings of IWPC' 1998 (International Workshop on Program Comprehension)*. IEEE Computer Society Press, 1998.

[17] H. Ishibuchi, N. Tsukamoto, and Y. Nojima, "Evolutionary many-objective optimization: A short review," in *IEEE Congress on Evolutionary Computation*, 2008, pp. 2419–2426.

[18] H. Sato, H. E. Aguirre, and K. Tanaka, "Local dominance and controlling dominance area of solutions in multi and many objectives eas," in *Proceedings of GECCO' 2008*. ACM, 2008, pp. 1811–1814.

[19] R. C. Martin, "The dependency inversion principle," *C++ Report*, 1996.

[20] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proceedings of CSMR' 2011*. IEEE Computer Society, 2011, pp. 181–190.