Automatic Package Coupling and Cycle Minimization

Hani Abdeen, Stéphane Ducasse, Houari Sahraoui, Ilham Alloui

Automatic Package Coupling and Cycle Minimization

Hani Abdeen*, Stéphane Ducasse*, Houari Sahraoui† and Ilham Alloui‡
*RMoD team, INRIA - Lille Nord Europe, USTL - CNRS UMR 8022, Lille, France
Email: {fname.name}@inria.fr
†DIRO, Université de Montréal, Montréal(QC), Canada
Email: sahraoui@iro.umontreal.ca
‡LISTIC, Université de Savoie, Annecy, France
Email: Ilham.Alloui@univ-savoie.fr

***Abstract*—Object-oriented (OO) software is usually organized into subsystems using the concepts of package or module. Such modular structure helps applications to evolve when facing new requirements. However, studies show that as software evolves to meet requirements and environment changes, modularization quality degrades. To help maintainers improve the quality of software modularization we have designed and implemented a heuristic search-based approach for automatically optimizing inter-package connectivity (*i.e.*, dependencies). In this paper, we present our approach and its underlying techniques and algorithm. We show through a case study how it enables maintainers to optimize OO package structure of source code. Our optimization approach is based on Simulated Annealing technique.**

***Keywords*-Reverse engineering; Re-engineering; Software modularization; Search algorithms**

I. Introduction

In object-oriented languages such as Java, Smalltalk and C++, package structure allows people to organize their programs into subsystems. A well modularized system enables its evolution by supporting the replacement of its parts without impacting the complete system. A good organization of classes into identifiable and collaborating subsystems eases the understanding, maintenance, test and evolution of software systems [6].

However code decays: as software evolves over time with the modification, addition and removal of new classes and dependencies, the modularization gradually drifts and looses quality [8]. A consequence is that some classes may not be placed in suitable packages [12]. To improve the quality of software modularization, optimizing the package structure and connectivity is required.

Software modularization is a graph partitioning problem [27], [28]. Since this last is known as a NP-hard problem [9], searching for good modularization by using deterministic procedures or exhaustive exploration of the search space is not feasible without additional heuristics [4], [27]. We chose then an alternative approach based on heuristic search procedures to identify a good solution within a reasonable amount of computing time [10].

Heuristic search methods have already been successfully applied to the software modularization problem [5], [13], for example, to automatically decompose and modularize software systems. They are mainly based on clustering [1], [2], [18], [22], [21], [32], [28] and evolutionary algorithms [7], [14], [17], [19], [20], [26], [27], [28], [31].

Few of these works address the problem of optimizing software modularization [14], [27], [28], [31], [33]. Often existing approaches change (to various degrees) the existing package structure of an application. In such a case, it can be difficult for a software engineering to understand the resulting structure and to map it back to the situation he knows. Our approach is to support remodularisation of existing package structure by explicitly taking it into account and avoiding creating new package or related abstractions.

In this paper, we present an approach for automatically optimizing existing software modularizations by minimizing connectivity among packages, in particular cyclic-connectivity. The objective of the optimization process is inspired by well known package $cohesion$ and $coupling$ principles already discussed in [3], [11], [23]. We limit ourselves to $direct$ cyclic-connectivity and restrict our optimization actions to moving classes over existing packages. By definition, there is a direct cyclic-connectivity between two packages if they mutually depend on each other.

Our approach is based on *Simulated Annealing* [10], [16], which is a *neighborhood (local) search-based* technique. Simulated Annealing is inspired by the annealing process in metallurgy [16]. We chose this technique because, it suits well our problem, *i.e.*, local optimization of an existing solution. Moreover, it has been shown to perform well in the context of automated OO class design improvement [29], [30] and more generally, in the context of software clustering problems [25], [27].

Contribution: we present an approach, using simulated annealing technique, for automatically reducing package coupling and cycles by only moving classes over packages while taking into account the existing class organization and package structure. In our approach, maintainers can define (1) the maximal number of classes that can change their packages, (2) the maximal number of classes that a package can contain, and (3) the classes that should not change their packages or/and the packages that should not be changed.

Section II defines the terminology we use and gives an overview about challenges in optimizing package structure. Section III presents the metrics used in evaluation functions of both modularization and package quality. We detail our optimization algorithm in Section IV. We validate our approach using real large applications and discuss the results in Section V. In Section VI we position our approach with related works, before concluding in Section VII.

II. Background and Vocabulary

In this section, we introduce the terminology used in this paper and give an overview of some challenging issues we address.

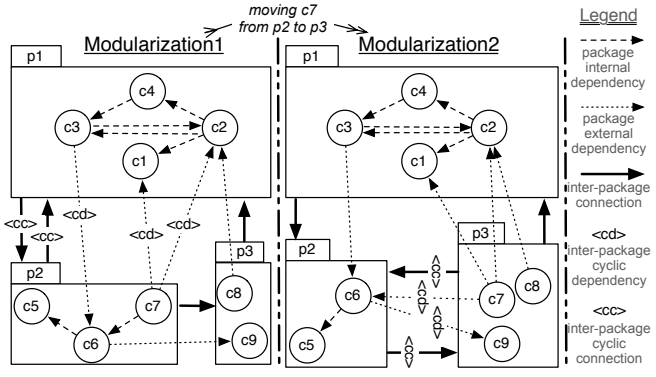

Figure 1. Example of two modularizations: different decompositions of the set of classes $[c1..c9]$ into 3 packages $[p1, p2, p3]$.

A. Terminology

Before all, we define OO software *Modularization* $\mathcal{M}$ as a decomposition of the set of software classes $\mathcal{M}_C$ into a set of packages $\mathcal{M}_P$. Fig. 1 shows two modularizations, both consists of 9 classes distributed over 3 packages: where *Package* represents an entity that only contains classes, where every class c belongs to only one package c_p. We define package size p_{size} by the number of its classes.

Every class c can be related to other classes, in consequence to their packages, through a set of *Dependencies* (c_D). This set consists of two subsets: *Outgoing Dependencies* ($c_{Out.D}$) and *Incoming Dependencies* ($c_{Inc.D}$). We denote a dependency that goes from a class c_i to another one c_j by the pair (c_i, c_j). A dependency might be either method calls, class access, or class inheritance.

Every dependency is either $internal$ if it is related to two classes belonging to the same package, or $external$ if not. The set of dependencies related to a package p (p_D) represents the union of the set of internal dependencies ($p_{Int.D}$) and the set of external dependencies ($p_{Ext.D}$) relatively to p_C, the set of p classes. The set $p_{Ext.D}$ consists of two subsets: dependencies that are either exiting p ($p_{Ext.Out.D}$) or that are pointing to p ($p_{Ext.Inc.D}$). The sets $p_{Ext.Out.D}$ and $p_{Ext.Inc.D}$ respectively relate p to its provider packages ($p_{Pro.P}$) and its client packages ($p_{Cli.P}$). $Modularization_1$ in Fig. 1 shows that there is one dependency exiting $p1$ and three dependencies pointing to $p1$: $p1_{Ext.Out.D} = [(c_3, c_6)]$; $p1_{Ext.Inc.D} = [(c_7, c_1), (c_7, c_2), (c_8, c_2)]$. It also shows that $p1$ has one provider package and two client packages: $p1_{Pro.P} = [p2]$; $p1_{Cli.P} = [p2, p3]$.

To determine connectivity at the package level, we say that there is a *Connection* from a package y to another one z if there is n ($n > 0$) outgoing dependencies y pointing to z. $Modularization_1$ in Fig. 1 shows that there are two connections going from $p2$ to its provider packages: $p2_{Out.Con} = [(p2, p1), (p2, p3)]$; and there is one connection going to $p2$: $p2_{Inc.Con} = [(p1, p2)]$.

Dependencies can form direct cyclic dependencies/connections between packages. If package y plays the role of client and provider for package z then we consider involved dependencies and connections between y and z as cyclic. We denote the set of cyclic dependencies related to a package p by $p_{Cyc.D}$. The set $p_{Cyc.D}$ consists also of two subsets: cyclic-dependencies that are either exiting p ($p_{Out.Cyc.D}$) or pointing to p ($p_{Inc.Cyc.D}$). $p_{Out.Cyc.D}$ and $p_{Inc.Cyc.D}$ are dependencies causing cycles between packages (and not classes) in the context of the client-provider relation. Similarly we denote the set of cyclic connections related to p by $p_{Cyc.Con}$: $p_{Cyc.Con} = p_{Out.Cyc.Con} \cup p_{Inc.Cyc.Con}$. $Modularization_1$ in Fig. 1 shows that $p1$ has one outgoing cyclic-dependency $[(c_3, c_6)]$ and two incoming cyclic-dependencies $[(c_7, c_1), (c_7, c_2)]$. Those cyclic-dependencies produce two cyclic-connections: $p1_{Cyc.Con} = p2_{Cyc.Con} = [(p1, p2), (p2, p1)]$.

B. Challenges in Optimizing Package Structure

Optimizing package structure in existing large applications is a difficult problem because of the following reasons:

Large applications are usually very complex: they contain thousands of heavily inter-connected classes. Many of the dependencies are between classes belonging to different packages, which increases the inter-package connectivity. In such situation, the optimization problem is more difficult.

Classes usually are not well distributed over packages: in real applications, some packages contain large sets of classes and other packages contain few number of classes. As consequence, most application packages depend on those large packages. Furthermore, it is difficult to determine an ideal package size since it may depend on external factors such as the team structure, domain, or coding practice...

Optimizing some criteria may degrade others: minimizing inter-packages dependencies/connections may increase the number of noncyclic ones. *e.g.*, Fig. 1 shows two modularizations, $Modularization_1$ and $Modularization_2$. Both are composed of the same set of classes and packages. The difference between them is c_7 that is moved from $p2$ to $p3$ in $Modularization_2$. In $Modularization_2$ there are 2 cyclic-dependencies $[(c_7, c_6), (c_6, c_9)]$ compared to 3 in $Modularization_1$ $[(c_3, c_6), (c_7, c_1), (c_7, c_2)]$. Thus moving c_7 has reduced the number of cyclic-dependencies. On the

other hand, moving c_7 increases the number of inter-package dependencies. In $Modularization_2$, there are 6 inter-package dependencies compared to 5 for $Modularization_1$.

III. Modularization Quality

Our goal is to automatically optimize the decomposition of software system into packages so that the resulting organization of classes/packages, *mainly*, reduces connectivity and cyclic-connectivity between packages. This goal is inspired from well known quality principles already discussed in [3], [11], [23] and in particular from the following principle: *packages are desired to be loosely coupled and cohesive to a certain extent* [11]. In such a context, we need to define measures that evaluate package *cohesion* and *coupling*.

In addition, *cyclic dependencies* between packages are considered as an anti-pattern for package design [23].
In this section we define two suites of measures: the first is used when evaluating modularization quality and the second is used when evaluating modularity quality of single package. Note that all measures we define in this section take their value in the interval $[0..1]$ where 1 is the optimal value and 0 is the worst value.

A. Measuring Modularization Quality

Inter-Package Dependencies: according to Common Closure Principle (CCP) [23], *classes that change together should be grouped together*. In such a context, optimizing modularization requires reducing the sum of inter-package dependencies ($IPD = \sum_{i=1}^{|\mathcal{M}_P|} |p_{i_{Ext.Out.D}}|$) [3], [11]. Since we do not change the dependencies between classes during our optimization process, we use the sum of inter-class dependencies ($ICD = \sum_{j=1}^{|\mathcal{M}_C|} |c_{j_{Out.D}}|$) as normalizer. We define the measure $\mathcal{CCQ}$ to evaluate the Common Closure Quality of a modularization $\mathcal{M}$ as follows:

$$\mathcal{CCQ}(\mathcal{M}) = 1 - \frac{IPD}{ICD}$$

Inter-Package Connections: according to Common Reuse Principle (CRP) [23], *classes that are reused together should be grouped together*. In such a context, optimizing modularization requires reducing the sum of inter-package connections ($IPC = \sum_{i=1}^{|\mathcal{M}_P|} |p_{i_{Out.Con}}|$) [3], [11]. We define the measure $\mathcal{CRQ}$ to evaluate the Common Reuse Quality of a modularization $\mathcal{M}$ as follows:

$$\mathcal{CRQ}(\mathcal{M}) = 1 - \frac{IPC}{ICD}$$

Inter-Package Cyclic-Dependencies: according to Acyclic Dependencies Principle (ADP) [23], *dependencies between packages must not form cycles*. In such a context, optimizing modularization requires reducing the sum of inter-package cyclic-dependencies ($IPCD = \sum_{i=1}^{|\mathcal{M}_P|} |p_{i_{Out.Cyc.D}}|$). We define the measure $\mathcal{ADQ}$ to measure the Acyclic Dependencies Quality of a modularization $\mathcal{M}$ as follows:

$$\mathcal{ADQ}(\mathcal{M}) = 1 - \frac{IPCD}{ICD}$$

Inter-Packages Cyclic-Connections: as for cyclic dependencies between packages, reducing cyclic connections between packages is required.
For example, in $Modularization_1$ in Fig. 1, there are 3 cyclic dependencies $[(c_3, c_6), (c_7, c_1), (c_7, c_2)]$ and 2 cyclic connections $[(p1, p2), (p2, p1)]$; moving c_7 to p_3 will reduce the number of cyclic-dependencies: in $modularization_2$ there are only 2 cyclic dependencies $[(c_6, c_9), (c_7, c_6)]$, but it remains 2 cyclic connections $[(p2, p3), (p3, p2)]$. We thus deduce that reducing inter-package cyclic dependencies does not necessarily reduce inter-package direct cyclic-connections ($IPCC = \sum_{i=1}^{|\mathcal{M}_P|} |p_{i_{Out.Cyc.Con}}|$).
We define the measure $\mathcal{ACQ}$ to evaluate the Acyclic Connections Quality of a modularization $\mathcal{M}$ as follows:

$$\mathcal{ACQ}(\mathcal{M}) = 1 - \frac{IPCC}{ICD}$$

B. Measuring Package Quality

In addition to measures presented in Section III-A, we define a set of measures that help us determine and quantify the quality of a single package within a given modularization. To normalize the value of those measures we use the number of dependencies related to the considered package ($|p_D|$) with $|p_D| > 0$.

Package Cohesion: we relate package cohesion to the direct dependencies between its classes. In such a context, we consider that the cohesion of a package p is proportional to the number of internal dependencies within p ($|p_{Int.D}|$). This is done according to the Common Closure Principle (CCP) [23]. We define the measure of package cohesion quality similarly to that in [1] as follows:

$$CohesionQ(p) = \frac{|p_{Int.D}|}{|p_D|}$$

Package Coupling: we relate package coupling to its efferent and afferent coupling (Ce, Ca) as defined by Martin in [24]. Package Ce *is the number of packages that this package depends upon* ($|p_{Pro.P}|$). Package Ca *is the number of packages that depend upon this package* ($|p_{Cli.P}|$). According to the common reuse principle, we define the measure of package coupling quality using the number of package providers and clients as follows:

$$CouplingQ(p) = 1 - \frac{|p_{Pro.P} \cup p_{Cli.P}|}{|p_D|}$$

Package Cyclic-Dependencies: for automatically detecting packages that suffer from direct-cyclic dependencies we define a simple measure that evaluates the quality of package cyclic dependencies ($CyclicDQ$) using the number of package cyclic dependencies:

$$CyclicDQ(p) = 1 - \frac{|p_{Cyc.D}|}{|p_D|}$$

Similarly we define another measure that evaluates package cyclic connections quality ($CyclicDQ$) using the number of package cyclic connections:

$$CyclicCQ(p) = 1 - \frac{|p_{Cyc.Con}|}{|p_D|}$$

IV. Optimization Technique (Methodology)

To optimize package connectivity, we use an optimization procedure that starts with a given modularization and gradually modifies it, using small perturbations. At each step, the resulting modularization is evaluated to be possibly selected as an alternative modularization. The evaluation of modularization quality is based on measures defined in Section III-A. This section describes our optimization approach and algorithm.

A. Technique Overview

To address the problem of optimizing modularization, we use heuristic optimization technique based on simulated annealing algorithm [10], [16]. Simulated annealing is an iterative procedure that belongs to the category of Neighborhood Search Techniques (NST).

Algorithm 1 Optimization Algorithm

Require: $T_{stop} \geq 1$, $T_{start} > T_{stop}$, $num > 1$ and $\mathcal{M}_{original}$
Ensure: $\mathcal{M}_{best}$
  $\mathcal{M}_{best} \leftarrow \mathcal{M}_{original}$
  $\mathcal{M}_{current} \leftarrow \mathcal{M}_{best}$
  $T_{current} \leftarrow T_{start}$
  –starting global search–
  while $T_{current} > T_{stop}$ **do**
    –starting local search–
    for $i = 1$ to num **do**
      –generating a new modularization and evaluating it–
      $\mathcal{M}_{trial} \leftarrow Neighborhood(\mathcal{M}_{current})$
      if $\mathcal{F}(\mathcal{M}_{trial}) > \mathcal{F}(\mathcal{M}_{current})$ **then**
        $\mathcal{M}_{current} \leftarrow \mathcal{M}_{trial}$
        if $\mathcal{F}(\mathcal{M}_{current}) > \mathcal{F}(\mathcal{M}_{best})$ **then**
          $\mathcal{M}_{best} \leftarrow \mathcal{M}_{current}$
        end if
      else if $AcceptanceCondition$ **then**
        –accepting a worse modularization–
        $\mathcal{M}_{current} \leftarrow \mathcal{M}_{trial}$
      end if
    end for
    –end of local search–
    $T_{current} \leftarrow CoolingSchedule(T_{current})$
  end while
  –end of global search–
  Return $\mathcal{M}_{best}$.

Algorithm 1 shows an overview of the optimization algorithm. The optimization process performs series of local searches with the global search parameter $T_{current}$. $T_{current}$ represents in simulated annealing technique the current temperature of the annealing procedure which started with the value T_{start}. A local search consists of num ($num \geq 1$) searches of suboptimal solution. At each of them, a new modularization $\mathcal{M}_{trial}$ is derived from a current one $\mathcal{M}_{current}$ by applying to this later a modification. The derivation of $\mathcal{M}_{trial}$ from $\mathcal{M}_{current}$ is performed by the *neighborhood* function. Then, the algorithm evaluates the $\mathcal{M}_{trial}$ and $\mathcal{M}_{current}$ *fitness* using the *fitness* function $\mathcal{F}$, where the bigger is the value of $\mathcal{F}(\mathcal{M})$, the better is modularization $\mathcal{M}$: if $\mathcal{M}_{trial}$ is better than $\mathcal{M}_{current}$ then $\mathcal{M}_{trial}$ becomes the $\mathcal{M}_{current}$; then, if $\mathcal{M}_{current}$ is better than the current best modularization $\mathcal{M}_{best}$, $\mathcal{M}_{current}$ becomes the $\mathcal{M}_{best}$. At the end of each local search, the parameter $T_{current}$ decreases and another local search starts with the new value of $T_{current}$. Decreasing $T_{current}$ is the responsibility of *CoolingSchedule* function. This latter is defined according to Keeffe *etal.* discussion [30] using a geometric cooling scheme: $CoolingSchedule(T) = 0.9975 * T$. Local searches are repeated until reaching T_{stop} ($T_{current} \leq T_{stop}$).

To circumvent the problem of *local optima* [10], a less-good modularization can be accepted with some probability: a less-good modularization $\mathcal{M}_{trial}$ can replace $\mathcal{M}_{current}$ under some conditions *AcceptanceCondition*. Simulated annealing technique defines acceptance conditions in a way that the probability of accepting a less-good modularization decreases over time. We define *AcceptanceCondition* as follows: $r > e^{-\frac{T_{current}}{T_{start}}}, r \in [0..1]$. The value of r is generated randomly in the interval $[0..1]$. The function $e^{-\frac{T_{current}}{T_{start}}}$ takes its value in the interval $[0..1]$ $\forall T_{current} \geq 0$, and $T_{current} \leq T_{start}$. It increases along the optimization process –since $T_{current}$ decreases. By doing so, the probability of accepting a less-good modularization decreases over time.

B. Evaluating Modularization Quality (Fitness)

As for any search-based optimization problem, the definition of the fitness function represents a central concern as it guides the search. We define our fitness function as a combination of the measures defined in Section III-A. We define dependency quality ($\mathcal{DQ}$) for a modularization $\mathcal{M}$ as the weighted average of *Common Closure Quality* ($\mathcal{CCQ}$) and *Acyclic Dependencies Quality* ($\mathcal{ADQ}$); and we define connection quality ($\mathcal{CQ}$) for $\mathcal{M}$ as the weighted average of *Common Reuse Quality* ($\mathcal{CRQ}$) and *Acyclic Connections Quality* ($\mathcal{ACQ}$). To give higher intention to cyclic dependencies/connections between packages we define a factor of importance γ ($\gamma = \frac{\beta}{\alpha}, \beta > \alpha \geq 1$):

$$\mathcal{DQ}(\mathcal{M}) = \frac{\alpha * \mathcal{CCQ}(\mathcal{M}) + \beta * \mathcal{ADQ}(\mathcal{M})}{\alpha + \beta}$$

$$\mathcal{CQ}(\mathcal{M}) = \frac{\alpha * \mathcal{CRQ}(\mathcal{M}) + \beta * \mathcal{ACQ}(\mathcal{M})}{\alpha + \beta}$$

Both functions $\mathcal{DQ}$ and $\mathcal{CQ}$ take their values in the interval $[0..1]$ where 1 is the optimal value. The final fitness function is defined by the average of $\mathcal{DQ}$ and $\mathcal{CQ}$:

$$\mathcal{F}(\mathcal{M}) = \frac{\mathcal{DQ}(\mathcal{M}) + \mathcal{CQ}(\mathcal{M})}{2}$$

Our hypothesis is: optimizing $\mathcal{F}$ will reduce inter-package dependencies and connections, particularly cyclic ones.

Furthermore, in addition to *AcceptanceCondition* for less-good modularizations we defined in Section IV-A, the optimization process may accept a less-good resulting modularization only if the number of inter-package dependencies decreases (*i.e.*, $\mathcal{DQ}$ increases). We expect such a decision facilitates the reduction of inter-package cyclic dependencies.

C. Modularization Constraints

In addition to the fitness function, our approach allows maintainers to define distinct constraints that should complete the evaluation process and guarantee maintainers' requirements. The rationale behind those constraints is to control the optimization process when optimizing a given modularization. *e.g.*, putting a major partition of classes into one package can effectively reduce inter-package (cyclic-) dependencies/connections; such an approach is clearly not the best one to optimize software modularization. This section presents three constraints allowing control the optimization process. Section IV-D explains how the optimization process favors these constraints when deriving new modularizations.

1) Controlling package size: to avoid having very large and dominant packages, we introduce the following constraint: the size of every package (p_{size}) should always be smaller than a predefined number ($size_{max}$). We define $size_{max}$ for every package p relatively to its size in the original modularization $p_{size_{V0}}$: $size_{max} = \delta + p_{size_{V0}}, \delta \geq 0$. Maintainers can define δ according to the context of the concerned software system. We cannot determine upfront the good interval in which δ should be taken. In the scope of this paper, we define δ as the *theoretical package size* in $\mathcal{M}_0$ which equals to the ratio: $\frac{|\mathcal{M}_{0_C}|}{|\mathcal{M}_{0_P}|}$. It is worth to note that maintainers can define a different δ for each package: *e.g.*, for a large package p, δ may be defined to 0; this way, p will never be larger than before.

2) Controlling modularization modification: maintainers should be able to define the *limit* of modifications that the optimization process can apply on the original modularization $\mathcal{M}_0$ when it proceeds. In other words, the optimization process must take into account the *maximal authorized distance* ($distance_{max}$) between resulting modularizations and $\mathcal{M}_0$. In our context, for two modularizations that entail the same set of classes, we define the $distance$ between them by the number of classes that changed packages. This way, $distance_{max}$ can be defined simply as the maximal number of classes that can change their packages.

3) Controlling modularization structure: moreover, we found that it is very helpful to allow maintainers decide whether some classes should not change their package and/or whether given packages should not be changed. We say that such classes/packages are *frozen*. This constraint is particularly helpful when maintainers know that a given package is well designed and should not be changed: *e.g.*, if a small package p contains a couple of classes that extend classes from other package, p may be considered a well designed package, even if it is not cohesive.

D. Deriving New Modularization (Neighbor)

The neighborhood function ($\mathcal{N}$) is the second main concern of the optimization process. Defining $\mathcal{N}$ requires: (1) the definition of the set of modifications that $\mathcal{N}$ can use to derive new modularizations, (2) and the definition of a process that derives a new modularization from another one. This section presents our definition of $\mathcal{N}$.

Since we search near optimal modularization by applying near minimal modification to the original modularization, we limit the set of modifications that $\mathcal{N}$ can use to only: moving a class c from its current package p_{source} to another one p_{target}. In this context, we say that c is the modification $actor$ (c_{actor}). To minimize search-space we reduce the selection-space of p_{target} to the set of client and provider packages of c_{actor}: $p_{target} \in (c_{actor_{Pro.P}} \cup c_{actor_{Cli.P}})$.

We specify the derivation of a neighbor modularization of a modularization $\mathcal{M}$ by 4 sequential steps: (1) selecting p_{source}, (2) selecting c_{actor}, (3) selecting p_{target} and then (4) moving c_{actor} to p_{target}. Selections in the first three steps are done arbitrary using a probability function. The probability function gives higher probability to the *worst* package into $\mathcal{M}_P$ to be selected as p_{source}, to the *worst* class into p_{source} to be selected as c_{actor} and to the *nearest* package to c_{actor} to be selected as p_{target}. The selection mechanism performs similarly to a roulette wheel selection, where each class/package is given a slice proportional to its probability to be selected and then we randomly take a position in the roulette and pick the corresponding class/package.

It is worth to note that packages and classes that are defined as *frozen* (Section IV-C3), do not belong to the selection spaces: a *frozen* package will never be a p_{source} or p_{target}, and a *frozen* class will never be a c_{actor}.

The following subsections explain our definition of the probability of being selected as p_{source}, c_{actor} or p_{target}. Note that in our definition of this probability we use the factor γ, as defined in the fitness function (Section IV-B), to pay more intention to cyclic dependencies/connections.

1) Selecting p_{source}: the worst package in $\mathcal{M}_P$ is, the highest probability to be selected it has. We relate package badness to the quality of its cohesion, coupling and of its external dependencies (*i.e.*, the density of cyclic dependencies/connections related to the concerned package). We define the badness of package by using the measures: *CohesionQ, CouplingQ, CyclicDQ* and *CyclicCQ* (Section III-B): where we relate *CohesionQ* and *CyclicDQ* to package dependency quality ($\mathcal{DQ}$). Also we relate *CouplingQ* and *CyclicCQ* to package connection quality ($\mathcal{CQ}$). We define package quality functions, *similarly to modularization quality functions defined in Section IV-B* :

$$\mathcal{DQ}(p) = \frac{\alpha * CohesionQ(p) + \beta * CyclicDQ(p)}{\alpha + \beta}$$

$$\mathcal{CQ}(p) = \frac{\alpha * CouplingQ(p) + \beta * CyclicCQ(p)}{\alpha + \beta}$$

Both functions $\mathcal{DQ}$ and $\mathcal{CQ}$ take their values in the interval $[0..1]$ where 1 is the optimal value.
Finally, we define package badness basing on the average of $\mathcal{DQ}$ and $\mathcal{CQ}$:

$$\mathcal{B}adness(p) = 1 - \frac{\mathcal{DQ}(p) + \mathcal{CQ}(p)}{2}$$

In addition to satisfy constraints discussed in Section IV-C, we define the probability of selecting a package p as p_{source} by: $\rho * \mathcal{B}adness(p)$, where ρ is a factor takes its value in the interval $[0..1]$. It is the average of two sub-factors (ρ_1, ρ_2):

- ρ_1 is based on p_{size}: relatively to p size in the original modularization $\mathcal{M}_0$ ($p_{0_{size}}$), a package whose size increased has a higher probability to be selected than a package whose size decreased. By doing so, we expect that the package size in resulting modularizations will be similar to that in the original modularization;
- ρ_2 is based on the number of new classes into p: relatively to p_0, a package that acquired the largest number of new classes (*i.e.*, classes are not packaged in p_0) has the highest probability to be selected. By doing so, we favor moving classes that already changed their packages until they find their optimal package.

2) Selecting c_{actor}: the worse a class in p_{source} is, the highest probability to be selected it has. We relate class badness to the number of external dependencies related to the class ($|c_{Ext.D}|$) and to the number of its external cyclic-dependencies (related to its package p):

$$\mathcal{B}adness(c) = \frac{\alpha * |c_{Ext.D}| + \beta * |c_{Ext.D} \cap p_{Cyc.D}|}{\alpha + \beta}$$

In addition, to satisfy the constraint of $distance_{max}$ (Section IV-C2), when the distance (d) between resulting modularizations and the original one increases, classes that have already changed their packages have higher probability to be c_{actor}. In this context, we use the factor $\rho = 1 - \frac{d}{distance_{max}}$. If $\rho \leq 0$ then only classes which already changed their original packages can move. Only if $0 < \rho \leq 1$ then the optimization process can move more classes over packages but with a probability ρ. Thus we define the probability of selecting a class c as c_{actor} as following:

$$\begin{cases} 0 & \rho \leq 0, not(isMoved(c)) \\ \rho * \mathcal{B}adness(c) & \rho > 0, not(isMoved(c)) \\ \mathcal{B}adness(c) & isMoved(c) \end{cases}$$

Where the predicate $isMoved(c)$ is true if c is already moved from its original package.

3) Selecting p_{target}: the nearest package to c_{actor} is, the highest probability to be selected it has. We simply relate the nearness of a package p to a class c to the number of dependencies that c has with p classes ($|c_D \cap p_D|$) and to the number of dependencies related to c and present cyclic dependencies between c_p and p ($|c_D \cap p_{Cyc.D}|$):

$$\mathcal{N}earness(p, c) = \frac{\alpha * |c_D \cap p_D| + \beta * |c_D \cap p_{Cyc.D}|}{\alpha + \beta}$$

To satisfy the constraint on p_{size} ($size_{max}$ defined in Section IV-C1), when package size increases its probability to be selected as $p_{targert}$ decreases. In this context, we use the factor $\rho = 1 - \frac{p_{size}}{size_{max}}$. If $\rho \leq 0$ then the package size should not increase anymore. Only if $0 < \rho \leq 1$ then the package size can increase but with a probability ρ which decreases when p_{size} increases. Thus we define the probability of selecting a package p as p_{target} for a class c, as following:

$$\begin{cases} 0 & \rho \leq 0 \\ \rho * \mathcal{N}earness(p, c) & 0 < \rho \leq 1 \end{cases}$$

V. Experiments and Validation

To validate our optimization approach, we applied it to several software applications that differ in terms of: number of classes ($|\mathcal{M}_C|$), number of packages ($|\mathcal{M}_P|$), number of inter-class dependencies (ICD); number of inter-package dependencies (IPD), connections (IPC), cyclic dependencies ($IPCD$) and cyclic connections ($IPCC$). Table I shows information about the original modularization of those software applications.

Since the search process is not deterministic, we applied our algorithm 10 times for each software application and we calculated the average of modularization parameters cited in Table I. We used the parameters T_{start}, T_{stop} and num (Algorithm 1) with value 50, 1 and 30 respectively. On another hand, we weighted cyclic dependencies/connections to be three times more important than noncyclic dependencies/connections. We performed our experience twice: the first time, we did not used the constraint $distance_{max}$. In the second time, we limited $distance_{max}$ to 5%, which means that only 5% of classes can change their original packages.

General optimization: Table III shows optimization results. In the resulting modularization for JEdit ($JEdit_1$), 10.2% of inter-package dependencies IPD, 23.3% of inter-package cyclic-dependencies $IPCD$, 24% of inter-package connections IPC and 37.2% of inter-package cyclic-connections $IPCC$ have been removed. This significant improvement of inter-package connectivity was obtained by moving only 8.9% of the classes ($d = 8.9\%$). Similarly for other case studies, the optimization process has improved original modularizations by moving a relatively small number of their classes. When limiting $distance_{max}$ to 5%, the algorithm obtained similar results.

Class distribution and package size: Table IV shows that some packages were $empty$ in resulting modularizations -since their classes moved to other packages. For example, in $ArgoUML_1$, 25.4% of packages were empty. By inspecting packages in the original modularization we found that those empty packages are packages which have originally very small sizes (*i.e.*, in average two or three classes) and have low quality for cohesion, coupling and/or cyclic dependencies. This conclusion were also true for the other case studies.

Fig. 2 shows an overview about package size and cohesion for the original modularization of $ArgoUML$ and for the resulting modularizations (Table III). We can see that empty packages in the resulting modularizations are packages whose sizes are small and whose cohesion is relatively worse. On another hand, Fig. 2 shows also that the size of some small packages, annotated by *small packages*, is increased in the resulting modularization.

Dominant packages, annotated by *dominant packages*, is a main cause of bad distribution of classes: moving classes from

Table I
INFORMATION ABOUT USED SOFTWARE APPLICATIONS.

$\mathcal{O}riginal$	$\lvert\mathcal{M}_C\rvert$	ICD	$\lvert\mathcal{M}_P\rvert$	IPD	$IPCD$	IPC	$IPCC$	$maxP_{size}$	$\frac{\lvert\mathcal{M}_C\rvert}{\lvert\mathcal{M}_P\rvert}$
JEdit	802	2683	19	1430	1032	110	26	173	42.2
ArgoUML	1671	7432	76	5661	1406	517	63	156	22
Jboss	3094	8859	455	7219	296	1898	41	80	6.8
Azureus	4212	13945	380	10929	1319	2037	136	213	11

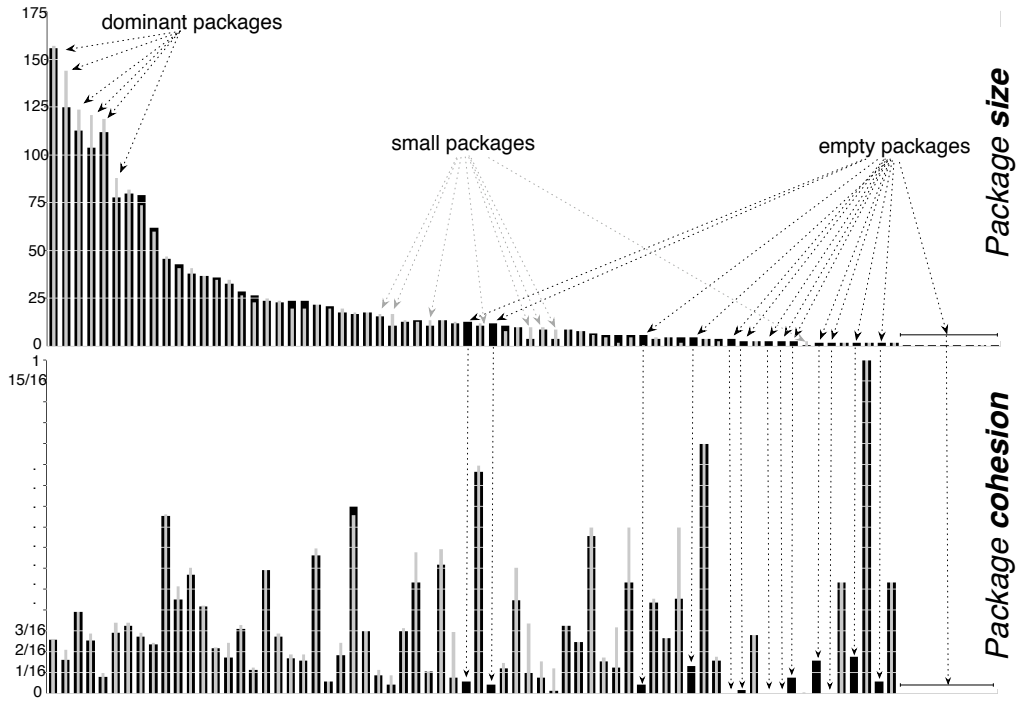

Figure 2. Package size and cohesion into *ArgoUML original* (dark gray) and $resulting$ (light gray) modularizations. Packages have the same order in diagrams.

dominant packages to small ones generally produces more dependencies and connections among packages. Maintainers can avoid moving classes from small packages to dominant ones by limiting the $size_{max}$ (Section IV-C1) of dominant packages to their original size: $p_{size_{max}} = p_{size_{V0}}$. This way, the dominant package size will never increase and the optimization process will search better modularizations by moving classes among/to smaller packages.

Fortunately, in the case of $JEdit_1$, only 10.5% (2/19) of packages are empty (Table IV), where Table III shows that our optimization process has effectively optimize package connectivity in $JEdit_1$.

Now as a future work, we have to perform a deep manual validation since in presence of late-binding and frameworks, some small packages may extend larger ones and as such may have a real reason to exist. Note that defining such packages as $forzen$ (Section IV-C3) will keep those packages existing.

While some packages became empty, Table IV shows that the average package size ($\frac{\lvert\mathcal{M}_C\rvert}{\lvert\mathcal{M}_P\rvert}$) for the resulting modularizations is really close to the average package size for the original ones Table I. Similarly, we can see that for the maximum package size ($maxP_{size}$). This shows that the optimization algorithm conserves the original system shape.

Package quality optimization: Table V shows that package quality average is also optimized: cohesion quality average ($CohesionQ_{Avg}$), coupling quality average ($CouplingQ_{Avg}$) and cyclic-dependency quality average ($CyclicsDQ_{Avg}$) for resulting packages are also almost all optimized, even if $distance_{max}$ is limited to only 5%. This can be seen also in Fig. 2.

In only one case ($JEdit_1$ and $JEdit_2$), the package coupling quality ($CouplingQ$) decreased with a very good improvement of $CyclicsDQ$ and $CohesionQ$. We explain this by the fact that the optimization process gives more importance to inter-packages cyclic-dependencies. Indeed, $CyclicDQ$ had a very bad value in the original modularization $JEdit$ (Table II): the ratio of inter-package cyclic-dependencies ($\frac{IPCD}{ICD}$) shows that 38.4% of inter-class dependencies form cyclic-dependencies between packages. Moreover there are 802 classes distributed over only 19 packages, so that the search space for generating new modularization is limited.

Table II
PACKAGE QUALITY IN ORIGINAL MODULARIZATIONS

$\mathcal{O}riginal$	$CohesionQ_{Avg}$	$CouplingQ_{Avg}$	$CyclicsDQ_{Avg}$
JEdit	28.8%	91.4%	40.6%
ArgoUML	17.2%	76.6%	81.6%
Jboss	12.5%	61.8%	96.4%
Azureus	11.7%	72.3%	84.7%

Table III
OPTIMIZATIONS ON INTER-PACKAGE CONNECTIVITY. THE TOP TABLE SHOWS THE PERCENT OF REDUCTION OF IPD, .. , IPCC (TABLE I) INTO RESULTING MODULARIZATIONS. THE BIGGEST NEGATIVE VALUE IS, THE BEST OPTIMIZATION IS. THE BOTTOM TABLE SHOWS THESE INFORMATION WHEN $distance_{max}$ IS SPECIFIED AND LIMITED TO 5%.

$\mathcal{O}ptimization_1$	IPD	$IPCD$	IPC	$IPCC$
$JEdit_1(d = 8.9\%)$	−10.2%	−23.3%	−24.0%	−37.2%
$ArgoUML_1(d = 8.3\%)$	−04.4%	−09.0%	−32.7%	−31.8%
$Jboss_1(d = 11.9\%)$	−08.3%	−37.7%	−18.5%	−51.2%
$Azureus_1(d = 9.5\%)$	−06.0%	−23.2%	−6.2%	−28.4%

$\mathcal{O}ptimization_2$	IPD	$IPCD$	IPC	$IPCC$
$JEdit_2(d = 05.0\%)$	−06.5%	−09.4%	−20.6%	−24.3%
$ArgoUML_2(d = 05.0\%)$	−02.5%	−04.2%	−25.9%	−24.9%
$Jboss_2(d = 05.0\%)$	−03.2%	−12.6%	−11.3%	−21.6%
$Azureus_2(d = 05.0\%)$	−03.2%	−09.3%	−05.7%	−16.7%

Table IV
MODIFICATIONS ON PACKAGE SIZE. THE TOP TABLE SHOWS THE PERCENT OF EMPTY PACKAGES (TABLE I), THE BIGGEST AND THE AVERAGE PACKAGE SIZE INTO RESULTING MODULARIZATIONS. THE BOTTOM TABLE SHOWS THESE INFORMATION WHEN $distance_{max}$ IS SPECIFIED AND LIMITED TO 5%.

$\mathcal{O}ptimization_1$	$EmptyP$	$maxP_{size}$	$\frac{\lvert\mathcal{M}_C\rvert}{\lvert\mathcal{M}_P\rvert}$
$JEdit_1(d = 8.9\%)$	10.5%	176	47.2
$ArgoUML_1(d = 8.35\%)$	25.4%	157	29.3
$Jboss_1(d = 11.9\%)$	22.4%	79	8.8
$Azureus_1(d = 9.48\%)$	15.8%	219	13.2

$\mathcal{O}ptimization_2$	$EmptyP$	$maxP_{size}$	$\frac{\lvert\mathcal{M}_C\rvert}{\lvert\mathcal{M}_P\rvert}$
$JEdit_2(d = 05.0\%)$	5.3%	176	44.6
$ArgoUML_2(d = 05.0\%)$	21%	155	27.9
$Jboss_2(d = 05.0\%)$	14.7%	81	7.9
$Azureus_2(d = 05.0\%)$	12.9%	215	12.7

Consistency of resulting modularizations: since our optimization approach uses random selection, different executions produce different modularizations. To evaluate the consistency of our optimization approach, we have applied it 10 times on each case study (Table I). As a result, each application has 10 modularizations $[\mathcal{M}_1..\mathcal{M}_{10}]$. Table VI shows the average distance between every pair $(\mathcal{M}_i, \mathcal{M}_j)$. For example, between resulting modularizations for $JEdit$, there are, *in average*, only 3% of classes that have not the same packages. For $Jboss$, only 5.6% of the classes have different packages in distinct resulting modularizations.

We mainly relate this very good consistency of resulting modularizations to the improvements we introduced to the neighbor function $\mathcal{N}$ Section IV-D (*i.e.*, the probability function to being selected).

In conclusion, the obtained results are very convincing. For all the case studies, the new modularizations are clearly better than the original ones. Moreover, our optimization process produces very similar results.

Table V
OPTIMIZATIONS ON PACKAGE QUALITY. THE TOP TABLE SHOWS THE AVERAGE OPTIMIZATIONS ON PACKAGE QUALITY INTO RESULTING MODULARIZATIONS. VALUES ARE BASED ON TABLE II. THE BIGGEST POSITIVE VALUE IS, THE BEST OPTIMIZATION IS. THE BOTTOM TABLE SHOWS THESE INFORMATION WHEN $distance_{max}$ IS SPECIFIED AND LIMITED TO 5%.

$\mathcal{O}ptimization_1$	$CohesionQ_{Avg}$	$CouplingQ_{Avg}$	$CyclicsDQ_{Avg}$
$JEdit_1$	+06.1%	−00.8%	+10.4%
$ArgoUML_1$	+08.1%	+07.4%	+00.4%
$Jboss_1$	+08.5%	+11.6%	+01.8%
$Azureus_1$	+05.8%	+03.6%	+05.4%

$\mathcal{O}ptimization_2$	$CohesionQ_{Avg}$	$CouplingQ_{Avg}$	$CyclicsDQ_{Avg}$
$JEdit_2$	+05.4%	−02.2%	+05.2%
$ArgoUML_2$	+06.0%	+07.4%	+00.1%
$Jboss_2$	+04.0%	+09.4%	+00.5%
$Azureus_2$	+03.9%	+04.6%	+02.2%

Table VI
RESULTING MODULARIZATION CONSISTENCY. TABLE SHOWS THE AVERAGE DISTANCE BETWEEN TEN RESULTING MODULARIZATIONS FOR EACH APPLICATION.

$\mathcal{O}ptimization_1$	$Distance_{Avg}$
$JEdit_1$	3%
$ArgoUML_1$	4.1%
$Jboss_1$	5.6%
$Azureus_1$	4.9%

VI. RELATED WORKS

Our work is mostly related to work on software modularization and decomposition [1], [14], [15], [21], [22], [25], [26], [27], [31].

Mitchell *etal.* [21], [22], [25] introduced a search-based approach based on hill-climbing clustering technique to cluster software modules (classes in our context). Their approach starts with an initial population of random modularizations. The clustering algorithm clusters each of the random modularization and selects the result with the largest quality as the suboptimal solution. Recently, they used Simulated Annealing technique to optimize resulting clusters [25], [26], [27]. Their optimization approach creates new modularizations by moving randomly some classes (a block of classes) to new clusters. The goal of their approach is increasing cluster internal dependencies.

Harman *etal.* [15] introduces a non-exhaustive hill climbing approach to optimize and determine a sequence of class refactorings. Similarly to our approach, they also restricted their approach to only move methods (classes in our context) over existing classes (packages in our context). The goal of their approach is reducing the class coupling, basing on the Coupling Between Objects (CBO) metric [3]. To avoid having a very large classes, they also used the dispersion of methods over classes (the standard deviation of methods per class metric) as a factor to measure the quality of resulting class refactoring sequences.

Abreu *etal.* [1] used hierarchical agglomerative clustering methods to decompose software classes into packages. Their clustering methods starts with a set of classes considering that each class is placed within a singleton cluster.

The goal of their approach is also increasing package internal dependencies (*i.e.*, package cohesion). In addition to the package cohesion, they used the dispersion of classes over packages (*i.e.*, package size dispersion) as a factor to measure the modularization quality.

Seng *etal.* [31] and Harman *etal.* [14] proposed genetic algorithms to partition software classes into subsystems (packages). Their algorithms start with an initial population of modularizations. These algorithms apply genetic operators on packages to modify current modularizations and/or create new modularizations into the population. The goal of both works is increasing package internal dependencies. Seng *etal.* consider also cyclic-dependencies between packages as anti-pattern for package design quality.

Our approach has several advantages compared to those works.

Considering explicitly the original modularization and controlling the optimization process: our approach tackles the problem of optimizing existing software modularizations rather than the problem of software re-modularization. Indeed, our optimization approach starts from one original modularization instead of an initial population of modularizations or a flat set of classes. Although we use an optimization technique similar to that in Mitchell *etal.* work, we restrict ours to moving classes over existing packages rather than creating new packages since we want to minimize the distance for a maintainer between the initial situation and the resulting one. Even if some prior works support the notion of importing a defined clustering [33] and restrict modifications to only moving classes over existing packages [15], we did not found, in the software re-modularization literature, approaches that explicitly take into account the original modularization structure as we do. Our approach allows maintainers to specify a set of constraints (*e.g.*, the number of classes that may change their package, the package maximal size and the packages/classes which should not be changed/moved) and therefore make them control the optimization process. Although those constraints are simple, they are very important and helpful for the automatic optimization of software modularization.

Searching better modularizations by doing near-minimal modifications: differently of those cited works, which search good modularizations without considering the distance between resulting modularizations and the original ones, we introduce a probability function that improves the derivation of neighbor modularizations by taking into account the distance between resulting modularizations and the original one, in addition to other constraints (Section IV-C) and package quality parameters (Section III-B). The great advantage of the probability function is finding better modularization by doing near-minimal modifications (Section V, the distance between resulting modularizations and the original one is very small).

Reducing package coupling and cycles in two levels (inter-package dependencies and connections): another advantage of our approach is that we use an evaluation function consisting of a combination of multiple measures. This allows us to have a much richer quality model than the approaches cited above which are mostly based on the unique goal of maximizing package internal dependencies. Although those approaches aim at reducing the global number of inter-package dependencies (*e.g.*, the fitness measure MQ used by Mitchell *etal.* [25], [26], [27]), they do not take into account the number of coupled packages (*i.e.*, the number of inter-package connections). As consequence, they do not check whether package coupling is reduced or not along the optimization process. In addition, those cited prior works do not consider package cycles. Excepted Seng *etal.* [31], they consider inter-package cyclic-dependencies, without taking in account inter-package cyclic-connections.

VII. Conclusion and Future Work

In this paper, we addressed the problem of optimizing existing modularizations by reducing the connectivity, particularly the cyclic-connectivity, among packages. We proposed an optimization algorithm and a set of measures that our optimization process uses to automatically evaluate the quality of a modularization. When designing our optimization approach, we exploited several principles of package design quality to guide and to optimize the automatic derivation of new modularizations from an existing one. We limited the optimization process to only moving classes over existing packages. We also introduced constraints related to package size, to the number of classes that are allowed to change their packages and to the classes/packages that should not be moved/changed. The results obtained from 4 case studies on real large applications shown that our optimization algorithm has been able to reduce, significantly, package coupling and cycles, by moving a relatively small number of classes from their original packages. These results are important because the chosen applications have radically different original modularizations (in terms of number of classes/packages, inter-class/inter-package dependencies, *etc.*).

As future work, we intend to enhance our approach by: (1) supporting indirect cyclic-dependencies among packages, (2) taking into account visibility of classes and particular cases of classes (*e.g.*, inner classes in Java). Indeed in this paper, we considered that classes are always public and then can change their packages, (3) setting up a real validation supported by proper statistics and qualitative analyses of resulting source code structures. We also envisage a deep comparison with the closest related work (*e.g.*, Bunch).

Acknowledgment

We gratefully acknowledge the financial support of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872).

REFERENCES

[1] F. B. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: are we being over-persuaded? In *CSMR '01*, pages 47–57, 2001. IEEE Computer Society.

[2] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of OO systems. In *CSMR '04*, pages 3–14, 2004. IEEE Computer Society.

[3] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[4] W. L. Chapman, J. Rozenblit, and A. T. Bahill. System design is an np-complete problem: Correspondence. *Systems Engineering*, 4(3):222–229, 2001.

[5] J. Clarke, J. J. Dolado, M. Harman, B. Jones, M. Lumkin, B. Mitchell, K. Rees, and M. Roper. Reformulating software engineering as a search problem. In *IEEE Proceedings on Software*, volume 3, pages 161–175, 2003.

[6] F. DeRemer and H. H. Kron. Programming in the large versus programming in the small. *IEEE TSE*, 2(2):80–86, 1976.

[7] D. Doval, S. Mancoridis, and B. S.Mitchell. Automatic clustering of software systems using a genetic algorithm. In *STEP '99*, page 73, 1999. IEEE Computer Society.

[8] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE TSE*, 27(1):1–12, 2001.

[9] A. Farrugia. Vertex-partitioning into fixed additive induced-hereditary properties is np-hard. *The Electronic Journal of Combinatorics*, 11, 2004.

[10] J. A. Ferland and D. Costa. Heuristic search methods for combinatorial programming problems, 2001.

[11] M. Fowler. Reducing coupling. *IEEE Software*, 2001.

[12] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[13] M. Harman. The current state and future of search based software engineering. In *FOSE '07*, pages 342–357, 2007. IEEE Computer Society.

[14] M. Harman and R. Hierons and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO '02*, pages 1351–1358, 2002.

[15] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *GECCO'07*, pages 1106–1113, 2007.

[16] S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[17] X. Liu, S. Swift, and A. Tucker. Using evolutionary algorithms to tackle large scale grouping problems. In *GECCO '01*, 2001.

[18] C.-H. Lung, X. Xu, M. Zaman, and A. Srinivasan. Program restructuring using clustering techniques. *J. Syst. Softw.*, 79(9):1261–1279, 2006.

[19] R. Lutz. Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture*, 47(7):613–634, 2001.

[20] H. Maini, K. Mehrotra, C. Mohan, and S. Ranka. Genetic algorithms for graph partitioning and incremental graph partitioning. In *Supercomputing '94*, pages 449–457, 1994. IEEE Computer Society Press.

[21] S. Mancoridis and B. S. Mitchell. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98*. IEEE Computer Society Press, 1998.

[22] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM '99*, 1999. IEEE Computer Society Press.

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

[24] R. C. Martin. The tipping point: Stability and instability in OO design, 2005. Software Development.

[25] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO '02*, pages 1375–1382, 2002.

[26] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE TSE*, 32(3):193–208, 2006.

[27] B. S. Mitchell and S. Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12(1):77–93, 2008.

[28] B. S. Mitchell, S. Mancoridis, and M. Traverso. Using interconnection style rules to infer software architecture relations. In *GECCO '04*, 2004.

[29] M. O'Keeffe and M. O. Cinnéide. Search-based software maintenance. In *CSMR '04*, 0:249–260, 2006.

[30] M. O'Keeffe and M. O. Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[31] O. Seng, M. Bauer, M. Biehl, and G. Pache. Search-based improvement of subsystem decompositions. In *GECCO '05*, pages 1045–1051, 2005.

[32] G. Serban and l. G. Czibula. Restructuring software systems using clustering. *ISCIS'07*, pages 1–6, 2007.

[33] V. Tzerpos and R. C. Holt. The orphan adoption problem in architecture maintenance. *Reverse Engineering, Working Conference on*, 0:76, 1997.