Automated Software Remodularization Based on Move Refactoring

A Complex Systems Approach

Marcelo Serrano Zanetti, Claudio Juan Tessone, Ingo Scholtes and Frank Schweitzer

Chair of Systems Design – www.sg.ethz.ch – ETH Zurich – Switzerland
{mzanetti, tessonec, ischoltes, fschweitzer}@ethz.ch

Abstract

Modular design is a desirable characteristic of complex software systems that can significantly improve their comprehensibility, maintainability and thus quality. While many software systems are initially created in a modular way, over time modularity typically degrades as components are reused outside the context where they were created. In this paper, we propose an automated strategy to remodularize software based on *move refactoring*, i.e. moving classes between packages without changing any other aspect of the source code. Taking a complex systems perspective, our approach is based on complex networks theory applied to the dynamics of software modular structures and its relation to an n-state spin model known as the *Potts Model*. In our approach, nodes are probabilistically moved between modules with a probability that nonlinearly depends on the number and module membership of their adjacent neighbors, which are defined by the underlying network of software dependencies. To validate our method, we apply it to a dataset of 39 JAVA open source projects in order to optimize their modularity. Comparing the source code generated by the developers with the optimized code resulting from our approach, we find that modularity (i.e. quantified in terms of a standard measure from the study of complex networks) improves on average by 166 ± 77 percent. In order to facilitate the application of our method in practical studies, we provide a freely available ECLIPSE plug-in.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Modules and interfaces, Object-oriented design methods; D2.8 [*Metrics*]: Complexity measures; D.3.3 [*Language Constructs and Features*]: Modules, Packages

Keywords remodularization, refactoring, complex networks

1. Introduction

The modular design of complex software systems is an important factor that contributes to the success of software engineering projects. It is enabled by a set of design principles, among which *information hiding* and *separation of concerns* are the most influential ones [13, 30, 31, 34]. These two principles translate into commissioning different modules to different purposes, such that their internal implementation is transparent to developers making use of their functionalities. This approach has been shown to limit necessary coordination efforts and fosters the simple replacement of obsolete software modules by new ones [9, 39], thus bearing great relevance to the maintenance of sustainable software engineering regimes [37, 42].

In the modular design of software the question about the right level of granularity for a module is quite important. Ideally, to represent a reasonable *module*, a software component should be composed of a *highly cohesive* set of interdependent subcomponents which cannot be easily separated into smaller modules. At the same time, to represent a separate module, such a software component should exhibit a reasonably *low degree of coupling* to other modules. The goal of designing a modular software architecture in which modules exhibit at the same time *high cohesion* and *low coupling* is often achieved in the design phase of a project. However, empirical studies have shown that modularity often deteriorates throughout the subsequent phase of extending and maintaining a software [42–44]. Hence, in order to retain the favorable properties of a modular design, remodularization strategies are needed. They rely on a software restructuring strategy known as *refactoring* [16].

In this paper, we address the question of how automated suggestions for refactoring can be used to improve the modularity of code. In order to minimize the impact on the actual code structures, and thus simplifying the application of our approach in practical settings, we focus on the particular type of *move refactoring*: software constructs are moved between modules without changing other aspects of the source code. If these move refactorings are applied in such a way that the cohesion within modules increases, while the coupling between modules decreases, the modularity of the software improves without affecting the behavior and functionality of the software. While move refactoring is considered as a standard technique to remodularize software, approaches in the literature emphasize difficulties in its practical application that are due to cascades of subsequent move refactorings triggered by the moving of a single software construct [10, 15]. To avoid this caveat, we take a complex systems perspective and frame the remodularization of software based on move refactoring with a scheme similar to simulated annealing [22], in which the system is driven to an equilibrium state [8] by simple local changes. Based on this view, we derive a stochastic optimization algorithm which automates remodularization via move refactoring and validate it in a empirical study on the source code of 39 JAVA open source projects. We show that this approach creates software structures that have higher modularity than the original architectures extracted from the aforementioned empir-

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2772-5/14/04…$15.00.
http://dx.doi.org/10.1145/2577080.2577097

ical dataset. We further show that the achievable gain in modularity is related to the level of modularity in the initial architecture, hence indicating the presence of a significant modularization potential in architectures that exhibit low modularity. Although focused in software written in JAVA, we argue that our methodology can be easily extended to other programming languages and paradigms. To foster the reproduction of our results and catalyze their potential impact, we also provide a software prototype of our implementation as an ECLIPSE plug-in.

The rest of this paper is organized as follows: we present our methodology in section 2, discuss our results and their limitations in section 3 and section 4, relate our approach to previous works in section 5 and present our conclusion in section 6.

2. Methods

In this section we describe the steps required to understand and reproduce our results. We start with our empirical datasets, then we move to the interpretation of software constructs and their dependencies in terms of the network structures manipulated during our remodularization strategy, followed by the description of its algorithm. We take inspiration from complex networks theory and apply the Newman's Q modularity measure introduced in [28, 29] and reinterpreted in [43, 44] to score the congruence between coupling and cohesion in a given modular decomposition and finally, we introduce the prototype of an ECLIPSE plug-in implementing a framework that will be expanded to include other approaches, fostering future research on this topic.

2.1 Datasets

We consider two distinct datasets. The first is composed of a curated collection of official releases of 14 JAVA open source software (*OSS*) projects, with a minimum of at least 10 releases each. These releases include the source code as well as the compiled binaries. This dataset is known as QUALITAS CORPUS [35]. The second dataset is composed of 28 JAVA OSS projects, for which fine grained CVS repository logs are available. The logs are aggregated over periods of 30 days such that each aggregation constitutes a full release of the given project. This dataset was previously used in [17, 18, 44], and it was not updated due to the fact that for most of these projects, the development on CVS repositories became obsolete. In Table 1, we present the list of projects, the respective number of snapshots and the date corresponding to the last one.

Table 1. Our datasets of JAVA OSS projects. For the QUALITAS CORPUS dataset, the column "Snapshots" indicate the number of releases of a given project, while in the case of CVS logs it indicates the number of monthly snapshots aggregated over the recorded project history.

	QUALITAS CORPUS	
Project	Snapshots	Last Snapshot Date
ANT	21	2010-12-27
ANTLR	20	2011-07-18
ARGOUML	16	2011-12-15
AZUREUS	57	2011-12-02
ECLIPSE_SDK	40	2011-09-10
FREECOL	28	2011-09-27
FREEMIND	16	2011-02-19
HIBERNATE	100	2012-02-08
JGRAPH	39	2009-09-28
JMETER	20	2011-09-29
JUNG	23	2010-01-25
JUNIT	23	2011-09-29
LUCENE	28	2011-11-20
WEKA	55	2011-10-28

	CVS logs	
Project	Snapshots	Last Snapshot Date
ARCHITECTURWARE	46	2008-02-04
ASPECTJ	62	2008-02-01
AZUREUS	54	2008-01-01
CJOS	87	2008-02-04
COMPOSESTAR	26	2008-07-04
ECLIPSE	83	2008-03-01
ENTERPRISE	64	2008-02-04
FINDBUGS	58	2008-02-04
FUDAA	60	2008-07-01
GPE4GTK	18	2008-07-04
HIBERNATE	50	2008-02-04
JAFFA	59	2008-01-28
JENA	86	2008-02-01
JMLSPECS	71	2008-01-28
JNODE	32	2008-02-03
JPOX	41	2008-01-28
OPENQRM	13	2008-03-01
OPENUSS	44	2008-07-01
OPENXAVA	38	2008-02-04
PERSONALACCESS	39	2008-07-04
PHPECLIPSE	66	2008-07-04
RODINBSHARP	27	2008-07-04
SAPIA	62	2008-07-01
SBLIM	79	2008-07-01
SPRINGFRAMEWORK	59	2008-02-03
SQUIRRELSQL	74	2008-07-04
XMSF	48	2008-07-04
YALE	71	2008-02-01

2.2 Software Dependency Networks

In the following description, we focus on software written in JAVA. However, our approach can be applied right away to software projects developed in other programming languages and paradigms that have suitable abstractions for *modules* and *dependencies*. In particular, we assume that dependencies between JAVA packages represent the *coupling* between modules. Although JAVA was not designed with a specific abstraction for modules [20], it allows *classes* to be grouped into namespaces that are called *packages*. It is considered good practice to organize these packages following modularity principles: high intra-package cohesion and low inter-package coupling [2, 7, 21]. We adopt the same approach and consider a JAVA package as a reasonable approximation for a module. Furthermore, we assume that a package A depends on a package B when a JAVA class (i.e. network node) a in A depends on a class b in B. Here, dependency stands for any kind of relationship between classes such as inheritance, as well as references to attributes or methods. A single link between a and b is created if there is at least one such dependency[1]. By this definition a package is highly cohesive when its classes are tightly connected. Similar approaches were applied in [7, 44]. Figure 1 provides an illustration of our method.

In order to extract such dependency networks (also known as call graphs) from the OSS projects found in the QUALITAS CORPUS dataset, we use a customized version of an OSS parser called DEPENDENCYFINDER [36]. An alternative approach is used to parse the dataset composed of CVS logs. For regular intervals of 30 days, we check out all the corresponding logs and aggregate them, resulting in monthly releases. The dependency network is then extracted by employing the abstract syntax tree parser JDT. For both datasets, the output of this process is a list of links of the form a, b, A, B, meaning class a, which belongs to package A, depends on a class b found in package B.

2.3 A Complex Systems Approach to ReModularization

Our approach to remodularization is based on *move refactoring*, a technique to reorganize source code which does not modify neither the software dependency network, nor the behavior or func-

[1] in this *simplification* links have no weights, but we argue that it can be generalized to weighted links

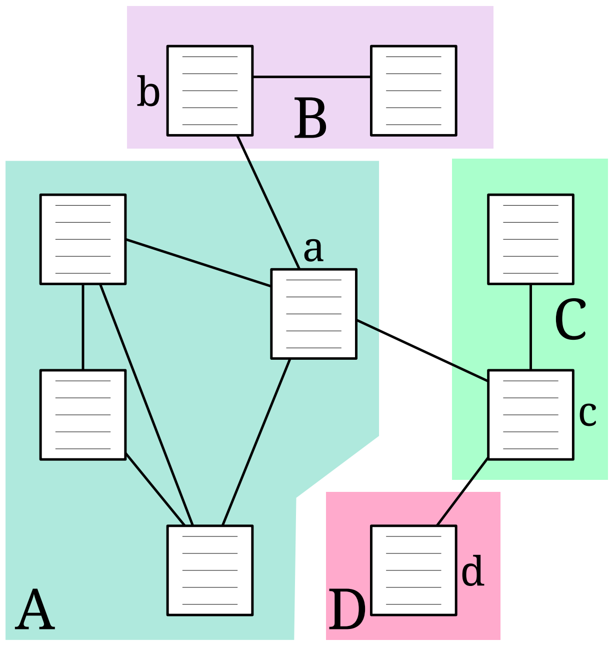
```
package A;
import B.*;
import C.*;

public class a extends b{
        public static void main (String[] args) {
                c object_c = new c();
                object_c.runMethod();
                ...
        }
        ...
}
```

```
package C;
import D.*;

public class c{
        public static void main (String[] args) {
                d object_d = new d();
                object_d.runMethod();
                ...
        }
        ...
}
```

(a) JAVA source code excerpt



(b) corresponding dependency network

Figure 1. Example of a modular software. (a) Source code excerpt. (b) Corresponding undirected network structure. The shaded areas represent modules (e.g. JAVA packages), which are internally composed of software constructs (e.g. JAVA classes). Links between such elements indicate structural dependencies (e.g. class inheritance, reference to attribute or method, etc).

tionality of the software itself. As an example, consider the modular software system (e.g. written in an object oriented programming language) which is illustrated in Figure 2(a). This system is composed of three coupled modules A, B and C. As described in section 2.2, these dependencies are the result of the interaction between the classes within each module, which can be located internally (intra-module dependencies) or across different modules (inter-module coupling). Too much inter-module coupling hinders modular architectures. For example, in terms of developer cognition, highly coupled modules cannot be easily isolated, forcing the developer to go over all the inter-module dependencies in order to understand the functionalities of a single module. In summary, the more coupling exists between modules, the harder it becomes to maintain and expand the software [12, 24, 38].

Move refactoring offers a simple solution to this problem. It consists of moving software constructs within a module to adjacent modules without changing the dependency structure of the software. In terms of the example discussed above, by carefully moving classes from their original modules into other modules, it is possible to reduce the coupling between modules. Thus, move refactoring applied to a software dependency network translates into relabeling the network nodes (e.g JAVA classes) according to module membership (e.g. JAVA package membership). In Figure 2, we illustrate the result of five move refactorings involving a single class each (the classes are $a1, a2, b1, c1, c2$). The modules in the refactored system, represented by Figure 2(b), are indeed less coupled. It is important to note that when moving content around, while ignoring the semantics of each module, it is likely that the principle of separation of concerns will be violated [13, 30, 34]. We further discuss this issue in section 4.

For small systems, such as the one illustrated in Figure 2, move refactoring is a trivial task and can be performed manually. Due to the structural complexity of software, the larger the system, the harder it is for a developer to grasp which could be suitable move refactoring steps. As described in section 5, most of the literature addresses this issue by means of optimization techniques. In most of these techniques, every possible move needs to be scored by the evaluation of a global optimization criterion (e.g. an *objective function* quantifying coupling and cohesion). In this paper, we propose a stochastic move refactoring strategy that does not require to keep track of such optimization criteria[2]. Besides providing an interesting, new, and simpler, perspective on remodularization based on complex system theory, our approach also addresses concerns in the literature regarding the explicit use of coupling-cohesion metrics when guiding the optimization search.

Our algorithm works as follows: For a modular system composed of n packages and k classes, at each time step, we pick a class c at random and count the number of links $N_j^{(c)}$ connecting it to other classes in each package j, such that $j \in \{module(c')|c' \in \mathcal{N}(c)\}$. Here, $\mathcal{N}(c)$ represents the set of classes adjacent to c (or in other words, the neighborhood of c). The probability $P_j^{(c)}$ that this class will be moved to package j is

$$P_j^{(c)} = \frac{\exp\left(N_j^{(c)}/T\right)}{\sum_{i=1}^{n} \exp\left(N_i^{(c)}/T\right)}. \tag{1}$$

Thus, this randomly picked class has higher probability to be moved into a package where it maintains most of its connections. Indeed, this could be its current package. In such a case this class has higher probability to not undergo move refactoring. The *temperature* parameter T (constant) controls the likelihood of moves that would deteriorate the modularity of this architecture.

[2] see Algorithm 1

This deterioration is characterized by the increase of the number of inter-module links if "bad" moves actually occur. The smaller T, the smaller the chance to select such move refactorings. Although small, this probability is not zero. This nonvanishing probability fosters the exploration of rugged problem landscapes, allowing the search to escape local optima.

From a computational point of view, it is worth remarking that (for projects with large number of classes) the exponential term $\exp\left(N_j^{(c)}/T\right)$ may yield an out-of-bounds error because of numerical precision. In order to avoid this, we can find $N_{max}^{(c)} = \arg\max_{l\in[1,n]} N_l^{(c)}$, i.e. the maximum number of nodes connected to c by inter-module links. Then, we compute

$$P_j^{(c)} = \frac{\exp\left(-\frac{N_{max}^{(c)}-N_j^{(c)}}{T}\right)}{\sum_{i=1}^{n}\exp\left(-\frac{N_{max}^{(c)}-N_i^{(c)}}{T}\right)}, \tag{2}$$

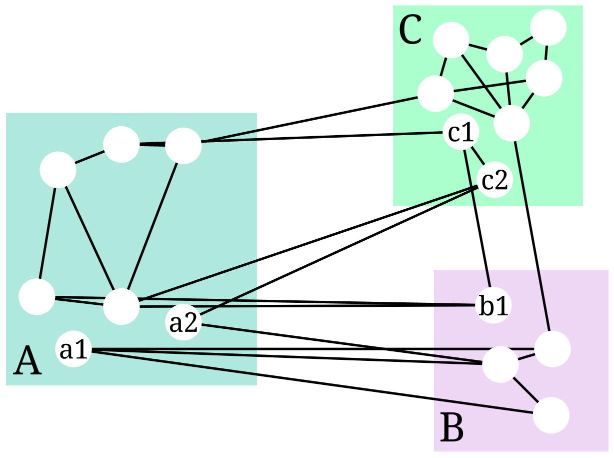
which is equivalent to Eq. 1, and each exponential term is smaller than one.

To summarize, at each step we perform a move refactoring iteration according to the probability distribution P. This procedure is repeated for a finite number of steps. Algorithm 1 presents the pseudocode of our stochastic move refactoring strategy, while in Figure 3 we illustrate one step of this algorithm.

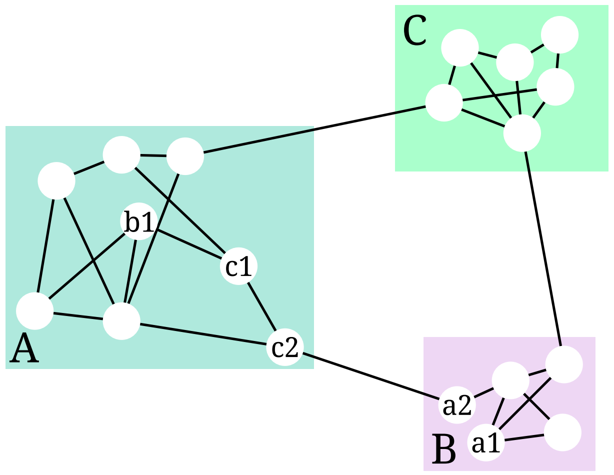
```
initializeParameters($T$, $n\_iterations$);
$network$ := loadNetworkFromSourceCode();
**for** $i \leftarrow 1$ **to** $n\_iterations$ **do**
    $node$ := pickRandomNode($network$);
    $normTerm$ := 0;
    $N_{max}$ := $node$.mostLinkedModule.numberOfLinks;
    $P$ := emptyArray();
    **for each** $j$ **in** $modulesInNeighborhoodOf(node)$ **do**
        /*Count the number of links between $node$ and
        module $j$*/
        $N_j$ := countLinksToModuleJ($node$.neighbors, $j$);
        /*The probability to move $node$ to module $j$.*/
        /*The temperature parameter $T$ controls the
        likelihood of bad moves.*/
        $p$ := $\exp -\frac{N_{max}-N_j}{T}$;
        $normTerm$ := $normTerm + p$;
        $P.append((p, j))$;
    **end**
    /*Normalize the probability distribution $P$*/
    **for** $j \leftarrow 1$ **to** $P.length()$ **do**
        $P[j].p := P[j].p/normTerm$;
    **end**
    /*Decide which module receives $node$ according to
    probability distribution $P$*/
    $node$.module := moveRefactoring($node$, $P$);
    $network$ := updateNetwork($node$, $network$);
**end**
```

Algorithm 1: Stochastic move refactoring algorithm. The temperature parameter T is a constant, therefore a cooling schedule is not required. We emphasize that a node can only be move refactored to adjacent modules in which it maintains software dependencies.

In statistical physics, the model described by Eq. 1 is similar to the n-state *Potts Model* [40]. In a fully connected graph, this system is a paradigmatic model to study the equilibrium phase transition (as a function of temperature) from an ordered state, where all the nodes reside in the same module–to a disordered one–where all the nodes are randomly located in different modules. In the case



(a) original



(b) refactored

Figure 2. Illustration of move refactoring. Shaded areas represent modules, which are composed of classes (i.e. circles) bound by undirected software dependencies. Moving classes across modules can decrease the coupling between modules. (a) original modular decomposition. (b) modular decomposition after move refactoring. The resulting modules are less coupled. We emphasize that move refactoring only modifies the module membership of a class. The dependencies (i.e. links) on other classes remain untouched.

of complex topologies–like those found in class dependencies–the equilibrium configuration will depend on the modular coherence inside of the software: the more interdependent particular groups of classes are, the more likely they will be assigned–in equilibrium–to the same module.

There are several properties of this system which made it the objective of a large body of literature in the realm of physics. Here, we will simply mention a few properties that are sufficient to understand the relevance of using this model within the context of this paper.

For the n-state *Potts Model*, it is possible to write for each node an individual *objective function*, which dictates the score of the current configuration of package assignment. Let π_c denote the package a class c is assigned to. Then, the objective function for

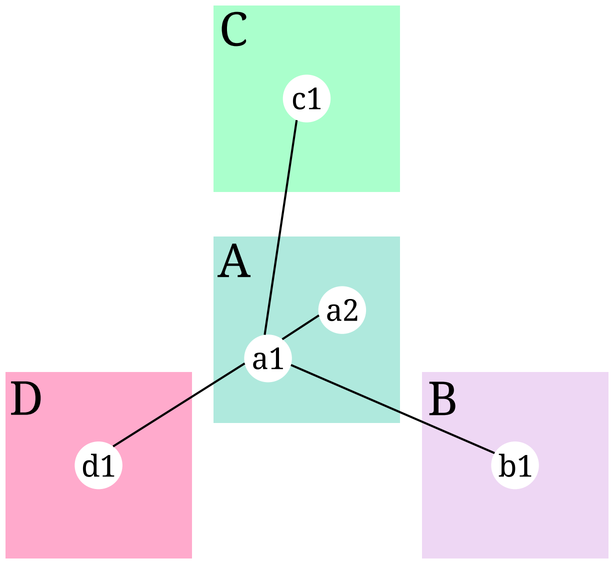


Figure 3. Class $a1$ will be refactored. It can remain in package A, or be moved to package B, C or D. Due to the topology of this simple example (i.e. a single link to each package), each possibility has equal probability to take place. We emphasize that–using Algorithm 1–class $a1$ can only be moved to modules where it maintains software dependencies. Further generalizations are possible and will be investigated in future research.

class c reads

$$u_c = -\sum_{c' \in \mathcal{N}(c)} \delta(\pi_c, \pi_{c'}).$$

The Kronecker delta function δ is equal to one if both arguments are equal (i.e. if classes c and c' belong to the same package), zero, otherwise. The sum runs over all classes c' which have dependency relations with class c, i.e. the neighborhood of c, represented by $\mathcal{N}(c)$. Summing up over all the nodes, we obtain

$$U = \sum_{c=1}^{k} u_c = -\sum_{c=1}^{k} \sum_{c' \in \mathcal{N}(c)} \delta(\pi_c, \pi_{c'}), \tag{3}$$

which measures the total score of the current configuration. Interestingly, when class c is moved from package π_c to another π'_c, it is very simple to show that the total change is $\Delta U(\pi_c \to \pi'_c) = 2\Delta u_c$. This implies that the *local* maximization procedure, is equivalent to the *global* maximization. For this particular problem, this is a very important property, as it implies that this simple local rule is equivalent to a global one. This also implies that U in Eq. 3 is the *total energy* of the system.

During the simulations, at every time step there are many possible configurations of module assignment for every node in the source code of the project. Over time, the algorithm *samples* the space of all possible assignments, such that the sampling probability of a given configuration is a function of Equation 3. The process of sampling is thus equivalent to the *Metropolis* algorithm [26], which also allows the convergence time to be determined in a standard way [11, 19, 27]. Because of the results shown in section 3, it is apparent that the energy landscape is not rugged, but smooth. Thereby, the modularization process proposed in this paper always converges to a stationary state, and a simulated annealing approach (meaning the cooling schedule for the temperature) is not needed.

2.4 An Alternative Metric for Coupling and Cohesion

We follow the progress of our automated move refactoring strategy by applying the Newman's Q measure, a quantitative approach widely used in complex networks theory [28, 29]. This was reinterpreted in [42–44] as an alternative method to monitor the evolution of software modularity. In those empirical studies, we focus on JAVA open source projects and show that Q successfully expresses the congruence of the clusters of software dependencies between classes and the decomposition of source code in terms of JAVA packages. It is defined as

$$Q = \frac{\sum_i^n e_{ii} - \sum_i^n a_i b_i}{1 - \sum_i^n a_i b_i} \tag{4}$$

where e_{ij} is the fraction of links that connect nodes in module i to nodes in module j, $a_i = \sum_j^n e_{ij}$ and $b_i = \sum_j^n e_{ji}$ are the column and row sum respectively, while n corresponds to the number of modules. If the network is undirected, the matrix defined by $\mathbf{e}$ is symmetric and $a_i = b_i$ [28]. We use Q to measure the fraction of links that connect nodes within the same module ($\sum_i^n e_{ii}$) minus the value of the same quantity expected from a randomized network ($\sum_i^n a_i b_i$). If the former is not better than random $Q = 0$ [29]. However, Q would not be defined if all links are concentrated within a single module. For such trivial case, the scaling factor equals zero ($1 - \sum_i^n a_i b_i = 1 - 1 = 0$). To avoid such a division by zero, we define $Q = 0$. In general, $Q \in [-1, 1]$. That is, the less coupled the modules and the higher their cohesion, the closer Q is to 1. As an illustration of its application, $Q = 0.37$ for the network in Figure 2(a), while $Q = 0.84$ for the one in Figure 2(b).

2.5 SOMOMOTO: An Eclipse Plugin for ReModularization

SOMOMOTO is an ECLIPSE plug-in and its name stands for "software modularization and monitoring tool". Its initial goal is providing a framework for remodularization of software written in JAVA. It is a tool that developers can use to monitor the evolution of a modular software architecture, both quantitatively and visually. For the quantitative part, we implement Q as described in section 2.4, and we are planning to include other approaches available in the literature. For the visualization of modular software architectures, we make use of GEPHI's library for graph and network layout [5]. Besides monitoring software modularity, we are also able to act against its deterioration. This is achieved by implementing our automated strategy discussed in section 2.3. Furthermore, we plan to include competing approaches to foster direct comparison with our methodology. We also plan to allow developers to interfere with the algorithm's behavior, for example, by enabling manual move refactoring aided by an interactive network visualization interface. Moreover, we plan to allow the developers to define binding constraints to forbid or prioritize specific move refactoring options, to which any automated approach must comply. The source code, freely distribute with a GPL V3 license, is available at `http://sourceforge.net/projects/somomoto/`.

3. Results

In the following, we apply our strategy to the JAVA OSS datasets described in section 2.1. For each project listed in Table 1, we follow the procedure outlined in section 2.2 to extract the software dependency network of its last snapshot. This network is used as the input of our strategy (see Algorithm 1) and we run it for 20 different values for the temperature parameter T. We choose $T \in [0.01, 1000]$ such that these values are uniformly distributed on a logarithmic scale. We repeat this process 20 times in order to average the dynamics with respect to T.

3.1 The Temperature and the Equilibrium Configuration

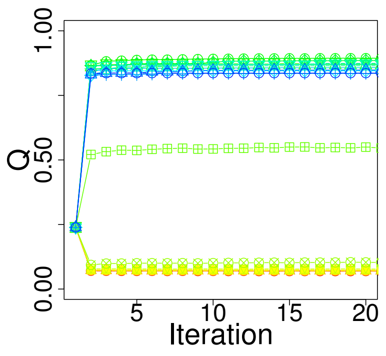
In Figure 4 and 5, we depict the Q value and the number of modules with respect to the iterations executed by our strategy. We show three projects belonging to the QUALITAS CORPUS dataset: the IDE ECLIPSE_SDK, the graphical library JUNG and the database interface HIBERNATE, because the results obtained for these three projects are representative for the projects listed in Table 1. In accordance with the theoretical discussion presented in section 2.3, low temperature values (i.e. $T < 0.1$) lead to equilibrium configurations with low inter-module coupling and high intra-module cohesion. This range of temperature makes deteriorating move refactoring steps very unlikely. Thus software modularity improves substantially, as expressed in terms of the high Q values seen in figures 4(a), 4(b) and 4(c).

Interestingly, the highest Q values and the lowest number of modules are obtained within an intermediate temperature range (i.e. $0.1 < T < 10$). For this range, we show in Table 2 that a small improvement in Q (i.e. $\approx 4.0\%$)–with respect to the range $T < 0.1$–is associated with a comparably larger drop in the number of modules (i.e. $\approx 17\%$). Furthermore, as depicted in Figures 5(a), 5(b) and 5(c), the execution of our strategy always leads to a significant drop in the number of modules. For the lowest temperature (i.e. $T = 0.01$) this drop is lowest and corresponds to losing $68.4 \pm 13.2\%$ of the original modules. For higher temperatures, the drop is even larger. Thus, as a side effect of our strategy, a substantial fraction of the original structure of the source code is lost. Although associated with an improvement in modularity, it is not understood how this drop in the number of modules can affect development performance. More research is needed to study if for example, this extra improvement of $\approx 4\%$ in Q values (e.g. from 166% to 170%) justify a further drop of 17% in the number of modules (e.g. from 68% to 85%). As a rule of thumb–if remodularization is expected to preserve the most possible of the original modular structure–only values of $T \ll 0.1$ should be considered.

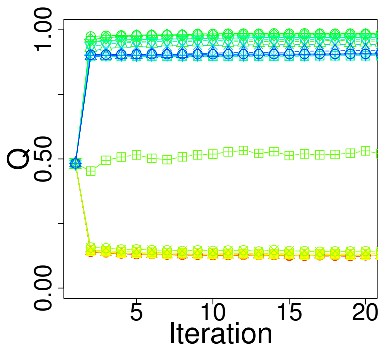
Table 2. Average values for the change in Q and in the number of modules for different temperature ranges. For the lowest temperature (i.e. $T = 0.01$) our strategy improves Q in $166.6 \pm 77.3\%$, while decreasing the number of modules in $68.4 \pm 13.2\%$.

Temperature Range	ΔQ (%)	Δ Modules (%)
$T = 0.01$ (lowest)	166.6 ± 77.3	-68.4 ± 13.2
$T < 0.1$	166.5 ± 77.6	-68.4 ± 13.2
$0.1 < T < 10$	170.5 ± 105.2	-85.4 ± 9.7
$10 < T$	-50.1 ± 18.6	-82.9 ± 9.7
$T = 1000$ (highest)	-52.1 ± 16.7	-82.4 ± 9.9

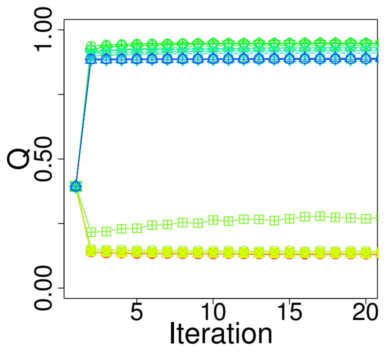
Figure 6(a) depicts the relation between Q and the number of modules on the temperature parameter T. In this figure, we only consider the equilibrium values of the former two quantities. We bin the data points with respect to T and calculate the median value. We also show the 90% and 10% quantiles. The first insight is that the variability in Q is almost constant with respect to T, decreasing slightly during the abrupt change between high and low Q values. For small T, the variability in the number of modules is comparably higher, but decreases significantly as T increases. Another insight is the abrupt change in Q about $T = 10$. For $T < 10$ we observe values of Q which are significantly higher than for $T > 10$. This is further illustrated in Figure 6(b), which depicts the potential energy difference between these two states: high potential energy (i.e. high modularity and high Q) and low potential energy (i.e. low modularity and low Q). As a final remark, these two contrasting potential energy levels are the reason why we only observe few equilibrium states in figures 4(a), 4(b) and 4(c): high Q (i.e. high



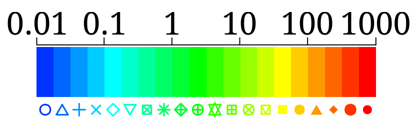
(a) ECLIPSE_SDK



(b) JUNG

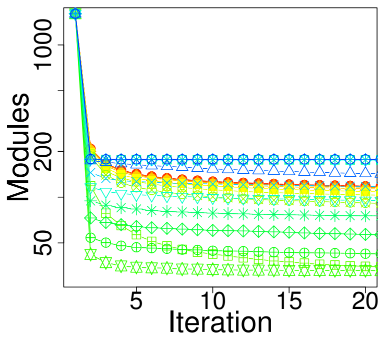


(c) HIBERNATE

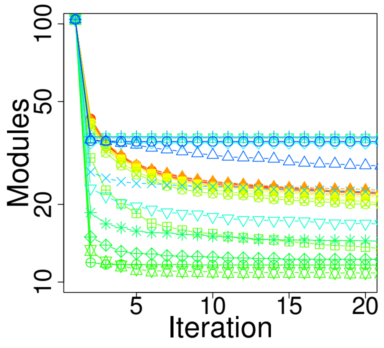


(d) T

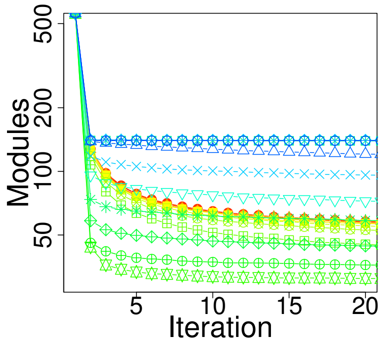
Figure 4. Evolution of Q during move refactoring steps. The iteration number k displayed in the horizontal axis of each figure corresponds to $100 \times m \times k$ move refactoring steps (i.e. m being the number of JAVA classes). Each curve represents the average of 20 runs of our strategy with different values of the temperature parameter T.



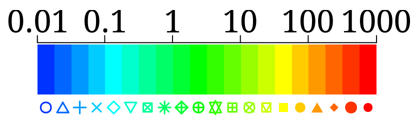
(a) ECLIPSE_SDK



(b) JUNG



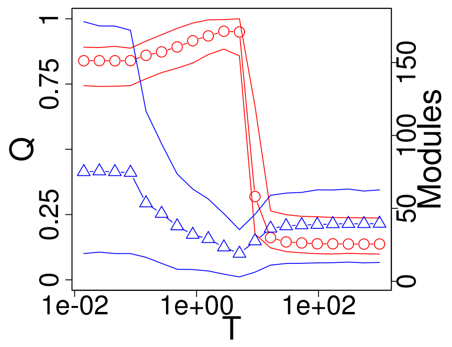
(c) HIBERNATE



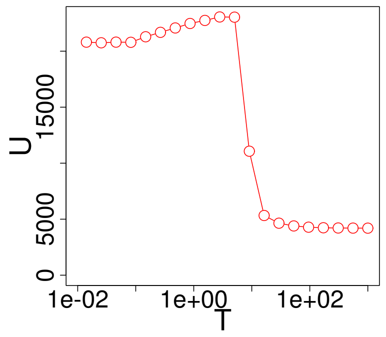
(d) T

Figure 5. Evolution of the number of required modules (i.e. non-empty modules) during move refactoring steps. For intermediary values (i.e. $0.1 < T < 10$) we obtain the highest Q values on the expense of losing a significant fraction of the original modules. Thus, the use of $T < 0.1$ is recommended (see Figure 4).

potential energy), intermediary Q (i.e. transitional state) and low Q (i.e. low potential energy).



(a) modularity and T

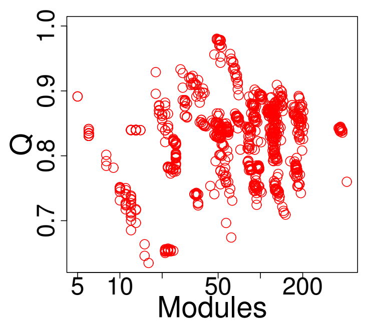


(b) potential energy and T

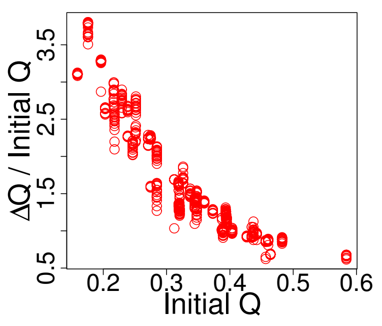
Figure 6. The role of the temperature T as a control parameter. (a) Dependency of Q (i.e. dashed red circles) and the number of required modules (i.e. dashed blue triangles) with the temperature T. Each curve is obtained by measuring the median value of the corresponding measures, when considering the simulation results aggregated over T. The solid curves above and below the corresponding measure represent the 90.0% and 10.0% quantiles respectively. There is an abrupt change in the value of Q as a function of the control parameter T. (b) Median value of the corresponding potential energy U. Structured or well modularized software falls into the T range mapping to a higher potential energy level (i.e. $T < 10$), while poorly structured software falls into the deep valley with low potential energy level (i.e. $T > 10$).

3.2 Remodularization Performance of our Strategy

In Figure 7(a), we show that the performance of our strategy does not depend on the number of modules (i.e. no correlation between Q and the number of modules). Furthermore, our strategy improved the modularity of all projects considered in this paper, resulting in remodularized software with an average value of $Q = 0.8 \pm 0.1$ for $T = 0.01$. Finally, the worse the modularity of a given architecture, the higher the relative improvement as a result of the application of our strategy. We depict this in Figure 7(b). Further research will investigate if these results hold for different datasets.



(a) Q does not depend on Modules



(b) improvement relative to initial Q

Figure 7. The performance of our strategy at equilibrium with $T = 0.01$. (a) In the studied dataset, the number of modules does not correlate with Q, thus we can discard any dependency of this kind. (b) The worse the initial value of Q (i.e. the worse the initial modular design), the larger the improvement achieved.

3.3 Move Refactoring in Empirical Data

In this section, we verify if the move refactoring suggestions discovered by our strategy were actually executed in empirical data. We focus on the CVS logs dataset, which reflects the iterative development process with greater regularity, following closely the coding decisions undertaken by the software developers.

In order to perform this comparison, we first need to be able to detect move refactoring taking place within our datasets. We solve this problem in the following way. We define a time stamped CVS log snapshot s_t, which corresponds to the set of class dependencies and respective package (module) membership observed at time t. Each class in s_t is named with respect to the pattern $package\_name_t.class\_name_t$. To detect move refactoring, we take the simple approach of looking for unique class names ($class\_name_t$) in s_t, verifying if these names are found in s_{t+1}. If the answer is positive, we check for modifications in the respective package names ($package\_name_t$). Thus, move refactoring is detected when $class\_name_t = class\_name_{t+1}$ and $package\_name_t \neq package\_name_{t+1}$. We emphasize that this approach only detects move refactoring of the kind defined in this paper: a refactoring step that only modifies the package membership of a class, without touching upon any of its contents and the network of software dependencies.

With the move refactoring detection method outlined above, we are able to compare our strategy output with the work of the software developers. For each two consecutive CVS log snapshot s_t and s_{t+1}, we extract the respective empirical software dependency networks net_t^e and net_{t+1}^e (see section 2.2). Let D be the set of move refactoring steps performed by the developers between net_t^e and net_{t+1}^e. Furthermore, we use net_t^e as the input of our algorithm and let it run until convergence (for $T = 0.01$). The network of software dependencies resulting from this procedure is defined as net_{t+1}^s. Finally, let S be the set of move refactoring steps performed by our strategy and detected between net_t^e and net_{t+1}^s. We compare these two sets, thresholding on the ΔQ between t and $t+1$, so that we focus on move refactoring taking place during significant improvements in software modularity. For different values of ΔQ, we calculate $precision$ and $recall$ and present the results in Table 3. The results show that our strategy correctly suggest most of the move refactoring steps performed by the software developers, as indicated by the relatively high values listed in the column $recall$. In fact, our algorithm is much more *aggressive* than the developers when suggesting move refactoring steps. This is further discussed this in section 4. Thus, our resulting set of suggestions is much larger than the set chosen by developers. This is the reason why our $precision$ values are relatively small: the software developers do not use move refactoring consistently as mean to restore software modularity.

Table 3. Comparison between the set of move refactoring steps suggested by our strategy S, against the set of steps performed by the developers D upon the empirical data. Quantitatively: $precision = \frac{|S \cap D|}{|S|}$ and $recall = \frac{|S \cap D|}{|D|}$. We present these measures for different values of the threshold parameter ΔQ (i.e. change in modularity measured in empirical data), thus allowing us to focus on the move refactoring steps that had significant impact on software modularity.

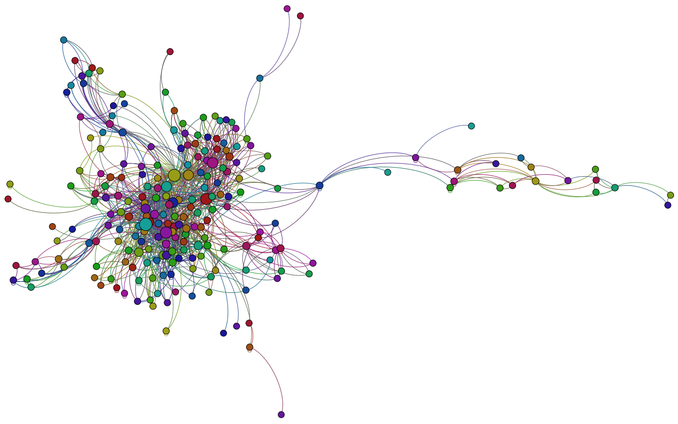
Δ Q (%)	$precision$ (%)	$recall$ (%)
1	4.9 ± 15.7	59.9 ± 35.4
5	7.0 ± 16.9	62.4 ± 35.3
10	8.1 ± 19.2	62.7 ± 39.0
15	5.7 ± 8.9	52.4 ± 40.8

3.4 SOMOMOTO in Action

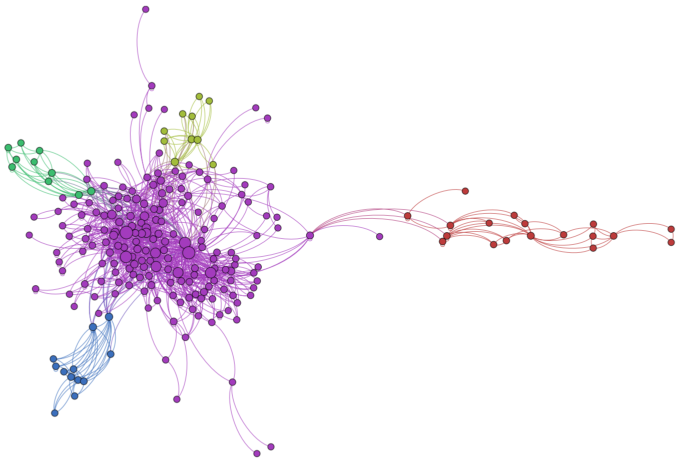
As a simple test case, we employ SOMOMOTO in the remodularization of a JAVA graphical library called JGRAPHX. Figure 8 depicts the software dependency network and the module membership of classes of JGRAPX, before and after remodularization. The resulting network, depicted in Figure 8(b), clearly shows the congruence between the clusters of software dependencies and the source code decomposition into JAVA packages. Network nodes (i.e. classes) bearing the same color are members of the same modules (i.e. packages).

4. Threats to Validity

Here, we address some of the concerns related to the results of our approach. The first issue is our conscious decision of not considering the semantics of modules during the remodularization via automated move refactorings. We are well aware of the fact that there are modules whose contents should not be move refactored, in despite of their significant impact on inter-module coupling. For example, modules responsible for user interfaces may fall within this category. Related to this issue, there might be modules that are



(a) original JGRAPHX ($Q = 0.04$)



(b) refactored JGRAPHX ($Q = 0.85$)

Figure 8. Test case: the remodularization of JGRAPHX (a JAVA graphical library). The JAVA classes are depicted as circles, while their color reflects the corresponding package membership (same color, same package). (a) original. (b) after remodularization by SOMOMOTO.

believed to be already well structured. In such cases, further refactoring them would be detrimental. The simplest solution, which we are planning to include in SOMOMOTO, is to allow developers to mark modules and also classes that should not be remodularized by an automated refactoring strategy. Further ideas related to direct interference in the behavior of the algorithm, allowing it to cope with developer preferences are possible. For example, the contents of obsolete modules might need to be move refactored into other modules. For such cases, our strategy can be applied by focusing on a few modules, redistributing their content.

Another issue that might be circumvented by allowing the direct interference of software developers is the observed significant drop in the number of modules, even for small values of the temperature parameter T. Our results show that at least $\approx 68\%$ become empty. One possible explanation is found in [7], where the authors study a similar dataset of JAVA OSS projects, showing that the minimization of the inter-module coupling and maximization of intra-module cohesion is not a dominating module design principle. Thus, a more realistic perspective on automated remodularization should include complementary quantitative dimensions. These additions, together with the implementation of competing approaches, will be included in our ECLIPSE plug-in, in order to foster direct comparison with our methodology, and also to provide a unified framework for the remodularization of JAVA software. These steps will foster its use in practice. We are further interested in the opinion of software developers on the outcome of our automated move refactoring strategy, also to understand if the seldom use of move refactoring observed in our datasets is a general issue. We expect that move refactoring, based on our automated strategy, will be more frequently applied in practice. As shown in this paper, the underlying problem landscape seems to be smooth, at least with respect to the temperature parameter T. Thus, a convergence to favorable software modularities can be ensured.

5. Related Work

Software evolves in ways that do not necessarily reflect positively in its modularity. In order to cope with the deterioration of the latter, refactoring strategies can be employed. It has been argued in [10, 15], that approaches considering developer expertise–to directly refactor the source code–seldom allow for a significant improvement in software modularity. The difficulties are mainly related to the problem of detecting possible candidates for refactoring. This opens up many opportunities for the development of automated refactoring methodologies. Among the available approaches, the ones that imply a reformulation of software modularity as a combinatorial problem are quite common. Furthermore, most of those are mainly concerned with the minimization of inter-module coupling and maximization of intra-module cohesion [3], as dictated by software engineering wisdom [13, 30, 34], both have potentially high impact on maintenance costs. One of the earliest approaches in this direction offers an optimization search guided by a genetic algorithm [14]. Their search starts with an initial modular decomposition, which at each iteration is replaced by the best decomposition found in a population controlled by the algorithm. A simple variation of this approach is to allow multiple searches to take place in parallel, such that a majority rule is used to determine the best modular decomposition [25]. An alternative way to escape local optima is discussed in [1]. Similar to our own approach, they apply simulated annealing allowing the acceptance of moves that do not always improve the functional being maximized. Moves that improve the respective functional are always accepted. Our approach is different for being completely governed by Eq. 1, such that every move bears a probability of being executed. Their absolute contribution to the energy function influences this probability but do not force an immediate acceptance. The authors also introduce constraints to limit some aspects of the optimization search that are missing here: maximal number of classes that can change their packages, maximal number of classes that a package can contain and the classes that should not change their packages. These are in line with the idea of having software developers interfering with automated approaches more effectively, as discussed in [23, 32]. We plan to include this methodology in future releases of our plug-in. Furthermore, [1] report results on modularity improvement only for highly limited values for these three constraints. These result in small improvement in modularity, which cannot be compared to the results–significantly higher–that we present in our work. Complementary to the discussion above, the work presented in [7] classifies modules by their role within the architecture. They show that modules controlling *io* and *gui* functions are the most congruent regarding cohesion and coupling metrics. Moreover, [6, 33] advocate the use of metrics based on the semantics of modules besides structural dependencies. According to [17, 18], structural dependencies are not uniformly important with respect to the propagation of changes. Thus they emphasize that future research should focus on their semantics rather than the structure. Other approaches in the literature seek to group software constructs into modules according to measures that express their similarity, a technique better known as *clustering*. Examples of works within this context are presented in [4, 23, 32]. In [41], a comparison between different clustering strategies concludes that clustering algorithms do not reproduce the existing modular decomposition of software projects, calling for further research.

6. Conclusion

In conclusion, we have introduced a simple stochastic algorithm that allows to remodularize software architectures based on an automated suggestion of *move refactorings*. This algorithm is based on the assumption that an optimum modular design of software minimizes the *coupling* between modules, while the *cohesion* within modules is maximized. We take a complex networks perspective on modularity in software dependency networks and capture both cohesion and coupling by a network-based, quantitative measure. Furthermore, making use of the n-state *Potts Model* known from statistical physics, our stochastic algorithm provides a complex systems approach to the optimization of software modularity in dependency networks. We validate the remodularization performance of our algorithm by applying it to two datasets which allows us to study the evolution of software dependency networks for 39 JAVA open source software projects. The results of our analysis validate that the modularity of these projects can be increased on average by $166 \pm 77\%$. We further show that the achievable gain in modularity is related to the level of modularity in the initial architecture, hence indicating the presence of a significant modularization potential in architectures that exhibit low modularity. Based on empirical data on the evolution of software modularity in JAVA projects, we further extract *move refactorings* performed by developers to remodularize the software architecture. We then compare the suggestions of our algorithm with the actual actions of developers and compare *precision* and *recall* of the refactoring suggestions. The fact that our approach achieves a comparably high recall while the precision is low highlights that a) our method suggests most of the move refactorings that were identified by developers and b) that our method was able to identify many more move refactoring than were actually implemented by real developers. We argue that this finding opens a number of interesting further research directions: First, it can be seen as a challenge for the assumption that optimal modular designs (from the perspective of developers) coincide with a maximization of cohesion and a minimization of coupling. Reasons for this most likely include the importance of context in the choice of the package decomposition of projects, as well as the existence of dependencies to third-party packages whose modular structure cannot be easily changed. Secondly, it can be interpreted in such a way that our method highlights a significant modularization potential that currently goes unused in actual software projects. Finally, it highlights the necessity of introducing an additional parameter to our algorithm, that influences how *aggressive* it is. In summary, we argue that our work is a promising example for the applicability of models, methods and abstractions from the study of complex systems and complex networks in software engineering.

7. Acknowledgment

We acknowledge ETH ZÜRICH for financial support and also Vahan Hovannisyan for implementing the prototype of our ECLIPSE plug-in.

References

[1] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui. Automatic package coupling and cycle minimization. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 103–112. IEEE, 2009.

[2] D. Ancona and E. Zucca. True modules for java-like languages. In *ECOOP 2001Object-Oriented Programming*, pages 354–380. Springer, 2001.

[3] N. Anquetil and J. Laval. Legacy software restructuring: Analyzing a concrete case. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 279–286. IEEE, 2011.

[4] G. Antoniol, M. Di Penta, and M. Neteler. Moving to smaller libraries via clustering and genetic algorithms. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 307–316. IEEE, 2003.

[5] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Proceedings of the ICWSM '09*. AAAI, 2009.

[6] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Software remodularization based on structural and semantic metrics. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 195–204. IEEE, 2010.

[7] F. Beck and S. Diehl. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 354–364. ACM, 2011.

[8] E. Bertin. *A concise introduction to the statistical physics of complex systems*. Springer, 2012.

[9] K. Blincoe, G. Valetto, and S. Goggins. Proximity: a measure to quantify the need for developers' coordination. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1351–1360. ACM, 2012.

[10] S. Bryton and F. B. e. Abreu. Modularity-oriented refactoring. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 294–297. IEEE, 2008.

[11] J. Cruz and C. Dorea. Simple conditions for the convergence of simulated annealing type algorithms. *Journal of applied probability*, pages 885–892, 1998.

[12] J. S. Davis. Effect of modularity on maintainability of rule-based systems. *International Journal of Man-Machine Studies*, 32(4):439–447, 1990.

[13] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[14] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Software Technology and Engineering Practice, 1999. STEP'99. Proceedings*, pages 73–81. IEEE, 1999.

[15] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring-improving coupling and cohesion of existing code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 144–151. IEEE, 2004.

[16] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[17] M. M. Geipel. Modularity, dependence and change. *Advances in Complex Systems*, 15(06), 2012.

[18] M. M. Geipel and F. Schweitzer. The link between dependency and co-change: Empirical evidence. *IEEE Transactions on Software Engineering*, 38(6):1432–1444, 2012.

[19] V. Granville, M. Krivánek, and J.-P. Rasson. Simulated annealing: A proof of convergence. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(6):652–656, 1994.

[20] R. Hall, K. Pauls, S. McCulloch, and D. Savage. *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[21] E. Hautus. Improving java software through package structure analysis. In *The 6th IASTED International Conference Software Engineering and Applications*, 2002.

[22] S. Kirkpatrick, D. G. Jr., and M. P. Vecchi. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[23] R. Koschke. Atomic architectural component recovery for program understanding and evolution. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 478–481. IEEE, 2002.

[24] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[25] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 315–324. IEEE, 2003.

[26] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21:1087, 1953.

[27] D. Mitra, F. Romeo, and A. Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. In *Decision and Control, 1985 24th IEEE Conference on*, volume 24, pages 761–767. IEEE, 1985.

[28] M. E. J. Newman. Mixing patterns in networks. *Phy. Review E*, 67: 026126, 2003.

[29] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.

[30] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[31] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *Software Engineering, IEEE Transactions on*, 11 (3):259–266, 1985.

[32] S. Parsa and O. Bushehrian. Genetic clustering with constraints. *Journal of research and practice in information technology*, 39(1):47–60, 2007.

[33] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 469–478. IEEE, 2006.

[34] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[35] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 APSEC*, pages 336–345, 2010.

[36] J. Tessier. The dependency finder user manual. . Dependency Finder (2001-2012). Revised BSD License., 2012.

[37] C. J. Tessone, M. M. Geipel, and F. Schweitzer. Sustainable growth in complex networks. *EPL (Europhysics Letters)*, 96:58005, 2011.

[38] Y. Umeda, S. Fukushige, K. Tonoike, and S. Kondoh. Product modularity for life cycle design. *CIRP Annals-Manufacturing Technology*, 57(1):13–16, 2008.

[39] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams. Using software repositories to investigate socio-technical congruence in development projects. In *MSR '07*, pages 25–25. IEEE, 2007.

[40] F. Y. Wu. The potts model. *Reviews of Modern Physics*, 54:235, 1982.

[41] J. Wu, A. E. Hassan, and R. C. Holt. Comparison of clustering algorithms in the context of software evolution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 525–535. IEEE, 2005.

[42] M. S. Zanetti. The co-evolution of socio-technical structures in sustainable software development: Lessons from the open source software communities. In *Proceedings of the 34th ICSE*, pages 1587–1590. IEEE Press, 2012.

[43] M. S. Zanetti. *A complex systems approach to software engineering*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 21653, 2013, 2013.

[44] M. S. Zanetti and F. Schweitzer. A network perspective on software modularity. In *Architecture of Computing Systems (ARCS) Workshops 2012*, pages 175–186. GI, IEEE, 2012.