Muchou Wang, Weifeng Pan*, Bo Jiang and Chenxiang Yuan

CLEAR: Class Level Software Refactoring Using Evolutionary Algorithms

Abstract: The original design of a software system is rarely prepared for every new requirement. Software systems should be updated frequently, which is usually accompanied by the decline in software modularity and quality. Although many approaches have been proposed to improve the quality of software, a majority of them are guided by metrics defined on the local properties of software. In this article, we propose to use a global metric borrowed from the network science to detect the moving method refactoring. First, our approach uses a bipartite network to represent classes, features (i.e., methods and fields), and their dependencies. Second, a new metric is introduced to quantify the modularity of a software system as a whole. Finally, a crossover-only evolutionary algorithm that uses the metric as its fitness function is introduced to optimize the class structure of a software system and detect the methods that should be moved. Empirical results on the benchmark Java projects show that our approach can find meaningful methods that should be moved with a high stability. The advantages of our approach are illustrated in comparison with some other approaches, specifically one refactoring approach, namely search-based refactoring approach (SBRA), and two community detection algorithms, namely a graph theoretic clustering algorithm (MCODE) and a fast algorithm for community detection (FG). Our approach provides a new way to do refactoring from the perspective of software structure.

Keywords: Refactoring, class, software network, evolutionary algorithm, modularity.

DOI 10.1515/jisys-2013-0058
Received July 24, 2013; previously published online August 13, 2014.

1 Introduction

An object-oriented (OO) software system is usually composed of a set of classes that is in turn an encapsulation of related state (fields) and behavior (methods). To improve the quality of software systems, a fundamental approach is to modularize the design. In software engineering practice, modularization can be viewed as a reflection of the famous design principle *low coupling and high cohesion*. Thus, it is a good practice to keep the fields and methods within a class as compact as possible while the interactions between fields and methods that are from different classes as few as possible [5]. However, the original design of a software system is rarely prepared for every new requirement. Software systems should be updated frequently. Owing to the tight schedule, updates are usually performed by different people, which will break the software modularization and low software quality [13]. In general, the coupling between classes increases while the cohesion of a class declines.

To improve the quality of software, Fowler provided a concrete solution, termed as refactoring, which is defined as the process to improve the internal structure of the code yet does not alter its external

***Corresponding author: Weifeng Pan,** School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou, Zhejiang 310018, P.R. China, e-mail: panweifeng1982@gmail.com
Muchou Wang: Wenzhou University Library, Wenzhou University, Wenzhou, Zhejiang, P.R. China
Bo Jiang and Chenxiang Yuan: School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou, Zhejiang, P.R. China

behavior [8]. Until now, many refactoring strategies have been proposed. Among them, moving method is one of the effective refactoring strategies performed at the class level. It improves software quality by moving the method that is using or used by more features of another classes than the class where it is defined [8]. However, to perform moving method refactoring, we should identify the methods that should be moved and the target classes that they should be moved to. It is a challenging and time-consuming task.

The objective of this article is to use an evolutionary algorithm (EA) to help programmers identify the methods that should be moved to maximize cohesion of classes and minimize coupling between classes. Our approach, Class Level software refactoring using Evolutionary AlgoRithms (CLEAR), may require moving method between classes virtually. First, CLEAR adopts a bipartite software network to describe the topological structure of a piece of software, where nodes denote features (i.e., fields and methods) and classes, and the dependency between every pair of nodes, if any, is an edge. Second, a new metric is introduced from network science to quantify the software modularity. A crossover-only evolutionary algorithm (COEA) will be devised to optimize the class structure with regard to the value of modularity. Finally, CLEAR identifies the methods and the corresponding target classes that these methods should be moved to by comparing the original class structure with the optimized class structure. Empirical results on the benchmark Java projects show that CLEAR can provide meaningful results with a high stability.

The primary contributions of this article are as follows:

1. The introduction of the bipartite network to represent the structure of a software system.
2. The usage of a metric from network science to quantify the software modularity.
3. The perspective of identifying the refactoring candidates using an optimization technique.

The rest of the article is structured as follows. Section 2 provides an overview of the related work. Section 3 describes our approach in detail, with focus on the definition of bipartite software networks and the introduction of the EA. Section 4 presents empirical evaluations to investigate the effectiveness of the proposed approach. Section 5 concludes the article and gives future directions.

2 Related Work

Until now, several semiautomatic approaches have been proposed to detect those methods that should be moved. Some representative work is given as follows:

Tahvildari and Kontogiannis [18] investigate the usage of OO metrics as indicators to detect potential design flaws and suggest potentially useful transformations for correcting them. Trifu and Marinescu [19] propose to use detection strategies composed of various metrics and their threshold values to detect instances of a structural anomaly. O'Keeffe and O'Cinneide [11] formulate the software refactoring as a search problem guided by a quality evaluation function in the space of alternative designs. Seng et al. [16] design a methodology to improve the class structure of a system with respect to the values of several metrics and the number of violations of OO design principles. Tsantalis and Chatzigeorgiou [20] propose a distance metric between features and classes, and based on which present a very simple method for the identification of methods that should be moved. Alkhalid et al. [1] propose a similarity metric to quantify the similarity between methods. Based on the similarity metric, they use clustering techniques to group the methods and detect the methods that should be moved.

As we can see, the existing approaches they proposed generally use OO metrics to guide refactoring activities. Indeed, the OO metrics they used mainly focus on the local properties of software such as a method, a class, and a package. But what about using the global properties to guide refactoring activities? In our primary work [13], we have proposed to represent software at package level by unipartite network and use the community detection technique in the complex network theory to refactor software packages by optimizing a metric modularity. In this article, we continue this research at the class level.

3 The CLEAR Approach

Although this is not the first work on software refactoring at the class level, we will cover a different angle, i.e., from the perspective of software as a whole using EAs. Figure 1 gives a short overview of the workflow of the proposed approach. In the following sections, we will detail it.

3.1 Java Projects

In this article, we take the open-source Java projects as our research subjects. The rationale is threefold [13]:

1. There are many open-source Java projects with sufficient supplement materials on the web that can be easily accessed for our research objectives.
2. Java projects have a relatively clear internal structure, and the entities such as features, classes/interfaces, and packages are amenable to extraction and analysis.
3. The choice of Java programming language is limited by the tools developed to perform analysis and our interest in understanding software written in Java.

3.2 Software Entity Collection

We represent a piece of software as a bipartite network [7]. We should first determine the entities to be extracted. For software refactoring at the class level, features, classes, and the dependencies between them are chosen

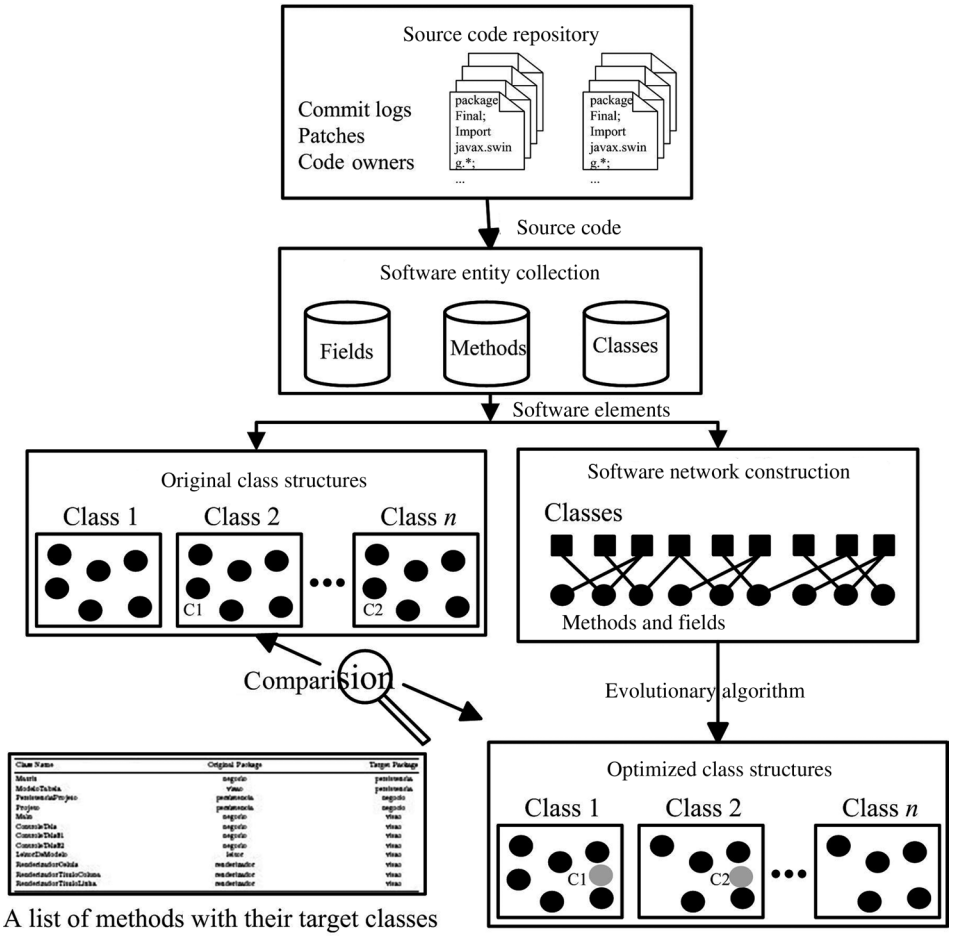

Figure 1. The Workflow of the Proposed CLEAR Approach.

as entities. Here we do not differentiate methods and fields, using the term feature to designate them. We also do not differentiate classes and interfaces, treating them as classes. Only two kinds of dependencies are taken into consideration, i.e., field reference dependency and method call dependency. The dependency between every pair of class and feature, if it exists, is obtained from the two kinds of dependencies.

3.3 Software Network Definition

Network analysis has revealed as a powerful approach to understand complex systems [21]. Software systems can also be represented as complex networks, termed as software network, where software entities are nodes and the relationships between them are edges. Network analysis has just recently been adopted to acquire better comprehension of complex software networks. It provides a simple yet effective tool to help us understand the structure and the forming mechanism of large software systems [12].

As we all know, Java software systems are composed of entities at different levels of granularity, varied from the feature level to the package level. The upper-level entities are built by the lower-level ones. Entities across different levels and their composition relationships form a topological structure that can be properly described by a new type of software network, the bipartite software network (or bipartite graph) [14]. Because the interest of the current work mainly focuses on the classes and features, the formal definition of the bipartite software network is given as follows.

Definition 1: A class is a composition of features. We use the Class-Feature bipartite SOftware Network, CFSON = (N_c, N_f, D), to explicate the dependencies between classes and features. CFSON consists of two sets of nodes, N_c (classes) and N_f (features). Only edges between nodes of unlike set are allowed, that is, $D = \{(c_i, f_j)\}$, where $c_i \in N_c$ and $f_j \in N_f$. The adjacency matrix ψ_{ij} for the bipartite network encodes the dependencies between every pair of class and feature:

$$\psi_{ij} = \begin{cases} 1 (c_i, f_j) \in D \\ 0 \quad \text{otherwise} \end{cases}, \tag{1}$$

that is, a $|N_c| \times |N_f|$ binary matrix, where $|N_c|$ is the number of classes and $|N_f|$ is the number of features. Further, we also assign a weight w_{ij} to each edge to denote the dependency strength between every pair of class i and feature j if they are connected. Indeed, a small value of weight indicates the low-dependency strength between the corresponding class and feature. It is desirable to keep the weight as small as possible for a specific software system as far as software maintainability is concerned [13]. Figure 2 shows a simple example of CFSON, where the number on each edge denotes the dependency strength.

Introduction of weights brings a flexibility that allows us to consider the dependency strength between classes and features, but it also raises a new problem: determining the weights. In this article, we will use the dependencies between the features in each class to quantify the weight on the corresponding edge between each class and the features it enclosed.

In the following, we will detail the way to calculate the weight matrix W clearly. But before that, we introduced an undirected feature dependency network (uFDN) defined in [13].

Definition 2: In uFDN, nodes denote the features of a specific Java projects. Edges between two nodes indicate the use dependency between the corresponding features, i.e., if feature A uses feature B, there is an edge

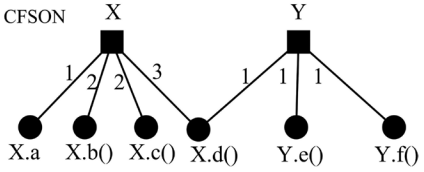

Figure 2. Illustration of CFSON.

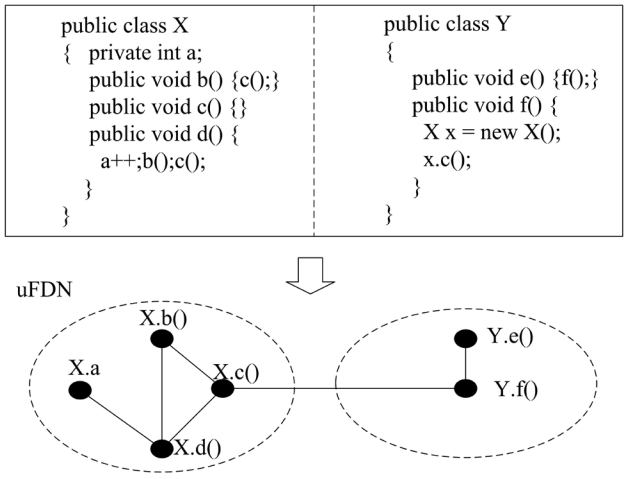

Figure 3. Illustration of uFDN.

between the nodes denoting the two features. Here we only consider the presence of dependency and neglect the multiplicity of dependencies, such as, A depends three times on B and its direction. Therefore, uFDN can be described as

$$\text{uFDN} = (N_f, E_f), \tag{2}$$

where N_f is the set of all nodes in uFDN and E_f is the set of edges. Figure 3 shows a simple code segment and its corresponding uFDN.

Once the uFDN is obtained, we can calculate the weight on each edge in CFSON. Denote $getFeature(c_i)$ as the set of all the features class c_i contains and $R_k(f_j)$ as the set of all reachable nodes originated from node f_j within a distance k. Then, the weight on the edge between class c_i and feature f_j if $(c_i, f_j) \in D$, namely $w(c_i, f_j)$, is defined as

$$w(c_i, f_j) = |getFeature(c_i) \cap R_1(f_j)|, \tag{3}$$

where |s| is used to count the elements in set s. We take the calculation of $w(X, X.b())$ as an example to show how to calculate the weight. Because $getFeature(X) = \{X.a, X.b(), X.c(), X.d()\}$ and $R_1(X.b()) = \{X.c(), X.d()\}$, $w(X, X.b()) = |getFeature(X) \cap R_1(X.b())| = |\{X.c(), X.d()\}| = 2$. Figure 2 is the corresponding CFSON of the uFDN shown in Figure 3.

3.4 Crossover-Only Evolutionary Algorithm

EAs are of the most effective techniques to solve many real-world optimization problems [17]. They are not restricted to a specific problem and can be applied to a variety of domains [9]. Because refactoring is the process to improve the quality of software gradually, software refactoring is also an optimization problem that can also be solved by EAs.

In this article, we transform the class level software refactoring problem as an optimization problem and put it under the framework of EAs. We use a simple COEA to obtain the optimized class structures and further to find the methods that need to be moved from one class to another. In the following subsections, we will detail the design of COEA from individual encoding, population initialization, fitness function, genetic operators, and algorithm flow.

3.4.1 Individual Encoding

COEA uses integer-encoding individuals (or chromosomes) to encode the potential solutions to a problem. Each gene denotes the class identifier that the corresponding software entity belongs to. The number of genes (i.e., the dimension) in the individual equals the sum of the number of classes $|N_c|$ and the number features $|N_f|$ in a specific Java project. The first $|N_c|$ genes with value starting from 1 to $|N_c|$ sequentially encode the class identifiers that each class belongs to. Each class only uses one integer as its identifier. Then the following genes signify the class identifiers that each field belongs to (the number of genes equals to the number of fields *#Fields* in a system). The last *#Methods* (the number of methods) genes signify the target class identifiers that each method should be moved to. Therefore, an individual in CLEAR encodes the potential class structure of a software system.

Thus, we can use {1,2,1,1,1,1,2,2} to encode the original real class structure of the code segment shown in Figure 3. The first gene denotes class X; the second gene, class Y; the third gene, field X.a; the fourth gene, method X.b(); the fifth gene, method X.c(); the sixth gene, method X.d(); the seventh gene, method Y.e(); and the eighth gene, method Y.f(). Because there are only two classes (X and Y), we use the first two genes (with identifiers being 1 and 2, respectively) to denote their class identifiers. Because there is only one field X.a defined in class X, we use the class identifier of class X (i.e., 1) to encode it. Because X.b(), X.c(), and X.d() are defined in class X, we use the class identifier of class X (i.e., 1) to encode them. Similarly, Y.e() and Y.f() are defined in class Y, so we use the class identifier of class Y (i.e., 2) to encode them. It is the original real class structure. If we have an individual {1,2,1,1,1,**2**,2,2} with the sixth gene changing from the original 1 to 2, it means in such an individual encoding, the feature X.d() should be moved from its original class X with identifier 1 to the target class Y with identifier 2.

3.4.2 Population Initialization

COEA is a population-based EA. Therefore, how to initialize the population can greatly affect the performance of the algorithm. When there is a lack of prior information, people usually use random initialization to generate all the individuals in the initial population. However, as for class level software refactoring, the situation is greatly different.

As we all know, classes are composed of a set of features. As a result of many design principles (e.g., low coupling and high cohesion), a majority of methods are defined in the right class, whereas only a small number of misplaced methods lower the quality of software systems and need to be moved. Thus, there is no need to start the COEA in a random way with every class and feature belonging to a random class. In light of this, COEA adopts a scheme that assigns each class and the fields defined in it with the same class identifier that is sequentially numbered and assigns a random class identifier chosen from the identifiers of classes to each method.

Thus, as for the CFSON shown in Figure 2, we can initialize the individual like {1,2,1,1,1,1,2,2}, {1,2,1,2,1,2,2,1}, or {1,2,1,1,2,1,1,2} (the entity that each gene denote is same as that we mentioned in Section 3.4.1). Because there are only two classes and one field, the identifiers for the classes and field are fixed to {1,2,1}. The identifiers for methods are randomly chosen from {1,2}. Similarly, we can generate a specific number of individuals to comprise the population.

3.4.3 Fitness Function

Fitness function is a function that used to assign fitness value to each individual. Generally, the greater is the fitness value, the better the solution it contains. Designing a suitable fitness function is of great importance for it controls the method moving process.

In complex software network research, people find that software networks have a distinct community structure, i.e., the existence of distinct groups of nodes such that there are dense connections internally and

sparser connections between groups [13]. As for software network at the feature level, these nodes denote features of a specific Java project. Therefore, generally, classes are the natural communities of features and packages are the natural communities of classes. Such a community structure of Java projects reflects the modularization and the low coupling and high cohesion principle.

There are many different metrics that can be used to quantify the modularity of a particular community division [2]. Because we use a bipartite network to represent a piece of software, the modularity of bipartite networks should be defined. Here we use the Q proposed in [2] as the modularity metric. However, the original Q is proposed for unweighted bipartite networks. Because CFSON is a weighted bipartite network, we use the weighted version of Q, which is given by

$$Q = \frac{1}{m}\sum_{i}\sum_{j}(w_{ij} - q_{ij})c(i,j), \tag{4}$$

where Q is the bipartite modularity of a particular individual (a class structure), w_{ij} is the weight on the edge between class node i and feature node j, $m = \sum_{i}\sum_{j} w_{ij}$, and $q_{ij} = m^{-1}\sum_{j} w_{ij}\sum_{i} w_{ij}$, and $c(i, j)$ is 1 if class node i and feature node j have the same class identifier and is 0 otherwise.

3.4.4 Genetic Operators

There are several kinds of genetic operators that can be used in EAs, such as selection operator, crossover operator, and mutation operator. COEA only uses selection operator and crossover operator.

1. Selection operator: Selection means to extract a subset of individuals from an existing population according to the fitness of an individual. It is usually the first operator applied on population to select parent individuals for crossover and mutation and further to produce child individuals. In COEA, we use roulette wheel selection [9] to select the potentially useful individuals for crossover.
2. Crossover operator: COEA uses one-point crossover operator [9], which selects one crossover point and then interchanges the parent individuals after the point to produce two child individuals. Figure 4 shows the crossover operation taking place at the sixth gene of two parent individuals {1,2,1,2,1,2,2,1} and {1,2,1,1,2,1,1,2}. The two individuals have been mentioned in Section 3.4.2. However, it should be noted that the crossover point is not randomly selected from any gene in the individual. Because the current work is talking about the detection of methods that should be moved, our crossover operator will only be applied on genes that denote methods. Those genes denoting classes and fields will be ignored. Thus, the crossover points will be chosen from ($|N_c|$ + #Fields) to ($|N_c| + |N_f|$). As for the two individuals in Figure 4, the crossover point is chosen from 4 to 8.

Indeed, crossover operation corresponds to the method moving operations between classes, e.g., in Figure 4, the parent individual {1,2,1,2,1,2,2,1} produces one child individual {1,2,1,2,1,2,1,2}, which is equivalent to two method moving operations, i.e., move the seventh method from the class with identifier 2 to the class with identifier 1 and move the eighth method from the class with identifier 1 to the class with identifier 2.

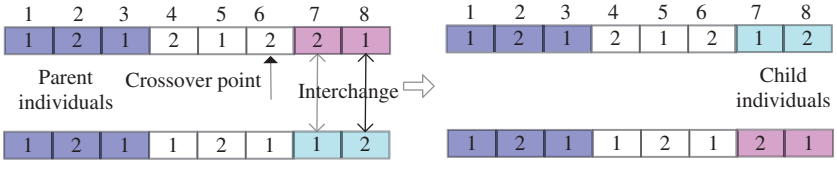

Figure 4. Illustration of One-Point Crossover.
See online version for color.

3.4.5 COEA Flow

The framework of COEA is shown in Algorithm 1, where P is the current population, $P_i = \{p_{i1}, p_{i2}, ..., p_{iD}\}$ is the ith individual in P, D is the dimension size, N_p is the population size, $p_c \in [0,1]$ is the predefined crossover probability, *rand(a,b)* is a random number within $(a,b]$, *fabs*(x) returns the absolute value of x, *FEs* is the number of fitness evaluations, and *mFEs* is the maximum number of evaluations. The ith individual P_i stores the class identifiers for all nodes the individual signifies, e.g., node j belongs to the class with identifier p_{ij} in P_i.

Algorithm 1 The COEA flow

Input:
CFSON, $|N_c|$, *#Methods*, *#Fields*
Output:
A list of methods with their target classes where they should be moved to
1: Initialize the parameters of COEA, including p_c, N_p, and *mFEs*
2: Generate the initial population with N_p individuals $P=\{P_1, P_2, ..., P_{Np}\}$
3: Calculate all the fitness values of the individuals in P according to Formula (4) and find the best individual P_{best} and the worst individual P_{worst} with their fitness being Q_{best} and Q_{worst}, respectively
4: $FEs = N_p$
5: **While** $FEs \leq mFEs$ && $fabs(Q_{best} - Q_{worst}) \geq 10^{-20}$ **do**
6: Select two different individuals P_a and P_b using roulette wheel selection from P
7: **if** $rand(0,1) \leq p_c$, **then**
8: Generate one random numbers $k = rand(|N_c| + \#Fields, |N_c| + |N_f|)$
9: Perform one-point crossover operation at point k to produce two child individuals, P'_a and P'_b
10: Calculate the Q of P'_a and P'_b, respectively
11: $FEs = FEs + 2$
12: **end if**
13: $P = P \cup \{P'_a, P'_b\}$
14: Rank the individuals in P in a descending order by their fitness values
15: Select N_p best individuals from P to form the new generation P
16: Update P_{best} and P_{worst} and their fitness values Q_{best} and Q_{worst}
17: **end while**
18: **return** A list of methods with their target classes where they should be moved to

As we can see from Algorithm 1, the most dominant steps of COEA are steps 6 to 16 within the loop. Because the computational complexity of Equation (4) is $O(|N_c|^2|N_f|^2)$, the computational complexity of COEA is $O(|N_c|^3|N_f|^3)$.

4 Experiments and Data Analysis

To investigate the effectiveness of the proposed CLEAR approach, we designed and conducted the controlled experiments. Our experiments were carried out on a PC at 2.3 GHz with 2 GB of RAM. The following subsections will describe in detail the subjects, process, and results of these experiments.

4.1 Subject

We tested the CLEAR approach on two Java projects LAN-simulation and JHotDraw. LAN-simulation is a benchmark refactoring example tjat has been widely used to evaluate the performance of the refactoring approaches [6]. We generate the version before applying the moving method refactoring. JHotDraw is a Java GUI framework for technical and structured graphics developed by Gamma and Eggenschwiler as a design exercise for using design patterns [16]. Table 1 reports the size of the two subject projects in terms of *LOC*

Table 1. Basic Data of the Subject Projects.

Subject projects	LOC	$\|N_c\|$	#Fields	#Methods
LAN-simulation	342	3	13	25
JHotDraw	8419	172	443	1277

(lines of codes), $|N_c|$, *#Fields*, and *#Methods*. We should point out that $|N_c|$ includes the number of inner classes and interfaces, and *LOC* is the practical lines of code, excluding the comment lines and blank lines.

4.2 Case Study and Results

In this section, we will show the process of our experiments on the subject projects and the results obtained. For Algorithm 1, the parameters p_c and N_p are fixed to 0.5 and 60, respectively, and the maximum number of fitness evaluation *mFEs* is 50,000. But how do we determine if the practical classes, methods, and fields that should be processed by our approach is still a problem?

Methods do not play the same role in a specific software system. As for methods that belong to design patterns, they always have special functions. Even such methods deliberately violate the design guideline like high cohesiveness and low coupling, they also cannot be moved and will be treated as special methods. JHotDraw is a design exercise for using design patterns. There might be many special methods. We use the approach proposed in [16] to detect those special methods. Thus, methods that belong to the type of pattern methods, getter and setter methods, state methods, factory methods, and delegation methods will be treated as special methods and will not be processed by our approach. In LAN-simulation, we do not find any special method, whereas in JHotDraw, we finally detect 1,050 special methods. Thus, for LAN-simulation, $|N_c| = 3$, *#Methods* = 25, and *#Fields* = 13, and for JHotDraw, $|N_c| = 172$, *#Methods* = 227, and *#Fields* = 443. In CLEAR, the special methods will not be encoded in individuals but will take part in the Q calculation.

We have developed a software analysis tool SNAT. It can parse the bytecode (files with.class and.jar extension), extract features, and their dependencies and further build CFSON. Figures 5 and 6 show the uFND and CFSON for the two subject projects. In uFDN, the notation on each node is the name of the feature the node denotes. In CFSON, the nodes denote the features or the classes. The notations on the nodes are their names. The positions of the nodes in uFDN are automatically calculated using a circular layout algorithm

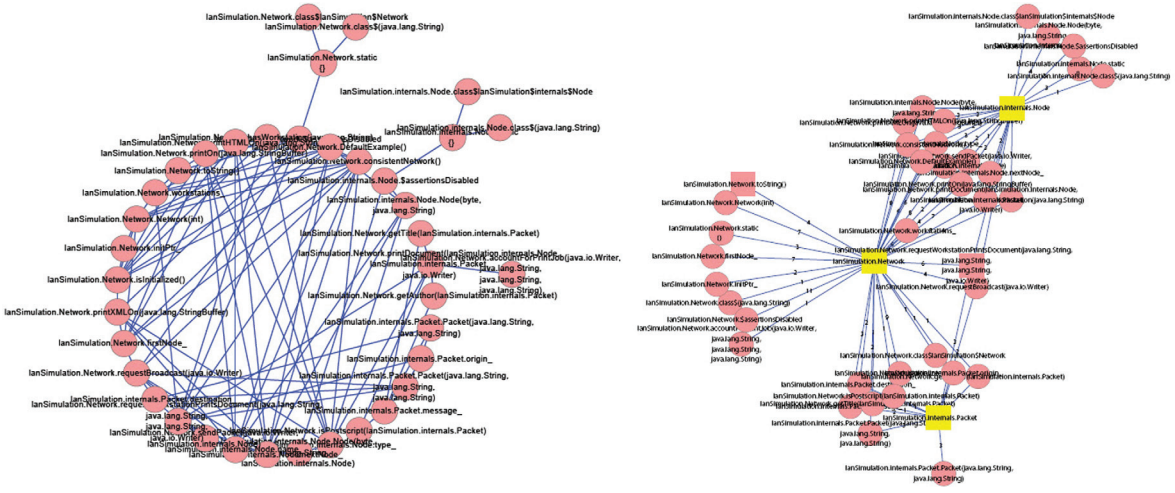

Figure 5. uFDN (Left) and CFSON (Right) for LAN-Simulation.
In CFSON, the red nodes denote the features and the yellow nodes denote the classes. See online version for colors.

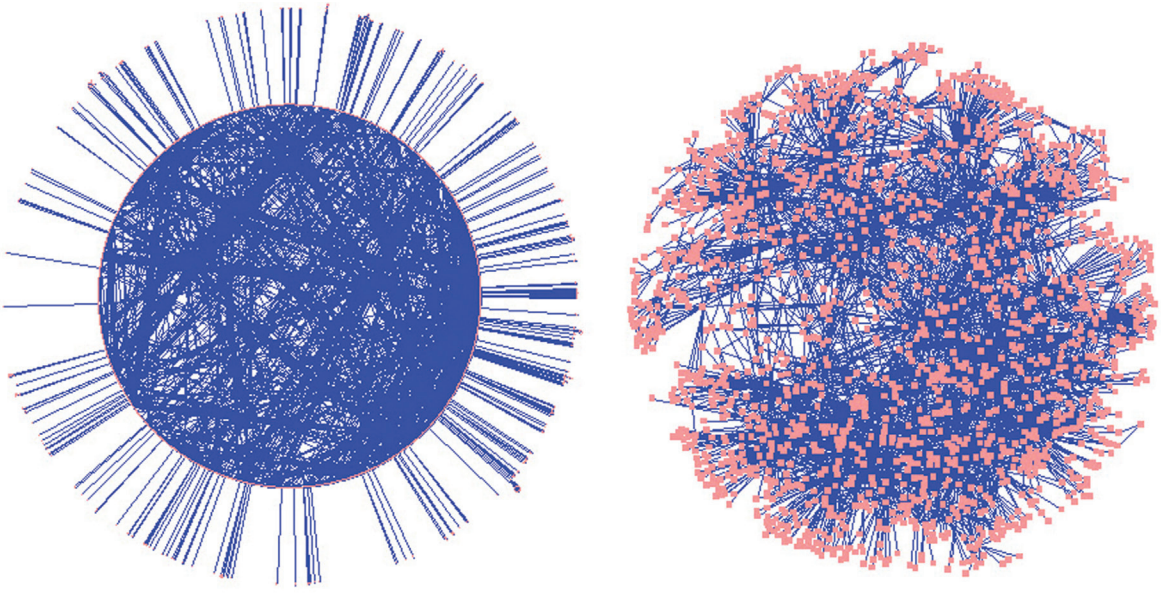

Figure 6. uFDN (Left) and CFSON (Right) for JhotDraw.
See online version for colors.

and that of CFSON are calculated using a spring-embedded algorithm [4]. Enlarging the networks can give you more information. Because the software network of JHotDraw is so large, the notations of nodes and the dependencies between nodes cannot be clearly shown. For the clear version of the figures, please refer to http://wfpan.3vfree.us/figs.rar.

We apply COEA to the CFSON of the two subject projects. The algorithm returns the methods that should be moved. These methods are shown in Tables 2 and 3, where the first column is the names of the methods, the second column is the original classes where they are defined, and the third column shows the suggested target classes that they should be moved to.

4.3 Analysis

COEA offers a list of methods with the target classes that they should be moved to. As a first goal, we wish to know whether these methods make sense to the developers. We manually checked all the detected methods one by one by referring to the source code. We find that all detected methods can be justified.

Table 2. Methods Should Be Moved for LAN-Simulation.

Method name	Original class	Target class
lanSimulation.Network.getAuthor	Network	Packet
lanSimulation.Network.getTitle	Network	Packet
lanSimulation.Network.isPostscript	Network	Packet

Table 3. Methods Should Be Moved for JHotDraw.

Method name	Original class	Target class
PertFigure.writeTasks	PertFigure	StorableOutput
PolygonFigure.chop	PolygonFigure	Geom
PertFigure.readTasks	PertFigure	StorableInput
TextTool.fieldBounds	TextTool	standard.TextHolder
TextTool.beginEdit	TextTool	standard.TextHolder

In LAN-simulation, methods getAuthor, getTitle, and isPostscript are suggested to be moved from the original class Network to the target class Packet, for all these methods are heavily used by (or use) methods in class Packet than that in class Network. As we can see from the source code, getAuthor heavily uses the field message_ with Packet type, calling its methods startsWith, indexOf, length, and substring many times, but getAuthor is only used by printDocument in class Network only one time. Method getTitle also heavily uses the field message_ with Packet type, calling its methods startsWith, indexOf, length, and substring, but getTitle is only used by printDocument in class Network only one time. isPostscript uses the field message_ with Packet type, calling its methods startsWith, but it is only called by printDocument once. Therefore, moving these three methods from their original class Network to the target Packet can reduce the coupling and improve the modularity. These methods are the right methods that should be moved, as suggested by Demeyer et al. [6].

Further, all detected methods in JHotDraw can also be justified. Method PertFigure.writeTasks is suggested to be moved from class PertFigure to StorableOutput. By referring to the source code, we find that PertFigure.writeTasks writes many Storable elements but does not directly use any fields or methods defined in class PertFigure. StorableOutput is a class to manage the storage of storable objects. Thus, from our perspective, PertFigure.writeTasks should be moved from PertFigure to StorableOutput. Method PolygonFigure.chop is suggested to be moved from class PolygonFigure to Geom. We can observe from the source code that chop makes heavy use of the methods length2 and intersect defined in Geom but does not use any field or method in the class PolygonFigure where it is defined. Therefore, we think the best place for chop would be class Geom. Methods PertFigure.readTasks, TextTool.fieldBounds, and TextTool.beginEdit can be similarly justified.

As a second goal, we want to check that if we modified the subject projects by randomly selecting 5 methods and misplacing them, whether CLEAR has the ability to move them back to the classes where they are defined. We apply CLEAR to the modified version of LAN-simulation and JHotDraw 20 independent runs. Tables 4 and 5 show the results. The first column shows the methods that are selected to be misplaced. In the 20 runs, the five methods are selected only once and kept the same in the remaining 19 runs. The second column shows the class that the corresponding method is suggested to be moved to. The last column is the number of runs that the corresponding method is rightly moved back to the class where it is defined.

As we can see from Tables 4 and 5, four of the five methods in LAN-simulation have been successfully moved back to their original classes in all runs, and all methods in JHotDraw have been successfully moved back, but the method getAuthor in LAN-simulation has never been moved back because it is a method (as we

Table 4. Manually Misplaced Methods for LAN-Simulation.

Methods	Misplaced to class	#
lanSimulation.internals.Node.Node(byte, java.lang.String)	Network	20
lanSimulation.Network.hasWorkstation(java.lang.String)	Node	20
requestWorkstationPrintsDocument(java.lang.String, java.lang.String, java.lang.String, java.io.Writer)	Node	20
lanSimulation.Network.consistentNetwork()	Node	20
lanSimulation.Network.getAuthor(lanSimulation.internals.Packet)	Packet	0

Table 5. Manually Misplaced Methods for JHotDraw.

Methods	Misplaced to class	#
PertFigure.layout	Quadtree	20
CompositeFigure.bringToFront	Quadtree	20
PolygonFigure.findSegment	FigureAttributes	20
Geom.ovalAngelToPoint	ChopEllipseConnector	20
MDIDesktopManager.resizeDesktop	MDIDesktopPane	20

Table 6. Results Obtained by Applying MCODE and FG to the uFDN of JHotDraw.

Approach	#Community	Rand index
MCODE	34	0.456
FG	32	0.414

talked above) that should be moved to the target class Packet. It proves that CLEAR can successfully move back the misplaced methods with a high stability.

As a third goal, we want to compare the effectiveness of our approach with one refactoring approach at the class level and other two community detection algorithms, specifically the search-based refactoring approach (SBRA) [16], a graph theoretic clustering algorithm (MCODE) [3], and a fast algorithm for community detection (FG) [10]. As the focus of the current work is not on the introduction of these approaches, the detailed description of MCODE and FG is omitted. It should also be noted that since [16] only reports the results obtained on JHotDraw, here we also compare the effectiveness of these approaches only on JHotDraw.

Compared with SBRA, we find that the methods that should be moved by CLEAR are very similar to that reported in [16], but the time cost is less. The time for CLEAR is <20 s, whereas the time for SBRA is more than 30 min. We apply MCODE and FG to the uFDN of JHotDraw and compare the communities detected with the real class structures by computing the Rand index [15]. The Rand index is a measure of the similarity between two data clusterings. The results are shown in Table 6.

From Table 6, we can see that MCODE detects 34 communities in the uFDN of JHotDraw, whereas FG detects 32 communities. The number of communities detected by the two approaches is much smaller than the number of classes in JHotDraw. Thus, the identified communities would hardly be mapped to the real classes, preventing us from the comprehension of the results. The Rand index further indicates that the communities detected by the two approaches only weakly related to the real classes. The two approaches prove useless when applied to software networks. Such a disparity between communities and real classes may come from the initial setting of their approaches, i.e., the two algorithms start with random communities. However, CLEAR starts with the communities that represent original software classes. Therefore, CLEAR can obtain meaningful results better than the MCODE and FG.

5 Conclusions and Future Work

In this article, we have proposed an approach for identifying the methods that should be moved between classes. Our approach is performed from the perspective of software topological structure and uses a bipartite network across class and feature level to characterize the structure. Nodes denote classes and features. Edges between every pair of class and feature denote their composition relationships, and the weight on each edge denotes the dependency strength. Based on the bipartite network, we introduced a bipartite modularity metric to quantify the cohesion and coupling of the software as a whole. Further, a COEA that takes the bipartite modularity metric as its fitness function is proposed to find the optimized class structures. By comparing the optimized class structures with the real class structure, we can detect the methods that should be moved and their target classes.

We conducted two case studies to assess the proposed approach. Our manual checking of the suggested methods indicated that the proposed approach is capable of extracting sound suggestions.

Although our approach shows some feasibility, the broad validity of our approach demands further demonstration. Thus, the future work include (1) evaluating the approach using other large-scale open-source Java projects, (2) implementing more refactorings such as extracting classes and splitting classes, and (3) developing a refactoring tool that can refactor software systems at different levels of granularity.

Acknowledgments: This work was supported by the Zhejiang Provincial Nature Science Foundation of China under Grant Nos. LQ14F020006, LY14F020035, and LY12F02014.

Bibliography

[1] A. Alkhalid, M. Alshayeb and S. A. Mahmoud, Software refactoring at the class level using clustering techniques, *J. Res. Pract. Inf. Technol.* **5** (2011), 285–306.

[2] M. J. Barber, Modularity and community detection in bipartite network, *Phys. Rev. E* **76** (2007), 066102.

[3] G. D. Bader and C. Hogue, An automated method for finding molecular complexes in large protein interaction networks, *BMC Bioinform.* **30** (2003), 121–141.

[4] I. F. Cruz and R. Tamassia, Graph drawing tutorial. http://www.cs.brown.edu/people/rt/papers/gd-tutorial/gd-constraints.pdf (accessed 29 May 2012).

[5] D. P. Darcy, C. F. Kemerer, S. A. Slaughter and J. E. Tomayko, The structural complexity of software: an experimental test, *IEEE Trans. Software Eng.* **31** (2005), 982–995.

[6] S. Demeyer, F. V. Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. D. Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall and M. El-Ramly, The LAN-simulation: a refactoring teachning example, in: *Proceedings of the 8th International Workshop on Principles of Software Evolution*, pp. 121–134, IEEE, Lisbon, Portugal, 2005.

[7] C. F. Dormann, A method for detecting modules in quantitative bipartite networks, *Method Ecol. Evol.* **5** (2014), 90–98.

[8] M. Fowler, *Refactoring: Improving the Design of Existing Code*, pp. 260–266, Addison-Wesley, New York, 1999.

[9] M. Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA, 1998.

[10] M. E. J. Newman, Fast algorithm for detecting community structure in networks, *Phys. Rev. E* **69** (2004), 066133.

[11] M. O'Keeffe and M. O'Cinneide, Search-based software maintenance, in: *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, 22–24 March 2006, pp. 249–260, IEEE, Bari, Italy, 2006.

[12] W. F. Pan, B. Li, Y. T. Ma and J. Liu, Multi-granularity evolution analysis of software using complex network theory, *J. Syst. Sci. Complex.* **24** (2011), 1068–1082.

[13] W. F. Pan, B. Jiang and B. Li, Refactoring software packages via community detection in complex software networks, *Int. J. Autom. Comput.* **10** (2013), 157–166.

[14] W. F. Pan, B. Li, B. Jiang and K. Liu, Recode: software package refactoring via community detection in bipartite software networks, *Adv. Complex Syst.* (2014), DOI: 10.1142/S0219525914500064.

[15] W. M. Rand, Objective criteria for the evaluation of clustering methods, *J. Am. Stat. Assoc.* **66** (1971), 846–850.

[16] O. Seng, J. Stanmmel and D. Burkhart, Search-based determination of refactoring for improving the class structure of object-oriented system, in: *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pp. 1909–1916, ACM, Seattle, WA, 2006.

[17] A. Sinha, P. Malo, A. Frantsev and K. Deb, Finding optimal strategies in a multi-period multi-leader–follower Stackelberg game using an evolutionary algorithm, *Comput. Oper. Res.* **41** (2014), 374–385.

[18] L. Tahvildari and K. Kontogiannis, A metric-based approach to enhance design quality through meta-pattern transformations, in: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pp. 183–192, IEEE, Benevento, Italy, 2003.

[19] A. Trifu and R. Marinescu, Diagnosing design problems in object-oriented systems, in: *Proceedings of the 12th Working Conference on Reverse Engineering*, PA, pp. 155–164, IEEE, Pittsburgh, 2005.

[20] N. Tsantalis and A. Chatzigeorgious, Identification of move method refactoring opportunities, *IEEE Trans. Software Eng.* **35** (2009), 347–367.

[21] H. Wang, C. Y. Xu, J. B. Hu and K. F. Cao, A complex network analysis of hypertension-related genes, *Physica A* **394** (2014), 166–176.