International Journal of Automation and Computing 10(2), April 2013, 157-166
DOI: 10.1007/s11633-013-0708-y

Refactoring Software Packages via Community Detection in Complex Software Networks

Wei-Feng Pan[1,2] Bo Jiang[1] Bing Li[2]

[1]School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, China

[2]State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China

Abstract: An intrinsic property of software in a real-world environment is its need to evolve, which is usually accompanied by the increase of software complexity and deterioration of software quality, making software maintenance a tough problem. Refactoring is regarded as an effective way to address this problem. Many refactoring approaches at the method and class level have been proposed. But the research on software refactoring at the package level is very little. This paper presents a novel approach to refactor the package structures of object oriented software. It uses software networks to represent classes and their dependencies. It proposes a constrained community detection algorithm to obtain the optimized community structures in software networks, which also correspond to the optimized package structures. And it finally provides a list of classes as refactoring candidates by comparing the optimized package structures with the real package structures. The empirical evaluation of the proposed approach has been performed in two open source Java projects, and the benefits of our approach are illustrated in comparison with the other three approaches.

Keywords: Refactoring, community detection, complex networks, package, software.

1 Introduction

The original design of a software system is rarely prepared for every new requirement appearing over the life cycle of software system, such as fault corrections, performance improvements, and adaption to new environments. Software has to evolve to accommodate them. However, due to the tight schedule in real life software development process, change has to be made quickly by different people, which usually lowers its quality[1].

Refactoring, proposed by Fowler, is regarded as an effective way to improve the design of the code, both at the time of the original development and during the maintenance of legacy code[2, 3]. However, software developers have to identify parts of code which have a negative impact on system′s maintainability, and apply appropriate refactorings in order to remove the so-called "bad-smells"[4]. But it is a challenging and time-consuming task to decide which refactoring to apply and where to apply it[5].

Complex network theory provides an effective tool to explore the whole structure of a system and its dynamics. In recent years, a few researchers introduced the complex network theory to software engineering domain, and used complex networks in software, hereafter referred to as software networks, to represent software structure at a higher level. Based on software networks, a lot of work has been carried out[6−9]. And it has also opened new and broad opportunities for software refactoring. In our primary work[3], we have proposed to represent object-oriented (OO) software by complex software networks, and used a community detection technique from complex network theory to refactor software at the class level. Empirical results show that introducing complex network theory into software refactoring is promising and can produce better performance than other approaches. But, to the best of our knowledge, how such an approach can be extended to software refactoring at the package level is still a problem which has never been explored.

The objective of this paper is to use the community detection technique in complex network theory as a method to help software developers do refactoring at the package level. Firstly, the proposed approach builds an undirected weighted class dependency network to describe the macro-topological structure of software at the class level, where classes are nodes and the interaction between every pair of classes if any is an edge. And we also assign a weight to each node to reflect the dependency strength of the two connected classes. Secondly, a constrained community detection algorithm is proposed to detect the community structures which correspond to the optimized package structures in the undirected weighted class dependency network. Finally, a list of classes will be suggested as refactoring candidates simply by comparing the optimized package structures with the corresponding real package structures in software systems.

The rest of this paper is organized as follows. Section 2 contains a brief summary of the related works. Section 3 describes our approach in detail, with focus on the formal definitions of related software networks, the procedures to refactor the package structures, and the algorithm we used to optimize community structures in the undirected weighted class dependency network. Section 4 presents the results of two case studies conducted on open source software systems. And we conclude the paper in Section 5.

Manuscript received April 11, 2012; revised December 17, 2012
This work was supported by National Natural Science Foundation of China (No. 61202048), Zhejiang Provincial Nature Science Foundation of China (No. LQ12F02011), and Open Foundation of State Key Laboratory of Software Engineering of Wuhan University of China (No. SKLSE-2012-09-21).

2 Related works

This section is a brief, but for reasons of space, incomplete, overview of the related works on source code refactoring. It mainly falls into three categories according to where the refactoring activities occurred: method level refactoring, class level refactoring, and package level refactoring.

2.1 Method level refactoring

The works belonging to this category try to detect statements poorly structured in methods. Some representative works of this category are as the following:

Maruyama and Shima[10] presented a mechanism to automatically refactor methods by using weighted program dependence graphs based on the methods′ modification histories. Atkinson and King[11] presented a low-cost, syntactic approach to automatically discover extract method refactoring opportunities. Tsantalis and Chatzigeorgiou[12] also proposed a methodology to automatically identify extract method refactoring opportunities. Kanemitsu et al.[13] presented a program dependency graph (PDG) visualization method based on the data connection strength between sentences in the source code to identify extract method refactoring opportunities.

2.2 Class level refactoring

The works belonging to this category try to detect attributes and methods poorly structured in classes. Some representative works of this category are as the following:

Tahvildari and Kontogiannis[14] proposed a framework using a catalogue of OO metrics as indicators, to suggest refactoring opportunities. Trifu and Marinescu[15] proposed to use detection strategies as a mean to detect instances of a structural anomaly. O′Keeffe and O′Cinneide[16] formulated the task of refactoring as a search problem guided by a quality evaluation function in the space of alternative designs. Seng et al.[5] proposed a search based approach, namely, genetic algorithm to support move-method refactoring. Tsantalis and Chatzigeorgiou[17] proposed the notion of distance between attributes/methods and classes. Besides, based on it, they presented a method for the identification of move-method refactoring opportunities. In our primary work[3], we have proposed to represent software at method/attribute level by unweighted software networks, and used the community detection technique to refactor software at the class level of granularity.

2.3 Package level refactoring

The works belonging to this category try to detect classes poorly structured in packages. Some representative works of this category are as the following:

Hautus[18] defined the package structure analysis (PASTA) metric for evaluating the quality of package structures, and implemented a tool that can assist the developers in developing a proper package structure through analysis and visualization. Melton and Tempero[19] presented a tool that can analyze the large cycles in dependency graphs (CDGs) in order to identify classes as possible refactoring candidates. Melton and Tempero[20] further developed a simple metric, class reachability set size (CRSS), that can be used to determine whether the system is with a good package design or not and to identify candidates for refactoring. Alkhalid et al.[21] investigated software refactoring at the package level using clustering techniques.

3 The approach

From the above review of the related works, it is clear that most research efforts mainly focus on refactoring at the class level. To the best of our knowledge, there has been little discussion about refactoring at the package level. Melton, Hautus, and Alkhalid are the only three pioneers. In general, the approaches they proposed use structural metrics to capture the inter-relationships between software entities (methods, attributes, classes, and packages), such as method calls, attribute references and class inheritances, to guide refactoring operations. Indeed, the structural metrics they used mainly focus on the local properties of software (e.g., dependency cycles, class-package similarity, etc.). Due to the lack of suitable tools and theories, people seldom investigate software refactoring at the package level from the perspective of software as a whole.

Though this is not the first work on software refactoring at the package level, we will cover a different angle, i.e., from the perspective of software as a whole using community detection techniques. Fig. 1 gives a short overview of the workflow of the proposed approach. In the following sections, we will detail it.

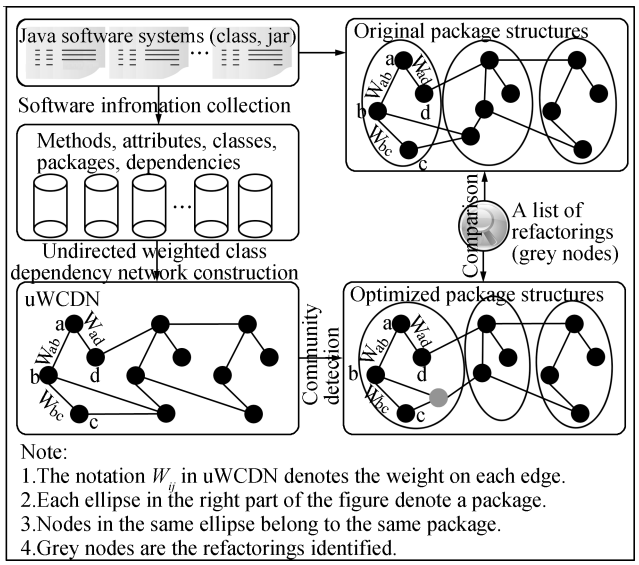


Fig. 1 The workflow of the proposed approach

3.1 Java software

This paper mainly focuses on the OO domain, and takes the open source Java software systems as research subjects. The rationale is threefold[9]: 1) OO has become the most widely used development paradigm since 1990 s. And there are a lot of open source OO software systems with sufficient supplement materials on the web which can be easily got for

our research objectives. 2) OO software systems have a relatively clear internal structure and the components, such as methods, classes/interfaces, packages, and their dependencies are amenable to extraction and analysis. 3) The choice of Java programming language is limited by the developed tools to perform analysis, and our interest in understanding software written in Java.

3.2 Software information collection

In order to construct the software networks, entities in the source code should be collected firstly. Specifically, the entities we care about in the current work mainly include attributes, methods, classes, packages and their dependencies. And only two kinds of dependencies are taken into consideration, i.e., method accessing attribute dependency and method call dependency. The dependencies between classes are obtained from these two kinds of dependencies. A dependency between two methods or method and attribute in two separate classes implies a dependency between the two classes.

3.3 Software network definition

Based on the collected data, two types of software networks, specifically undirected feature dependency network (uFDN) and undirected weighted class dependency network (uWCDN), can be built. We use the term feature to designate attributes and methods.

Definition 1. In uFDN, nodes denote the features of a specific OO software system, and each feature is represented by only one node. Edges between two nodes indicate the use dependency between the corresponding features, i.e., if feature A uses feature B, there is an edge between the nodes denoting the two features. And here we only consider the presence of dependency and neglect the multiplicity of dependencies such as A depends three time on B and its direction. Therefore, uFDN can be described as

$$\text{uFDN} = (V_f, E_f, W_f) \tag{1}$$

where V_f is the set of all nodes in uFDN (the subscript f denotes that the software network is built at the feature level), E_f is the set of edges, and W_f is a symmetric adjacency matrix. The entry $w_f(i,j)$ at position (i,j) is 1 if there is an edge between node i and node j, and is 0 otherwise.

Fig. 2 shows a simple source code segment and its corresponding uFDN, where W_f is described as matrix (2).

$$W_f = \begin{bmatrix} & a & b() & c() & d() & e() & f() \\ a & 0 & 0 & 0 & 1 & 0 & 0 \\ b() & 0 & 0 & 1 & 1 & 0 & 0 \\ c() & 0 & 1 & 0 & 1 & 0 & 1 \\ d() & 1 & 1 & 1 & 0 & 0 & 0 \\ e() & 0 & 0 & 0 & 0 & 0 & 1 \\ f() & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}. \tag{2}$$

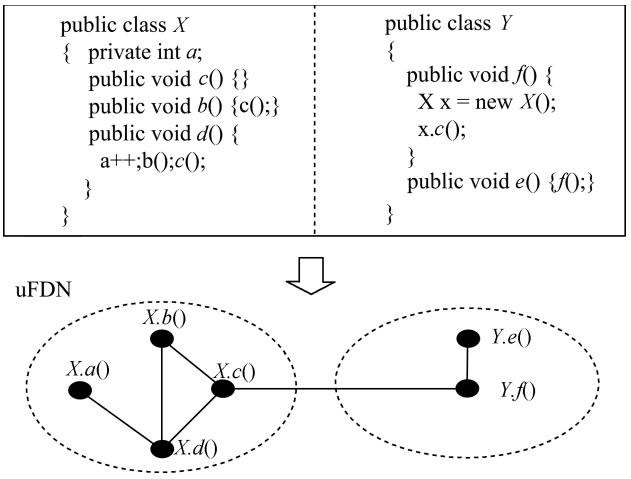


Fig. 2 Illustration of uFDN

Definition 2. In uWCDN, nodes denote the classes of a specific OO software system and each class is represented by only one node. Edges between two nodes indicate a certain dependency between the corresponding classes. Such dependencies are obtained from the dependencies between features they enclosed, i.e., a dependency between two methods or method and attribute in two separate classes implies a dependency between the classes. Each edge is also weighted with a value to signify the dependency strength, such as class X depends three times on class Y. Therefore, uWCDN can be written as

$$\text{uWCDN} = (V_c, E_c, W_c) \tag{3}$$

where V_c is the set of all nodes in uWCDN (the subscript c denotes the software network is at the class level), E_c is the set of edges, and W_c is a symmetric weight matrix. The entry $w_c(i,j)$ at position (i,j) is an integer no less than 1 if there is an edge between node i and node j, and is 0 otherwise. Obviously, W_c is also a symmetric adjacency matrix where a non-zero entry denotes an edge between the corresponding nodes.

Introduction of weights brings a flexibility that allows us to consider the dependency strength between classes, but it also raises a new problem: determining the weights. In this paper, we will use the dependencies between the features in two classes to quantify the weight on the corresponding edge between the two classes. Denote R_{ik} as the set of all reachable nodes originated from node i within a distance k, and F_i as the set of all features class i contains. Then, the weight $w_c(i,j)$ (or $w_c(j,i)$) is defined as

$$w_c(i,j) = w_c(j,i) = |\bigcup_{n_i \in F_i} R_{i1} \cap F_j| = |\bigcup_{n_j \in F_j} R_{j1} \cap F_i| \tag{4}$$

where $|*|$ denotes the number of elements in set $*$.

Indeed, small value of weight indicates the low dependency between the two classes. It is desirable to keep the weight as small as possible for a specific software system as far as software maintainability is concerned.

Fig. 3 shows the corresponding uWCDN of uFDN in Fig. 2. Here, the uWCDN has only two nodes, i.e., $V_c = \{X, Y\}$. Since $Y.f()$ in class Y directly depends on

$X.c()$ in class X, there is an edge between X and Y in uWCDN. At the same time, $R_{Y.f()1} = \{X.c(), Y.e()\}$ and $R_{Y.e()1} = \{Y.f()\}$. So $w_c(X,Y) = |\{X.c(), Y.e()\} \cup \{Y.f()\} \cap \{X.a, X.b(), X.c(), X.d()\}| = |\{X.c()\}| = 1$.

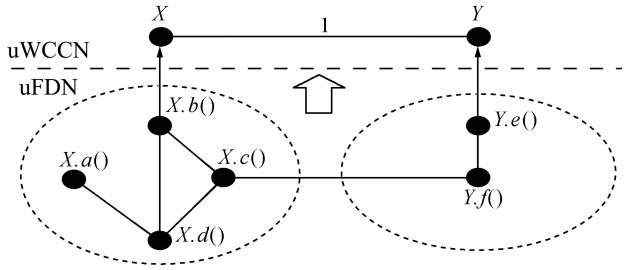


Fig. 3 Illustration of uWCDN

3.4 Constrained community detection algorithm

Community structure, the gathering of nodes into groups such that there is a higher density of edges within groups than between them, is one of the network features that has been emphasized in recent complex network research[22]. A popular method now widely used is to optimize a quality index for a partition of a network into communities. And the approach proposed in the current work is also in this line of research.

As we all know, OO software systems consist of a set of packages, which in turn are composed of a set of classes. Such an organization of software entities forms an interesting community structures, i.e., the packages are the natural communities of classes, and the classes are the natural communities of features. As a result of many design principles (e.g., low coupling and high cohesion), a majority of classes are in the right packages, while only several misplaced classes lowering the quality of software systems are needed to be moved. So there is no need to start the community detection process in a random way with every class belonging to a random community as many community detection algorithms do in other domains[23].

Our approach starts from a state in which every class belongs to a specific community. The community, in essence, is the package where the class is defined. And our approach proceeds by a series of class-moving operations at classes with dependencies to other classes that are not defined in the same package. We propose a constrained community detection algorithm to fulfill this task.

3.4.1 Quality index

The quality index used to evaluate one partition is of vital importance to our approach, for it controls the class-moving process. This paper uses modularity Q[23] devised by Newman and Girvan as the quality index simply for its popularity. But the original Q is proposed to detect communities in unweighted networks. In order to make it suitable for weighted networks, we use its weighted version which is given as

$$Qw = \sum_i (we_{ii} - {wa_i}^2) \tag{5}$$

where Qw is the quality index of a particular partition, we_{ii} is the fraction of the total weight of the edges that connect two nodes within community i, while wa_i is the fraction of the total weight of the edges that have at least one endpoint within community i.

In the process of community detection, Qw will be recalculated on every class-moving operation to decide whether to accept or reject this movement. So how to calculate Qw will greatly influence the performance of our algorithm even in the cases that the number of classes is very large. This paper calculates ΔQw, the change in Qw, rather than calculating Qw. Because if a class is moved to a worse package (i.e., increase the coupling), there will be a decrease in modularity Qw. And if a class is moved to a better package, there will be an increase in Qw. The value of ΔQw denotes the suitability of a class-moving operation. Thus, to find the community that one class should be moved to means to find the largest ΔQw. Such a strategy to accelerate the speed of the algorithm is borrowed from [3] and [23].

The change in Qw, ΔQw, upon moving a class from community i to community j is given by

$$\Delta Qw = \begin{cases} we_{ij} + we_{ji} - 2\ wa_i\ wa_j, & \text{if community } i \text{ and } j \text{ are connected} \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

In our approach, the algorithm travels through every class (node) with dependencies on other classes not defined in the same package (community), iteratively searches for the changes resulted from class-moving operation, and moves the class to the package that makes the largest increase in Qw.

3.4.2 Constrained community detection algorithm flow

We propose a constrained community detection algorithm (CCDA) to fulfill the community detection task in uWCDN, which is shown as Algorithm 1. Here, array $Wc[][]$ stores the symmetric weight matrix W_c in uWCDN. $Wc[i][j] \geqslant 1$ means there is an edge between node i and node j with weight $Wc[i][j]$, otherwise there is no edge. $nodeCom[]$ is an array storing the community identifiers for all nodes, e.g. node i belongs to the community with identifier $nodeCom[i]$. $bVisited[]$ is an array with type boolean denoting whether node i has been visited or not.

Algorithm 1. CCDA algorithm

Input:

uWCDN (actually the largest weakly connected component of the whole uWCDN)

Output:

Qw and a list of classes that should be moved

Procedure:

1: Initialize $Wc[][]$, $nodeCom[]$ (nodes (classes) in the same package will be assigned the same community identifier), and $bVisited[] =$ false, $i = 0, j = 0$

2: Calculate Qw according to (5)

3: **for** $i = 1$ to $|N_c|$ do

4: **for** $j = 1$ to $|N_c|$ do

5: **if** $W_c[i][j] \geqslant 1 \&\& nodeCom[i] \neq nodeCom[j]$

```
        && !bVisited[i] then
6:          bVisited[i] = true
7:          Suppose move node i to community
            nodeCom[j] and calculate ΔQw according
            to (6), and store it into an array ΔQw[ ]
8:      end if
9:    end for
10:   Select the maximum ΔQw, ΔQw_max
11:   if ΔQw_max > 0 then
12:     Move node i to community nodeCom[j] that
        produces the largest ΔQw
13:     for j = 1 to i do
14:       if W_c[i][j] ⩾ 1 then
15:         bVisited[i] = false
16:       end if
17:     end for
18:     i = 1
19:     Qw = ΔQw + Qw
20:   end if
21: end for
22: return Qw and a list of classes that should be moved
```

4 Experiment and data analysis

In this section, we describe in detail the subjects, process and results of two case studies carried out to assess the proposed approach.

4.1 Subjects

The experiments were carried out on two open source Java software systems, namely Trama[24], and Front End for MySQL Domain (denoted as Font4MySQL)[25]. Trama is a tool that provides different graphical user interface to help user to easily work with matrices. Font4MySQL is a portable front end to the open source database server MYSQL, providing a complete front end and thus reducing the difficulty caused by using SQL query language. The two systems are selected as they are introduced in [21]. Using the same subjects lays a basis as comparing our approach with that they proposed.

Table 1 reports the size, in terms of thousand lines of code (KLOC), number of packages (#P), number of classes (#C), number of features (#F), and the versions of the systems used in our study. We should point out that the #P excludes the outer packages, #C includes the number of inner classes, and KLOC is the practical lines of code, excluding the comment lines and blank lines. For a clear presentation, here we have omitted the details about the organization of software entities such as classes in every package and features in every class. For details, please refer to the data set provided on homepage[26].

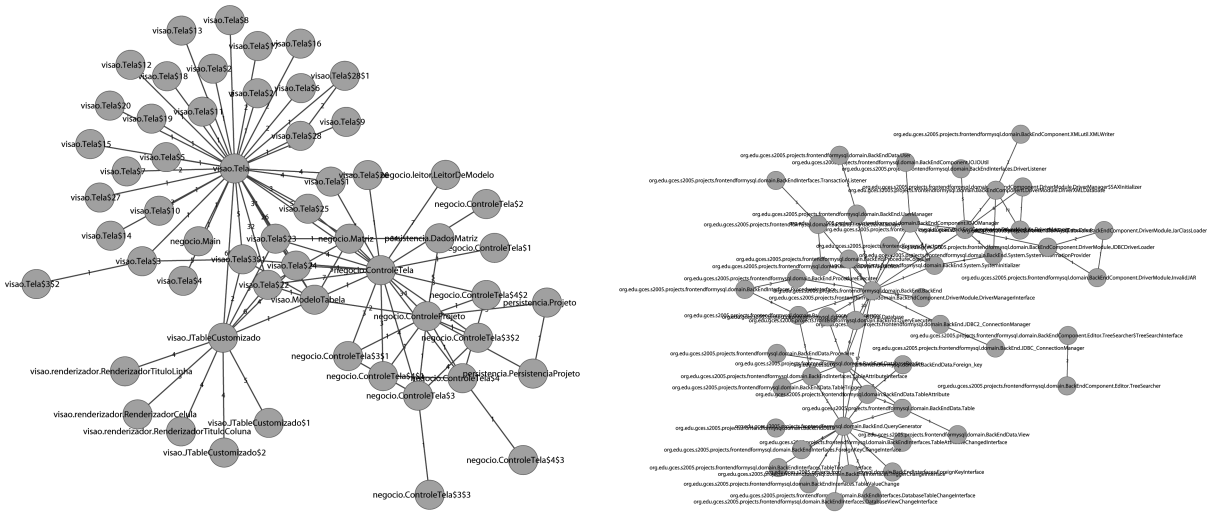
Table 1 Systems used in case studies

System	Version	KLOC	#P	#C	#F
Trama	1.0	4.019	6	57	546
Font4MySQL	1.0	3.187	8	51	553

4.2 Case studies and results

In this section, we follow the main steps shown in Fig. 1 to refactor the package structures of the two systems listed above. Our experiments were carried out on a PC at 2.30 GHz with 2 GB of RAM.

The software information (mentioned in Subsection 3.2) and the software networks (i.e., uFDN and uWCDN) used in this paper are all automatically generated by our own developed software analysis tool SNAT (software network analysis tool)[9]. It can parse the compiled Java code (files with .class and .jar extension), extract the relevant information, and further build the uFDN and uWCDN.



(a) uWCDN for Trama

(b) uWCDN for Font4MySQL

Fig. 4 uWCDNs for all systems under study

Figs. 4 (a) and (b) show the uWCDNs for all the systems under study. Enlarging the networks can give more information about the uWCDNs, such as the class each node denotes (the label beside the node is the name of the class), and the dependency between two classes if there is an edge between two nodes. The value on each edge is the weight between the two classes. The positions of nodes in uWCDNs are calculated using a force-directed layout algorithm[27].

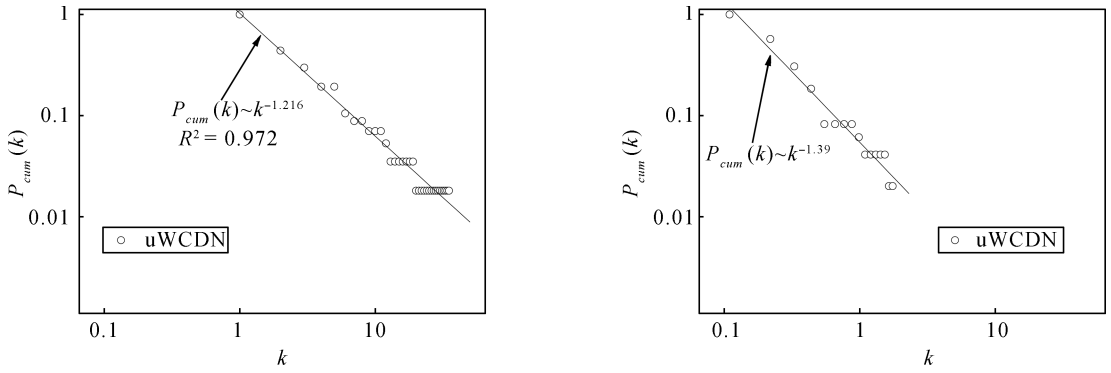
In order to clarify some properties of the uWCDNs, a statistical analysis has been carried out. Table 2 lists the basic statistical results of the tested software systems. In this table, $|V|$ is the number of nodes, $|E|$ is the total number of edges, d is the average distance, $\bar{C}$ is the average clustering coefficient, $d_{\rm rand}$ and $\bar{C}_{\rm rand}$ are the d and $\bar{C}$ of the corresponding random network with the same $|V|$ and $|E|$, K is the average degree, #WCC is the number of weakly connected components (WCC), and LWCC is the number of nodes belonging to the largest WCC. And we should point out that d is calculated among reachable pairs, not including the isolated nodes which have no edge to other nodes such as DocumentEditor and StatusBarDataStructure in Font4MySQL. Also Fig. 4 does not show the isolated nodes. Fig. 5 shows the cumulative degree distributions, $P_{cum}(k)$ for uWCDNs.

It is interesting to find that uWCDNs for the two systems are of small world type, with their d being very close to $d_{\rm rand}$ of the corresponding random networks and $\bar{C}$ being much larger. Further, we can also observe all $P_{cum}(k)$ fit power-law like tails, obeying a power law $P_{cum}(k) \sim k^{-\alpha}$. And the measures for goodness of fit (R^2) are higher than 0.962. Thus, uWCDNs are also scale free networks. And we also observe that both systems consist of a single LWCC, comprising a large fraction of the total nodes in the system (ranging from 80.97 % to 100 %), and a few very small remaining WCCs. Here, we only use the LWCC as the input of our community detection algorithm. It is reasonable from the perspective of statistics.

Table 2 Basic statistical results of the two software systems

Software	$\|V\|$	$\|E\|$	d	$d_{\rm rand}$	$\bar{C}$	$\bar{C}_{\rm rand}$	K	#WCC	LWCC
Trama	57	91	2.521	3.483	0.308	0.056	3.193	1	57
Font4MySQL	51	66	3.003	3.967	0.264	0.053	2.694	4	47

Tables 3 and 4 show the classes should be moved for each system using CCDA. The first column is the moving order of the classes, the second column contains the classes suggested to be moved, the third column is the original package the class is defined, and the last column shows the suggested the target package.



(a) $P_{cum}(k)$ for Trama (b) $P_{cum}(k)$ for Font4MySQL

Fig. 5 Cumulative degree distributions $P_{cum}(k)$ for uWCDNs of the two subject systems (log-log scale)

Table 3 Classes should be moved for Trama

Order	Class name	Original package	Target package
1	PersistenciaProjeto	persistencia	negocio
2	Projeto	persistencia	negocio
3	Matriz	negocio	persistencia
4	ModeloTabela	visao	persistencia
5	ControleTela	negocio	visao
6	LeitorDeModelo	leitor	visao
7	ControleTela$1	negocio	visao
8	ControleTela$2	negocio	visao
9	Main	negocio	visao
10	RenderizadorCelula	renderizador	visao
11	RenderizadorTituloColuna	renderizador	visao
12	RenderizadorTituloLinha	renderizador	visao

Table 4 Classes should be moved for Font4MySQL

Order	Class name	Original package	Target package
1	Factory	Frontendformysql	System
2	ProcedureGenerator	BackEnd	BackEndInterfaces
3	QueryGenerator	BackEnd	BackEndInterfaces
4	TransactionManager	BackEnd	BackEndData
5	UserManager	BackEnd	BackEndData
6	IOManager	IO	System
7	Database	BackEndData	BackEnd
8	ProcedureExecuter	BackEnd	BackEndInterfaces
9	DriverManagerInterface	DriverModule	BackEnd
10	DatabaseReader	BackEnd	IO
11	Tuple	BackEndData	BackEndInterfaces
12	DatabaseViewChangeInterface	BackEndInterfaces	IO
13	Driver	BackEndData	DriverModule
14	DriverListener	BackEndInterfaces	DriverModule
15	XMLWriter	XMLutil	DriverModule
16	IOUtil	IO	System

4.3 Analysis and comparison

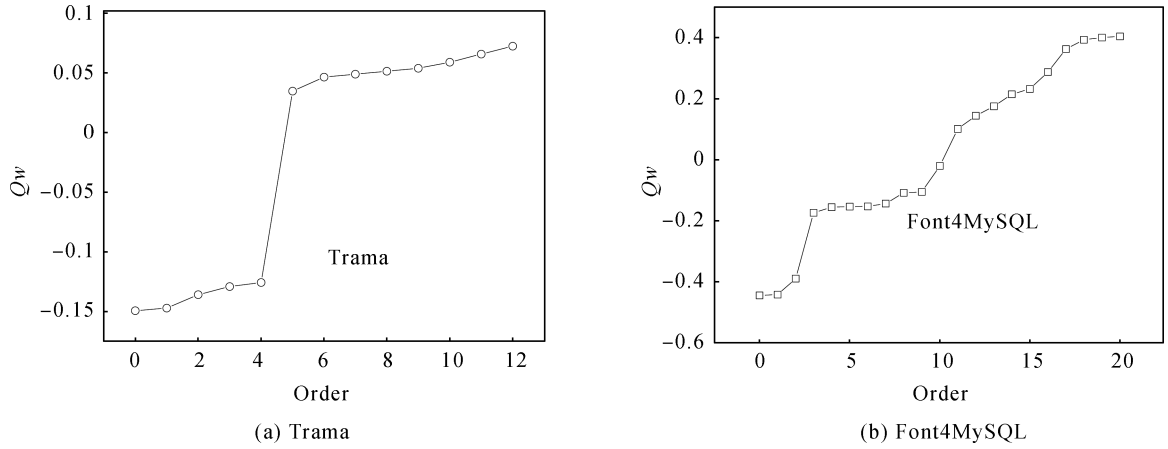
As shown in Tables 3 and 4, our approach suggested 12 classes and 16 classes to be moved for Trama and Font4MySQL, respectively. In order to judge whether these refactorings make sense to the developer, we manually checked them one by one. By referring to the source code files (with .java extension), we found that all the proposed refactorings can be justified.

In system Trama, classes PersistenciaProjeto and Projeto are suggested to be moved from the original package persistencia to negocio, because all these classes are heavily used by (or use) class ControleProjeto in negocio ($w_c = 5$ and 3 respectively) than that in persistencia ($w_c = 1$). Moving them from persistencia to negocio can reduce the coupling, and improve the modularity by 8.977%. Class Matriz is recommended to be moved to package persistencia as it is more heavily used by (or uses) class DadosMatriz in persistencia ($w_c = 34$). Class ModeloTabela′s refacotring mainly results from Matriz′s refactoring. Classes ControleTela, ControleTela$1, and ControleTela$2 can be similarly justified. Our approach also suggests to move class Main from package negocio to visao as it is only used by (or uses) class Tela in package visao, and does not use (or used by) classes inside the same package where it is defined. Classes LeitorDeModelo, RenderizadorCelula, RenderizadorTituloColuna, and RenderizadorTituloLinhacan can be similarly justified. Fig. 6 (a) shows the evolution curve of the modularity with each class movement. The order in Table 3 is used as abscissa in Fig. 6 (a).

The same observation is valid for the second system Font4MySQL. Classes Factory, UserManager, and ProcedureExecuter are recommended to be moved as their movements can improve the modularity of software by 0.659 %, 1 %, and 24.355 %, respectively. Classes ProcedureGenerator, QueryGenerator, TransactionManager, IOManager, and DatabaseReader are suggested to be moved from the original packages to the target packages, as all these classes are heavily used by (or use) classes in the target packages than that in the original packages. Moving them from the original package to the target package can improve the modularity by 11.865 %, 55.293 %, 10.694 %, 0.385 %, and 80.378%, respectively. Classes Database, DriverManagerInterface, DatabaseViewChangeInterface, Driver, DriverListener, and XMLWriter are suggested to be moved as they are only used by (or use) the classes in target packages, and do not use (or used by) classes inside the same packages where they are defined. Class Tuple′s movement mainly results from Database′s movement, and IOUtil′s movement results from Tuple′s movement. It is because Tuple is only used by (or uses) Database, and IOUtil is only used by (or uses) Tuple. Fig. 6 (b) shows the evolution curve of the modularity with each class movement. The order in Table 4 is used as abscissa in Fig. 6 (b).

To provide more confidence on the results obtained by our approach, we asked five software engineers to provide their judgments on the results. And we also ask them to manually detect the list of refactorings for the two systems. We provided the five engineers with the package designs before and after refactoring and asked them the following question: Which design is better in terms of high cohesion and low coupling? All the five engineers confirmed that the results obtained by our approach, maximized the package cohesion and lowered the package coupling, and thus improved the software modularity.

Further, the effectiveness of our approach is evaluated by recall and precision[28], two metrics that have been widely used in pattern recognition and information retrieval. Here, we borrow these two metrics from information retrieval, and adapt them to fit in with the software package refactoring problem.



(a) Trama (b) Font4MySQL

Fig. 6 The curve of modularity Qw versus order (class moving steps) over the running of CCDA

In software refactoring, precision is the fraction of recommended classes that are right (confirmed to be refactored). It is given by

$$\text{Precision} = \frac{|\{\text{Recommended right classes }\}|}{|\{\text{Recommended classes}\}|}. \quad (7)$$

Recall is the fraction of the classes that should be refactored that are successfully recommended. It is defined as

$$\text{Recall} = \frac{|\{\text{Recommended right classes}\}|}{|\{\text{Classes should be refactored}\}|}. \quad (8)$$

Here the classes that should be refactored in each system are provided by the five engineers mentioned above via manually checking every class in the system. And the recommended classes are automatically provided by a specific approach. Indeed, as for systems Trama and Font4MySQL, classes that should be refactored are the same as the list of classes we have provided in Tables 3 and 4. And the recommend right classes are obtained by making an intersection of the two set of classes.

We calculate the precision and recall of CCDA on the two systems to compare with the approaches cited in related works, i.e., CDGs, CRSS, and A-KNN. In CDGS and CRSS, the recommended classes are the top-12 classes participating in the largest number of cycles and the top-16 classes with the largest CRSS values, respectively. The results are shown in Table 5.

Table 5 Precision and recall

Approach	System	Precision	Recall
CCDA	Trama	100 %	100 %
	Font4MySQL	100 %	100 %
CDGs	Trama	58.33 %	58.33 %
	Font4MySQL	25 %	25 %
CRSS	Trama	58.33 %	58.33 %
	Font4MySQL	25 %	25 %
A-KNN	Trama	66.67 %	16.67 %
	Font4MySQL	60 %	18.75 %

Compared with CDGs, CRSS, and A-KNN, the advantages of the proposed approach can be illustrated as follows:

1) Obviously, it can be seen from Table 5 that the precision and recall of CCDA are 100 %, better than those of CDGs, CRSS, and A-KNN. The classes obtained by our approach are equal to the classes provided by the five engineers, while the other three approaches only contain part of the classes provided by the five engineers. Such difference may result from different optimization strategies: Our approach is carried out from the perspective of software modularity. It is a global optimization process. However, CDGs, CRSS, and A-KNN are proceeded by calculating local metrics such as CRSS value and similarity. It is a local optimization process. 2) The computational complexity of CCDA is better than that of A-KNN. The most dominant steps of CCDA are the construction of software networks, and Step 3 to Step 21 with the loop. Hence, the computational complexity of CCDA is $O(|N_c|^2)$. However, the computational complexity of A-KNN is $O(|N_c^2|A)$ as reported in [21], where A is the number of attributes of the classes. But we cannot compare the computational complexity of CCDA with that of CDGs and CRSS, since we have no idea about their specific implementations in [19, 20].

Our approach provides a new perspective for software refactorings. It only uses community detection technology in software networks to find the meaningful refactorings. It makes software refactoring a very simple task.

5 Conclusions and future work

Software refactoring at the package level of granularity has great importance for the software quality and maintenance tasks. However, deciding which refactoring to apply and where to apply it is nontrivial. In this paper, we have proposed an approach for identifying refactoring opportunities in packages. The proposed approach uses an undirected weighted software network at class level of granularity to represent classes and their dependency, where each node represents a class in the system. The weight of an edge that connects two nodes (classes) is a measure of the dependency strength. A constrained community detection algorithm is proposed to find the communities in such a software network. These communities are the optimized package structures which can be used to detect the classes

that should be moved.

We conducted two case studies to assess the proposed approach. Our manual checking and the assessment by independent engineers of the suggested refactorings for the two systems indicated that the proposed approach is capable of extracting sound suggestions. We also carried out a comparative study with the other three approaches on software refactoring at package level, highlighting the benefits such as better performance and lower computational complexity.

Although our approach shows some feasibilities, the broad validity of our approach demands further demonstration. Thus, the future work include: 1) Evaluate the approach using other open source software systems with different scales and from different domains. 2) Implement more refactorings such as extracting package and splitting package. 3) Develop a refactoring tool which can refactor software systems at different levels of granularity.

References

[1] T. Mens, T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

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

[3] W. F. Pan, B. Li, Y. T. Ma, J. Liu, Y. Y. Qin. Class structure refactoring of object-oriented softwares using community detection in dependency networks. *Frontiers of Computer Science in China*, vol. 3, no. 3, pp. 396–404, 2009.

[4] W. J. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crises*, New York, USA: John Wiley and Sons, pp. 47–56, 1998.

[5] O. Seng, J. Stanmmel, D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ACM, New York, USA, pp. 1909–1916, 2006.

[6] C. R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, vol. 68, no. 4, 046116, 2003.

[7] A. Potanin, J. Noble, M. Frean, R. Biddle. Scale-free geometry in OO programs. *Communications of the ACM*, vol. 48, no. 5, pp. 99–103, 2005.

[8] G. Concas, M. Marchesi, S. Pinna, N. Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 687–708, 2007.

[9] W. F. Pan, B. Li, Y. T. Ma, Y. Y. Qin, X. Y. Zhou. Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks. *Journal of Computer Science and Technology*, vol. 25, no. 6, pp. 1202–1213, 2010.

[10] K. Maruyama, K. Shima. Automatic method refactoring using weighted dependence graphs. In *Proceedings of the 21st International Conference on Software Engineering*, ACM, Los Angeles, CA, USA, pp. 236–245, 1999.

[11] D. C. Atkinson, T. King. Lightweight detection of program refactorings. In *Proceedings of the 11th Working Conference on Reverse Engineering*, IEEE, Taipei, Taiwan, China, pp. 663–670, 2005.

[12] N. Tsantalis, A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[13] T. Kanemitsu, Y. Higo, S. Kusumoto. A visualization method of program dependency graph for identifying extract method opportunity. In *Proceedings of the 4th Workshop on Refactoring Tools*, ACM, New York, USA, pp. 8–14, 2011.

[14] L. Tahvildari, 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*, IEEE, Benevento, Italy, pp. 183–192, 2003.

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

[16] M. O′Keeffe, M. O′Cinneide. Search-based software maintenance. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, IEEE, Washington, DC, USA, pp. 249–260, 2006.

[17] N. Tsantalis, A. Chatzigeorgious. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[18] E. Hautus. Improving Java software through package structure analysis. In *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications*, IEEE, Cambridge, USA, 2002.

[19] H. Melton, E. Tempero. Identifying refactoring opportunities by identifying dependency cycles. In *Proceedings of the 29th Australasian Computer Science Conference*, Australian Computer Society, Hobart, Australia, pp. 35–41, 2006.

[20] H. Melton, E. Tempero. The CRSS metric for package design quality. In *Proceedings of the 30th Australasian Computer Science Conference*, Australian Computer Society, Ballarat, Victoria, Australia, pp. 201–210, 2007.

[21] A. Alkhalid, M. Alshayeb, S. A. Mahmoud. Software refactoring at the package level using clustering techniques. *IET Software*, vol. 5, no. 3, pp. 276–284, 2011.

[22] S. Fortunato. Community detection in graphs. *Physics Reports*, vol. 486, no. 3–5, pp. 75–174, 2010.

[23] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Physical Review E*, vol. 69, no. 6, 066133, 2004.

[24] M. Fabio. Trama, [Online], Available: http://sourceforge.net/projects/trama, April 10, 2012.

[25] S. Shrestha, Y. Gurung. Front End For MySQL, [Online], Available: http://sourceforge.net/projects/ frontend4mysql, April 10, 2012.

[26] W. F. Pan. Weifeng Pan′s homepage, [Online], Available: http://www.whucn.com/wfpan.htm, April 10, 2012.

[27] I. F. Cruz, R. Tamassia. Graph drawing tutorial, [Online], Available: http://www.cs.brown.edu/people/ rt/papers/gd-tutorial/gd-constraints.pdf, April 10, 2012.

[28] J. Makhoul, F. Kubala, R. Schwartz, R. Weischedel. Performance measures for information extraction. In *Proceedings of DARPA Broadcast News Workshop*, Herndon, VA, USA, pp. 249–252, 1999.

Wei-Feng Pan received his Ph. D. degree from State Key Laboratory of Software Engineering (SKLSE) at Wuhan University (WHU), China in 2011. He is presently a lecture at School of Computer Science and Information Engineering (SCIE), Zhejiang Gongshang University (ZJGSU), China. He is also a member of China Computer Federation (CCF) and Association for Computing Machinery (ACM).

His research interests include software engineering, service computing, complex networks, and intelligent computation.

E-mail: panweifeng1982@gmail.com (Corresponding author)

Bo Jiang received her Ph. D. degree from Zhejiang University (ZJU) in computer science in 2007. She is presently a professor and M. Sc. supervisor at School of Computer Science and Information Engineering (SCIE), Zhejiang Gongshang University (ZJGSU), China. She is also a senior member of China Computer Federation (CCF) and a member of Association for Computing Machinery (ACM).

Her research interests include social computing and service computing.

E-mail: nancybjiang@mail.zjgsu.edu.cn

Bing Li received his Ph. D., M. Sc. and B. A. degrees from Huazhong University of Science and Technology (HUST), China in 2003, 1997 and 1990, respectively, all in computer science. He is presently a professor and Ph. D. supervisor at State Key Laboratory of Software Engineering (SKLSE) and School of Computer at Wuhan University (WHU), China. He is also a senior member of China Computer Federation (CCF) and a member of Association for Computing Machinery (ACM).

His research interests include requirements engineering, cloud computing, complex network, and semantic web service.

E-mail: bingli@whu.edu.cn