Zhou *et al. Applied Network Science* (2020) 5:50
https://doi.org/10.1007/s41109-020-00298-8

Applied Network Science

RESEARCH Open Access

Measurement and refactoring for package structure based on complex network

Yangxi Zhou, Yanran Mi, Yan Zhu and Liangyu Chen*

*Correspondence:
lychen@sei.ecnu.edu.cn
School of Software Engineering, East China Normal University, No.3663 North Zhongshan Road, 200062 Shanghai, China

Abstract

Software structure is the backbone for software systems. During the long time of software evolution, it is gradually weakened by continuous code modification and expansion driven by new requirements. Therefore, measuring software and refactoring codes are necessary to keep software structure stable and clean. In this paper, we propose two metrics of cohesion and coupling to characterize package structure. We consider not only the dependencies of intra-package and inter-package, but also the backward dependencies of inter-package. The two metrics are proved theoretically that they are satisfied with Briand's four properties. Based on these metrics, a refactoring algorithm is presented to improve the quality of package structure. Through tests on ten open source software systems, the experiment result shows our metrics can measure software structure correctly and improve codes to fit for the rule of high cohesion and low coupling.

Keywords: Software dependency network, Software metric, Software measurement, Software refactoring, High cohesion and low coupling

Introduction

It is well known that software lifecycle has two phases: a development phase and a maintenance phase. In the development phase, programmers make codes carefully under the guidelines of software architecture design, such as the rule of high cohesion and low coupling. Comparing to the development phase, the maintenance phase is much longer and can last for several years. During the long time of maintenance, the software is not stationary, but evolves gradually. Driven by the new requirements, software functionalities are continuously updated and refactored. Therefore, the amount of code increases and it also becomes more and more complicated. This may cause the software to deviate from the original design, and result in the degradation of software quality and comprehensibility, and finally generate a "technical debt" [1]. So it is necessary to keep software architecture stable and codes clean during the evolution to prolong software service life (Tom et al. 2013).

[1] Technical debt is the term used to describe the time/money/resources that will need to be spent in order to rebuild a software system that is already been "completed".

Faced with the increasing software functionality and complexity, it needs careful protection on software architecture without function degradation. Refactoring, is one of powerful tools to improve software design and increase maintainability and usability for software systems (Fowler 1997). Through code modification and software structure adjustment, refactoring makes software clean, which can pass code review and get good software measurement. However, simple refactoring with code modification in manual, is time consuming and has little effort. Thus, some researchers are looking for guidelines for automatically refactoring based on software metrics. On the other side, there are some researchers in software engineering focusing on the dynamic characteristics of software structure with methods in complex network. Based on the combination of complex network and software engineering, a software system can be represented into a network, then transferred into a unified object for evolution analysis.

In this paper, based on complex network theory, we present two metrics about package cohesion and coupling, to measure software quality better and guide refactoring automatically. Compared with previous work(Mi et al. 2019), the new metrics take into account overall dependencies between classes, and also consider the backwards dependencies of classes. To check the validity of our metrics, we first prove they strictly meet the four properties proposed by Briand (Briand et al. 1996). Based on these two metrics, we also provide a refactoring algorithm to adapt package-class relations for better balance of cohesion and coupling. Finally, through several experiments on multiple open source software systems, we verify the validity of our metrics, and efficiency of the refactoring algorithm.

We have presented a preliminary version of software measurement and refactoring in (Mi et al. 2019). Beside a general revision and improvement, this paper extends our previous work in the following directions:

- Besides the cohesion metric, we also present the coupling metric. The combination of cohesion and coupling can measure software package structure more objective and clear. We also prove the new coupling metric is strictly satisfied with the properties proposed by Briand (Briand et al. 1996).
- Cohesion and coupling are correlated but not overlapped. Bias to any one metric is not good to measure software correctly. Based on the relation of cohesion and coupling metrics, we present an evaluation model of package structure, then update the refactoring algorithm.
- We compare our new refactoring algorithm with other algorithms. In the experiment of disturb and recover, under different disturb ratios, our algorithm can find almost disturbed classes and place them back to the correct packages.

The remainder of this paper is structured as follows. In "Related work" section, we describe the current work of software measurement and complex network in software engineering. In "Fundmentals of software codes" section, we introduce some concepts of software codes and code dependencies. In "Software network and its attributes" section, we present the construction of software network and its related attributes. In "Our metrics" section, we describe the new metrics of cohesion and coupling, and prove them validity in theory, then give a new algorithm for package refactoring. In "Experiments and analysis" section, we design several experiments to verify the validity

of our metrics and the efficiency of refactoring algorithm. Finally, we conclude our work in "Conclusion" section.

Related work

A software system can be evaluated "GOOD", with not only satisfying the functionality requirements, but also meeting the design and programming guidelines. The typical one is high cohesion and low coupling. In these guidelines, many metrics have been developed to measure software quality. These metrics have different levels of granularity, from the basic program variable to the whole architecture structure. According to the implementation, these metrics can be divided into two categories: statistics-type and network-type.

We first review the statistics-type metrics, which are calculated based on different levels of programming objects, such as class, package and system. we list as follows.

(a). For the class level, Chidamber and Kemerer proposed the *CK* metric set for OO design evaluation (Chidamber and Kemerer 1994). *CK* is a group of six metrics on class and method, which are pure static statistics. Lee et al. proposed a dynamic metric of information-based coupling (*ICP*), which defines the coupling degree for every class based on information flow through method invocations (Lee 1995). Harrison et al. proposed *MOOD* metric set, including *CF* (coupling factor) (Harrison et al. 1998), which can measure cohesion indirectly through measuring coupling. It is calculated by the sum of all possible dependencies between classes, divided by the sum of existed dependencies of all classes. Bieman et al. proposed two cohesion metrics: *TCC* (Tight Class Cohesion) and *LCC* (Loose Class Cohesion), based on the instance variables shared in different methods (Bieman and Kang 1995).

(b). For the package level, Martin proposed seven statistical metrics (Martin 2002), which are easily implemented and widely used in many development tools to assist programmers. Misic studied on the package cohesion, and concluded that relying solely on the internal relationships of packages is not sufficient to determine cohesion (Misic 2001). Sarkar et al. proposed a new metric suite to characterize the modular quality of software packages (Sarkar et al. 2008). Abdeen et al. proposed a cohesive metric based cyclic dependence (Abdeen et al. 2009), to evaluate package modularity, encapsulation, variability, and reusability.

(c). For the system level, Gui et al. used an approach like (Lee 1995) to define system coupling as the average coupling of all classes in a software system (Gui and Scott 2006). This is a system-level coupling measurement that can be used to evaluate component reusability.

Next, we summarize the network-type metrics and their implementation. Compared to the common statistics methods, complex network technologies have become more and more popular since they provide a macro perspective to analyze software. Software systems can be represented as complex networks, which also called software networks. In a software network, nodes are software entities, such as methods, fields, classes, or packages, and edges represent dependencies between entities (Pan et al. 2011). Therefore, many recent studies have focused on software networks, and reached many useful results (Potanin et al. 2005; Concas et al. 2007; Pan et al. 2010; Pan et al. 2011).

One of the most interesting properties in complex network is community structure (Girvan and Newman 2002; Newman 2006; Fortunato 2010). Communities are generated

by dividing a network into several parts, based on the rules of tight internal connections and sparse external connections. Communities play an important role in understanding network characteristics (Li et al. 2013) (Pan et al. 2011). Software networks have the same community properties with other networks, such as being small world and scale-free (Myers 2003; Pan et al. 2011). Shen et al. made in-depth research on several Java software systems and found that the relevant networks contain same features (Ping-ting and Liang-yu 2017). Pan et al. represented Java software systems as bipartite networks, and proposed an algorithm to reconstruct the organizational structure of software packages (Pan et al. 2014). With the widespread application of complex network technology on software networks, many related tools are developed to analyze existed software systems. Zheng et al. used an improved degree model to analyze the Linux system (Zheng et al. 2008). Besides community structure analysis, complex network technologies have also been applied in the field of software evolution. Valverde et al. proposed a model based on node replication and edge rearrangement (Valverde and Solé 2005). He et al. proposed a model based on the growth of software design patterns (He et al. 2006). Li et al. proposed a software evolution model combining complex network theory and evolutionary algorithms (Li et al. 2006).

Fundmentals of software codes

Software codes

Software systems are made from codes written by professional programmers according to practical functionality requirements. These codes must obey the syntax rules and structure requirements of programming languages. Take *Java* as a typical language, the common program structure of object-oriented (OO) software systems is two-tier: class and package. All codes must be enclosed in mutiple class files, then these class files are collected in different packages according to their functionalities and roles. Class is the basic unit. And package, as an intermediate layer, can play the role of aggregating classes and regulating class access as well as can reduce system complexity and increase maintainability and understandability. Note that the rationality of package organization affects software quality to a certain degree.

For better demonstration, we show a Java system in Fig. 1. In this system, there are three packages. Detailedly, *package1* has three classes: *A*, *B*, *C*; *package2* has also three classes: *D*, *E*, *F*, and the last package *package3* has three classes: *H*, *I*, *J*.

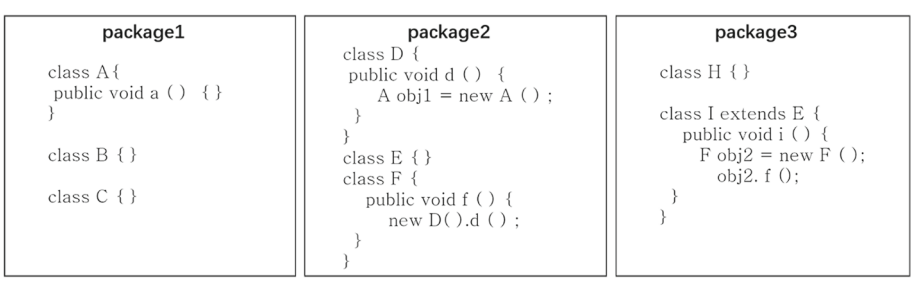

Fig. 1 An example of Java software system

Code dependencies

When a call occurs between two classes, a dependency is created. There are several types of dependencies in oriented-object programming languages. Kang et al. summarized code dependencies between classes into ten relations (Kang et al. 2004). These relations have different weights in the classical theory of software engineering, however, to our best knowledge, there is no authoritatively quantitative values for weighting factors.

Additionally, the influence of packages on dependencies can not be omitted, since packages also contribute to system dependencies. As a middle tier, package can aggregate classes with same role or functionality, and limit outside illegal access. Thus, the dependencies can be labelled into two types: intra-package dependencies and inter-package dependencies. An intra-package dependency means its caller and callee are in the same package, while the two participators of inter-package dependency locate in different packages.

High cohesion and low coupling

Programming languages are continuously growing with more and more powerful features, such as function encapsulation, inheritance and polymorphism. These features make codes implement the requirements right and efficiently, while they have to cause the dependency problem (Tom et al. 2013). As shown in Fig. 1, codes are split into several class files, where they generate many dependencies among them. For example, class I inherits class E and calls the function of class F. That means, class I are depended on the classes E and F. If there are some changes in classes E or F, it must put impacts on class I. Unfortunately, the dependencies are inevitable since it cannot put all codes into one file.

From Fig. 1, we can see that a class has higher risk of unstable modification if it has more dependencies on other classes. Therefore, programmers try to get rid of class dependencies, and make codes self-contained. This is called the famous principle of high cohesion and low coupling. The cohesion indicates the degree of every program module, like class or package, can finish its functionality with the support from inner codes. Conversely, the coupling presents how a module depends on other outside modules. To avoid the cascade modification and latent bugs, one of the promising methods is to make codes high cohesion and low coupling. The ideal system is one where all modules remain independent without any dependencies. Unfortunately, that is very difficult because of the massive and complex software requirements. Therefore, programmers need to design and implement changes carefully to increase code cohesion and decrease module coupling.

Software network and its attributes

Class dependency graph network

Definition 1 *In this paper, the software systems we studied are made from oriented-object programming languages. Thus, based on the package-class structure and the dependencies between classes, the software system can be represented as a Class Dependency Graph (CDG) network (Ping-ting and Liang-yu 2017), which is a directed graph. Let* $G = (V_c, E_c, C)$ *denote a CDG, where* V_c *is the set of vertexes/classes,* E_c *is the set of edges/dependencies, and* C *is the set of communities/packages respectively. Every package is mapped to be a community of the network. In this directed network, there is an edge* $v_i \rightarrow v_j$ *if and only if there is at least one following dependency between* v_i *and* v_j*:*

- R_1—*Inheritance and implementation:* v_i *extends or implements* v_j*;*
- R_2—*Aggregation:* v_j *is the data type of member variable in* v_i*;*
- R_3—*Parameter:* v_j *is the data type of parameter/return value/declared exception of member function in* v_i*;*
- R_4—*Signature:* v_j *is the type of local member variable in* v_i*;*
- R_5—*Invocation:* v_j *is invoked insides the member function in* v_i*;*

In other words, for a node, its outgoing edges denote the classes it depends on, that is, forwards dependencies. Similarly, the incoming edges mean the classes it supports, namely backwards dependencies. Additionally, for generality, we assume that the weights of above five dependencies are same, then the dependency between two classes is weighted by the add up of all the dependencies.

We use a software system developed with Java language as an example, to show how to construct a *CDG* network. Corresponding to the source codes shown in Fig. 1, we can generate the *CDG* shown in Fig. 2. Different to the existing coarse-granularity software networks, our *CDG* describes the software structure deeply and clearly, since it is based on five fine-granularity dependencies $\{R_1, R_2, \ldots, R_5\}$.

In Fig. 1, there are three packages. Detailedly, *package1* has three classes: *A*, *B*, *C*; *package2* has also three classes: *D*, *E*, *F*, and the last package *package3* has two classes: *H* and *I*. It is easily observed that there are four dependencies between classes: *D* depends on *A*, *F* depends on *D*, *I* depends on *E*, and *I* depends on *F* . Based on the definitions of five dependencies $\{R_1, R_2, \ldots, R_5\}$, the *CDG* is generated and shown in Fig. 2.

Attributes of software network

Definition 2 *Let* $G = (V, E, C)$ *stand for a directed network, where V, E, C denote the set of vertices, edges and communities respectively. Note that for the networks generated from software systems, the communities are formed based on the package-class structure naturally. Each vertex belongs to only one community, and there is no common vertices between communities, that is,* $\forall i \neq j$, $C_i \cap C_j = \varnothing$. m_{ij} *denotes the value between* v_i *and* v_j *in adjacency matrix M. And for two vertices* v_i, v_j *in a software network, if* v_i, v_j *belong to in the same community, then* $\alpha(v_i, v_j) = 1$. *Otherwise* $\alpha(v_i, v_j) = 0$.

Definition 3 *An internal edge is an edge whose two vertices are located in the same community. The number of internal edges in a community, is calculated with*

$$WPR = \sum_{i,j} m_{ij}\alpha(v_i, v_j). \tag{1}$$

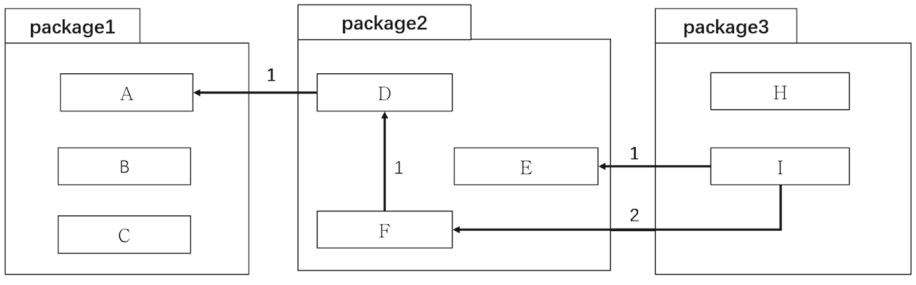

Fig. 2 An Example of *CDG*

Corresponding to a software network, WPR indicates the cohesion maturity for a package, since the internal edges are located in a package, that is, the package don't need any outer dependencies. Obviously, the larger WPR is, the greater cohesion is.

Definition 4 *An external edge is an edge whose two vertices are in two different packages. There are two types of external edges: outgoing edges and incoming edges. For a community, the number of outgoing edges is calculated with*

$$WPER = \sum_{i \in C_k} m_{ij}(1 - \alpha(v_i, v_j)), \tag{2}$$

while the number of incoming edges is

$$WPAR = \sum_{j \in C_k} m_{ij}(1 - \alpha(v_i, v_j)). \tag{3}$$

Corresponding to a software network, WPER means the "powerful" degree of a package which can support other packages. And WPAR is the "dependent" degree of a package which needs more support on other packages. Thus, for a package, the larger WPER is, the more important is; the larger WPAR is, the more dependent is.

Definition 5 *We can quantify the importance for a package. Let PRE indicate the number of other packages that a package depends upon, and PRA denote the number of other packages that depend on a package. The related calculation is listed as follows:*

$$PRE = \sum_{ij} \gamma(C_i, C_j), \tag{4}$$

$$PRA = \sum_{ij} \gamma(C_j, C_i). \tag{5}$$

In formulas (4) and (5), when classes in C_l depends upon classes in C_k, $\gamma(C_l, C_k) = 1$. Otherwise, $\gamma(C_l, C_k) = 0$.

Our metrics

It is well known that the rule of high cohesion and low coupling is very important in software architecture design. The degree of cohesion and coupling between packages, has a great impact on software maintainability and reusability. However, manual evaluation for cohesion and coupling is time consuming and labor intensive. So it is necessary to construct evaluation metrics and algorithms without manual operations, for better code evaluation and refactoring.

Cohesion and coupling metrics

In (Abdeen et al. 2009), Abdeen proposed a cohesion metric packages. For a package, this metric considers not only the intra-package dependencies, but also the inter-package dependencies. However, it omits the backwards dependencies, namely the case that a class is dependent on others. From the perspective of software quality, the inter-package calls brought by the backwards dependencies, have a high probability of affecting overall package reusability. Considering the affect of backwards dependencies, we define a new cohesion metric *COHM*, for measuring software package cohesion.

$$COHM = \frac{WPR}{WPR + WPER + \delta \cdot WPAR}. \tag{6}$$

When $WPR + WPER + \delta \cdot WPAR = 0$, *COHM* is set as 0. Though *WPER* and *WPAR* both denote inter-package dependencies, the influence of backwards dependencies on a class is smaller than the forwards dependencies' influence. Note that backwards dependencies are passive and not controlled by the callee class. However, a class can control its forwards dependencies. So, we multiply *WPAR* with an arbitrary coefficient δ less than 1 to emphasize that it's less important than *WPER*. Here, we tentatively fix $\delta = 0.5$. According to the meaning of cohesion, it is easily known that the larger value of *COHM* indicates higher cohesion.

Inspired by Martin's efferent and afferent couplings (Martin 2002), we also propose a new package coupling metric *COUM*. This metric considers the relations between packages caused by all relations between classes, which can truly reflect the hierarchical relations between packages. The *COUM* for one package is calculated as follows:

$$COUM = \frac{PRE + PRA}{WPR + PRE + PRA}, \tag{7}$$

where *WPR* represents the number of times the package depends on itself. Note that when the denominator is 0, *COUM* is set as 0. In this formula, the numerator denotes the sum of the number of associations between the community and other communities, and the denominator represents the sum of the number of all associations in the community. According to the meaning of coupling, the smaller the *COUM* value is, the lower the degree of package coupling is.

We present Algorithm 1 to calculate the metrics of cohesion and coupling for a package. In this algorithm, N is the number of all classes and N_{p_1} denotes the number of classes in the package p_1. We iterate all classes in the package p_1 to calculate cohesion and coupling. It is observed that we only consider the case that two classes have dependencies. Next, if the two classes are in the same package, we add up the dependency values from M to get the value of *WPR*, otherwise, we get the value of *WPER* or *WPAR* according to the direction of the dependency. Finally calculate cohesion and coupling for package p_1 after visit all classes.

Let's analyze the complexity of Algorithm 1. It is easily seen that there are a nested two-layers loop in Algorithm 1. Assume the number of classes of whole software is N, and for a package P_1, it has N_{P_1} classes. Then the outer loops runs N_{P_1} times, while the inner loop runs N times. Therefore, the complexity of Algorithm 1 is $O(N_{P_1}N)$. When performing Algorithm 1 on all packages, the total complexity is $O((N_{P_1} + N_{P_2} + \cdots + N_{P_x}) \cdot N)$. Since $N_{P_1} + N_{P_2} + \cdots + N_{P_x} = N$, the total complexity is $O(N^2)$. Remark that the total value of *COHM* and *COUM* for a software system, are set as the average values of all packages' counterparts.

Theoretical verification

The concept of cohesion and coupling has been used to represent the dependencies between modules. Briand et al. defined some mathematical properties to characterize the cohesion and coupling (Briand et al. 1996). Such a mathematical framework can generate a consensus in the software engineering community, provide better guidelines for communication among researchers, and better evaluation methods for commercial analyzers and practitioners. These properties are necessary and helpful to prove the usefulness of cohesion/coupling measurement although not completely precise.

Algorithm 1: Calculation of cohesion and coupling for a package

Input: Packages C, Adjacent matrix M, vertex number N, the package P_1.

Output: The cohesion and coupling values for P_1.

```
Initialize PRE, PRA, WPR as 0.
for i in N_{P_1} do
    for j in N do
        if M_ij != 0 then
            if C_vj = P_1 then
                WPR+ = M_ij;
            else
                WPER+ = M_ij;
                if C_vj doesn't be computed then
                    PRE++;
                end
            end
        end
    end
    for j in N do
        if M_ji != 0 then
            if C_vj != P_1 then
                WPAR+ = M_ji;
                if C_vj doesn't be computed then
                    PRA++;
                end
            end
        end
    end
end
calculate COHM with formula (6);
calculate COUM with formula (7);
return the result.
```

Here, we use *CDG* in Fig. 2 as an example to illustrate the calculation of *COHM* and *COUM*. For *package2*, class *F* depends on class *D* in the same package, class *D* depends on class *A* in *package1*, class *E* is depended on class *I* in *package3* and class *F* is depended on class *I* in *package3*. So, $WPR = 1$, $WPER = 1$, and $WPAR = 3$. Since *package2* depends on *package1* and *package3* is depend on *package2*, we have $PRE = 1$, and $PRA = 1$. According to the formulas (6) and (7), $COHM = 0.29$, and $COUM = 0.67$.

Here, we verify theoretically the validation of our inter-package cohesion and coupling by analyzing their mathematical properties. Briand presented five properties for cohesion and coupling. The definitions are listed as follows.[2]

[2] For generality and refinement, we combine properties 4 and 5 in (Briand et al. 1996) into one property, namely PROPERTY 4 including two parts: cohesion and coupling.

PROPERTY 1: Non-negativity The cohesion and the coupling of a modular system|modular is nonnegativity.
PROPERTY 2: Null Value If there is no intramodule relationship among the elements of a (all) module(s), then the module (system) cohesion is null. And If there is no intermodule relationship among the elements of a (all) module(s), then the module (system) coupling is null.
PROPERTY 3: Monotonicity Adding intramodule relationships does not decrease [module|modular system] cohesion. And adding intermodule relationships does not decrease [module|modular system] cohesion.
PROPERTY COHESION 4: Merging of Modules The cohesion of a [modulelmodular system] obtained by putting together two unrelated modules is not greater than the [maximum cohesion of the two original modules I the cohesion of the original modular system].
PROPERTY COUPLING 4: Merging of Modules The coupling of a [moduleImodular system] obtained by merging two modules is not greater than the [sum of the couplings of the two original moduleslcoupling of the original modular system], since the two modules may have common intermodule relationships.

Proposition 1 *Formulas (6) and (7) are satisfied with four verification properties proposed by Briand.*

Proof **(1) Non-negativity**
In formula (6), WPR, $WPER$, δ, $WPAR$ are all non-negative, therefore $COHM$ is also non-negative. In formula (7), PRE, PRA, and WPR are all non-negative, thus $COUM$ is also non-negative.
(2) Null Value
If the number of classes in a package is zero or the package has no relation with any other packages, that is, WPR, $WPER$, $WPAR$, PRE, PRA are all zero, then both denominator of $COHM$ and $COUM$ will be null.
(3) Monotonicity
There are two cases of adding new edges to a package. One is adding internal edges in a package, the other is linking external edges between different packages. For the first case, when adding some internal edges for a package C, we use C' to denote the new package. Then for $COHM$ monotonicity, we have

$$COHM_{C'} - COHM_C = \frac{(WPER_C + \delta \cdot WPAR_C) \cdot (WPR_{C'} - WPR_C)}{(WPR_{C'} + WPER_{C'} + \delta \cdot WPAR_{C'}) \cdot (WPR_C + WPER_C + \delta \cdot WPAR_C)},$$

since $WPER_C$ and $WPAR_C$ aren't changed under the condition of adding internal edges. Obviously, $WPR_{C'} > WPR_C$. So both denominator and numerator are non-negative, then $COHM$ is increasing monotonously. For $COUM$ monotonicity, we have

$$COUM_{C'} - COUM_C = \frac{(WPR_C - WPR_{C'}) \cdot (PRE_C + PRA_C)}{(WPR_{C'} + PRE_{C'} + PRA_{C'}) \cdot (WPR_C + PRE_C + PRA_C)},$$

since PRE_C and PRA_C aren't changed under the condition of adding internal edges. Obviously, $WPR_{C'} > WPR_C$, then the denominator is non-negative and the numerator is negative, so *COHM* is decreasing monotonously.

For the second case of adding external edges for a package C, let C' denote the new package. As to *COHM* monotonicity, we have

$$\begin{aligned} &COHM_{C'} - COHM_C \\ &= \frac{WPR \cdot (WPER_C + \delta \cdot WPAR_C - WPER_{C'} - \delta \cdot WPAR_{C'})}{(WPR_{C'} + WPER_{C'} + \delta \cdot WPAR_{C'}) \cdot (WPR_C + WPER_C + \delta \cdot WPAR_C)}, \end{aligned}$$

since *WPR* isn't changed under the condition of adding external edges. Obviously, $WPER_{C'} \geq WPER_C$ and $WPAR_{C'} \geq WPAR_C$, thus, the denominator is non-negative and the numerator is negative. So that, *COHM* is decreasing monotonously. As to *COUM* monotonicity, we have

$$\begin{aligned} &COUM_{C'} - COUM_C \\ &= \frac{WPR_C \cdot (PRE_{C'} + PRA_{C'} - PRE_C - PRA_C)}{(WPR_{C'} + PRE_{C'} + PRA_{C'}) \cdot (WPR_C + PRE_C + PRA_C)}, \end{aligned}$$

since *WPR* doesn't change under the condition of adding external edges. Obviously, $PRE_{C'} \geq PRE_C$ and $PRA_{C'} \geq PRA_C$, thus, both denominator and numerator are non-negative. So that, *COUM* is increasing monotonously.

To sum up the two above cases, adding the internal edges in a package, will increase *COHM* and decrease *COUM*; while add the external edges between different packages, will decrease *COHM* and increase *COUM*. These changes are coincident with the rule of high cohesion and low coupling. Therefore, we prove both *COHM* and *COUM* satisfies the monotonicity property.

(4) Merging of Modules

Without loss of generality, assume that two packages(modules) C_a, C_b, where all classes in package C_a have no dependencies or backwards dependencies on the classes in the package C_b. The cohesions of package C_a and C_b are calculated as follows:

$$\begin{aligned} COHM_{C_a} &= \frac{WPR_{C_a}}{WPR_{C_a} + WPER_{C_a} + \delta \cdot WPAR_{C_a}}, \\ COHM_{C_b} &= \frac{WPR_{C_b}}{WPR_{C_b} + WPER_{C_b} + \delta \cdot WPAR_{C_b}}. \end{aligned}$$

Then we combine C_a and C_b to generate a new package C_c. The cohesion of C_c is list as follows:

$$COHM_{C_c} = \frac{WPR_{C_a} + WPR_{C_b}}{WPR_{C_a} + WPER_{C_a} + \delta \cdot WPAR_{C_a} + WPR_{C_b} + WPER_{C_b} + \delta \cdot WPAR_{C_b}}.$$

We use $\frac{N_a}{D_a}$ to denote $COHM_{C_a} - COHM_{C_c}$, where

$$\begin{aligned} D_a &= (WPR_{C_a} + WPER_{C_a} + \delta \cdot WPAR_{C_a}) \cdot \\ &\quad (WPR_{C_a} + WPER_{C_a} + \delta \cdot WPAR_{C_a} + WPR_{C_b} + WPER_{C_b} + \delta \cdot WPAR_{C_b}), \\ N_a &= WPR_{C_a} \cdot (WPER_{C_b} + WPAR_{C_b}) - WPR_{C_b} \cdot (WPER_{C_a} + WPAR_{C_a}). \end{aligned}$$

We also use $\frac{N_b}{D_b}$ to denote $COHM_{C_b} - COHM_{C_c}$, where

$$\begin{aligned} D_b &= (WPR_{C_b} + WPER_{C_b} + \delta \cdot WPAR_{C_b}) \cdot \\ &\quad (WPR_{C_a} + WPER_{C_a} + \delta \cdot WPAR_{C_a} + WPR_{C_b} + WPER_{C_b} + \delta \cdot WPAR_{C_b}), \\ N_b &= WPR_{C_b} \cdot (WPER_{C_a} + WPAR_{C_a}) - WPR_{C_a} \cdot (WPER_{C_b} + WPAR_{C_b}). \end{aligned}$$

Obviously, $D_a, D_b > 0$, and $N_a = -N_b$, therefore $COHM_{C_a} - COHM_{C_c} \geq 0$ or $COHM_{C_b} - COHM_{C_c} \geq 0$ holds. Namely, $COHM_{C_c} \leq \max\{COHM_{C_a}, COHM_{C_b}\}$. In other words, the new cohesion of merged package is not bigger than two original cohesions.

For the coupling metric, the fourth property proposed by Braind requires $COUM_{C_a} + COUM_{C_b} \geq COUM_{C_c}$. For the new merged package C_c, we have

$$\begin{aligned} COUM_{C_c} &= \frac{PRE_{C_c} + PRA_{C_c}}{WPR_{C_c} + PRE_{C_c} + PRA_{C_c}} \\ &= \frac{PRE_{C_a} + PRE_{C_b} + PRA_{C_a} + PRA_{C_b}}{WPR_{C_a} + WPR_{C_b} + PRE_{C_a} + PRE_{C_b} + PRA_{C_a} + PRA_{C_b}}. \end{aligned}$$

Then, we can use $\frac{N_c}{D_c}$ to denote $COUM_{C_a} + COUM_{C_b} - COUM_{C_c}$, where

$$\begin{aligned} D_c &= (WPR_{C_c} + PRE_{C_c} + PRA_{C_c}) \cdot (WPR_{C_c} + PRE_{C_c} + PRA_{C_c}) \cdot \\ &\quad (WPR_{C_c} + PRE_{C_c} + PRA_{C_c}), \\ N_c &= (PRE_{C_a} + PRA_{C_a}) \cdot (WPR_{C_b} + PRE_{C_b} + PRA_{C_b})^2 + \\ &\quad (PREb + PRA_{C_b}) \cdot (WPR_{C_a} + PRE_{C_a} + PRA_{C_a})^2. \end{aligned}$$

It is easy to see both D_c and N_c are all non-negative. So, $COUM_{C_a} + COUM_{C_b} \geq COUM_{C_c}$. That proves the fourth property.

To sum up, we have proved that our metrics of cohesion and coupling are satisfied with all properties proposed by Briand. □

Refactoring algorithm

As mentioned above, for a software system, programmers pursue the goal of high cohesion and low coupling. Note that these two parts are not interchangeable. In software engineering, we tend to think the influence of cohesion and coupling are equally important. When we consider only one of them, we are not able to know the software system correct and clear. Therefore, combining cohesion with coupling can better reflect package modularity and fully measure software structure. Thus, we propose a refactoring algorithm based on *COUM* and *COHM* metrics to optimize software structure. Our algorithm is based on the principle of greedy algorithm to pursue high cohesion and low coupling. The detail of refactoring algorithm is summarized in Algorithm 2.

First, we move a candidate class to other packages, who have dependencies to it, for higher *COHM* and lower *COUM*. Obviously, the candidate class can only be a class that has inter-package dependencies. Moreover, for a class with less inter-package dependencies and more intra-package dependencies, moving it can disrupt the original software organization. Therefore, such classes should also be excluded from the set of candidate classes. In Algorithm 2, we adopt T_1 as the difference threshold of forwards dependencies on the target package and the source package, and T_2 is similar to T_1, but designed for backwards dependencies. In this paper, $T_1 = 2$, and $T_2 = 3$, are designed based on experience.

Next, when a candidate class move causes the value of *COHM* to increase and the value of *COUM* to decrease, a refactoring is performed. Unluckly, cohesion and coupling do not always change cooperatively in the opposite direction Therefore, there is a trade-off when *COHM* and *COUM* both increase or decrease together. Since software is carefully designed and implemented by professional programmers, refactoring is crucial,

Algorithm 2: Refactoring algorithm base on our metrics

Input: Adjacent matrix M; communities C; vertex number N.
Output: The original values of $COHM_{ogn}$, $COUM_{ogn}$; the new values of $COHM_{ref}$, $COUM_{ref}$.

Calculate the original values of $COHM_{ogn}$ and $COUM_{ogn}$ according to Algorithm 1.
while *there is a unvisited class* **do**
  Select a unvisited class A, and set $COHM_{max} = 0$, $COUM_{min} = 0$.
  Let P_s be the source package of A;
  for *traverse all packages* **do**
    Let P_t be the currently visiting package.
    Set $D_n =$ (as the number of dependencies of A depends on P_t).
    Set $D_o =$ (as the number of dependencies of A depends on P_s).
    set $D_{nb} =$ (the corresponding backwards dependencies on P_t).
    set $D_{ob} =$ (the corresponding backwards dependencies on P_s).
    if $(D_n - D_o > T_1 \parallel D_{nb} - D_{ob} > T_2 \parallel (D_n \geq 1 \&\& D_{nb} \geq 1))$ **then**
      Calculate $COHM_{ogn}$, $COUM_{ogn}$ for both P_s and P_t.
      Move A to P_t and update C.
      Re-calculate $COHM_{ref}$, $COUM_{ref}$ for both P_s and P_t.
      $Df_{COHM} = COHM_{ref} - COHM_{ogn}$.
      $Df_{COUM} = COUM_{ref} - COUM_{ogn}$.
      $RHU = |Df_{COHM}/Df_{COUM}|$.
      $RUH = |Df_{COUM}/Df_{COHM}|$.
      if $((COHM_{ref} > COHM_{max} \&\& COUM_{ref} < COUM_{min}) \|$
          $(Df_{COHM} > 0 \&\& Df_{COUM} > 0 \&\& RHU > 1.5) \|$
          $(Df_{COHM} < 0 \&\& Df_{COUM} < 0 \&\& RUH > 1.5))$ **then**
        if $(COHM_{ref} > COHM_{max} \&\& COUM_{ref} < COUM_{min})$ **then**
          $COHM_{max} = COHM_{ref}$, $COUM_{min} = COUM_{ref}$;
          mark p_{max} as P_t.
        end
        if $(Df_{COHM} > 0 \&\& Df_{COUM} > 0)$ **then**
          $COHM_{max} = COHM_{ref}$;
          mark p_{max} as P_t.
        end
        if $(Df_{COHM} < 0 \&\& Df_{COUM} < 0)$ **then**
          $COUM_{min} = COUM_{ref}$;
          mark p_{max} as P_t.
        end
      end
      Move A back to the P_s and update C.
    end
  end
  Set A as visited.
  if $COHM_{max} > 0 \&\& COUM_{min} < 0$ **then**
    Move A to package P_{max} and update C.
    Set all classes that depend on A as unvisited.
  end
end
Calculate $COHM_{ref}$ and $COUM_{ref}$ after refactoring, then return.

namely each refactoring should be of great value to the entire software system. Therefore, refactoring should occur when a good change is achieved to an extent that's not too low.

For the above reasons, if the values of *COHM* and *COUM* are changed in the same direction, we construct an evaluation model, that is: when the "good" (healthy to the software structure) changes are more than the "bad" changes at a threshold, the class is moved; otherwise the class stays without any refactoring. Here, this threshold is set at 50%. Empirically, performing a refactoring at this "good" extent is not wasteful. Finally, we repeat the above process and stopping moving until the entire software reaches the optimal configuration.

Let's analyse the complexity of Algorithm 2. Assume N be the number of classes, and N_p the number of packages. There is a nested two-layers loop in Algorithm 2. The outer loop is the while-loop at the 2nd line, while the inner loop is the for-loop at the 5th line. As to the inner for-loop, only the values of *COHM* and *COUM* of source package and target package are changed in the process of refactoring, therefore, we needn't consider other packages. According to Algorithm 1, the complexity of *COHM* and *COUM* for the source package is $O(N^2)$. Thus the complexity of for-loop at 5th line is $O(N^2 N_p)$. Therefore, the total complexity of Algorithm 2 is $O(N^3 N_p)$. Since our algorithm obeys the thought of greedy algorithm, it may encounter the problem of "local optimal". However, during the process of refactoring, the average values of cohesion and coupling for the whole software are always improved monotonously. So the correctness of Algorithm 2 is confirmed. Furthermore, the amount of classes is finite, so that the algorithm must be terminated after all classes are visited.

Experiments and analysis

Refactoring experiments and analysis

Our experiments are executed on a computer with configurations of i5-3230M, 8G DDR3, Windows 10. We select ten open-source software systems for experimental verification. These software systems have different functionalities and good maturity, and have also been widely applied in the industry. The basic information statistics are collected in Table 1.

In Table 2, we show the result of refactoring ten software systems. It can easily observed that for all software systems, after refactoring, the cohesion values are improved, while the coupling values are decreased. Figure 3 shows *COHM* comparison before and after refactoring. We can see that after refactoring, the value of *COHM* of each software is

Table 1 Basic information of multiple Java software systems

Name	PN	CN	EN	NP
Ant 1.9.9	58	998	5169	2
Cglib-nodep 3.2.6	9	202	914	1
Emma 2.0.5313	11	143	574	0
Hsqldb 2.4.0	21	553	4519	2
Jaxen 1.1.6	16	204	947	0
Jgroups 4.0.10	31	859	4454	2
Ormlite 5.0	11	176	938	0
PDF-Renderer 1.0.5	9	154	548	0
RabbitMQ Client 5.0.0	7	402	1661	1
Tomcat 9.0.1	42	663	1887	2

PN: Package number; CN: Class number; EN: Edge number; NP: Neglected packages

Table 2 Improvement of our metrics after refactoring

Name	RCN	WFR	Before/After/Diff(COHM)	Before/After/Diff(COUM)
Ant	671	49	0.223/0.249/ + 0.026	0.345/0.307/ − 0.038
Cglib-nodep	151	8	0.270/0.332/ + 0.062	0.163/0.159/ − 0.004
Emma	143	12	0.327/0.347/ + 0.019	0.213/0.165/ − 0.048
Hsqldb	311	31	0.288/0.319/ + 0.032	0.224/0.062/ − 0.162
Jaxen	204	17	0.315/0.424/ + 0.110	0.283/0.119/ − 0.163
Jgroups	460	95	0.230/0.295/ + 0.064	0.260/0.148/ − 0.112
Ormlite	176	35	0.262/0.333/ + 0.071	0.231/0.137/ − 0.093
PDF	154	14	0.357/0.389/ + 0.033	0.118/0.112/ − 0.006
RabbitMQ Client	189	4	0.492/0.510/ + 0.018	0.096/0.096/ − 0.001
Tomcat	522	18	0.414/0.448/ + 0.035	0.273/0.222/ − 0.051

RCN:Rest class number; *WFR*:Waiting for refactoring

significantly improved, up to 35%. The change of *COUM* value is shown in Fig. 4. Similarly, after refactoring, the value of *COUM* is significantly decreased, up to 72% improved. Therefore, through refactoring, the software structure is improved significantly to get closer to the goal of high cohesion and low coupling.

In our past work, Mi et al. proposed an effective package-level cohesion metric, which can effectively improve software structure (Mi et al. 2019). Pan et al. also proposed a community cohesion model for refactoring (Pan et al. 2014). We compare our refactoring algorithm to theirs respectively. Note that Mi and Pan only consider the cohesion metric. However, the coupling metric similarly plays an important role on software structure. So we also compare *COUM* at the stop of different refactoring algorithms. Table 3 records the result of time consumption of the refactoring algorithm and the value of *COUM*. It is remarked that the complexity of our refactoring algorithm is equal to Mi's and less than Pan's complexity $O(N^3 N_p^3)$. For easy observation, we show the comparison of *COUM* after refactoring with three algorithms in Fig. 5. For Mi's algorithm, the value of *COUM* is slightly higher than ours in most cases. As to Pan's algorithm, the *COUM* is much higher than ours, which means Pan's algorithm may cause high coupling between packages. And worse, their refactoring algorithm consumes more time, several hours for some software systems. Therefore, our refactoring algorithm can guide the software structure correctly and execute efficiently.

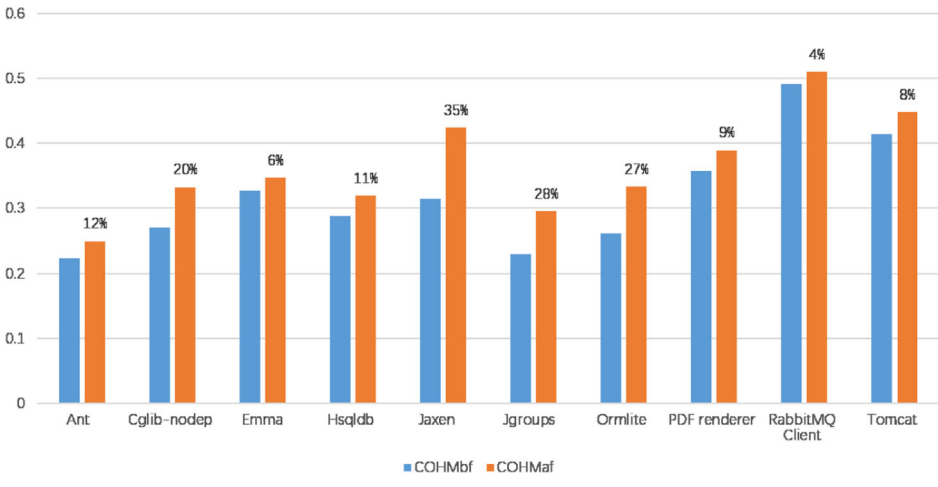

Fig. 3 The change of *COHM* after refactoring. *COHMbf* denotes the *COHM* value of the entire software before refactoring. *COHMaf* denotes the *COHM* value of the entire software after refactoring

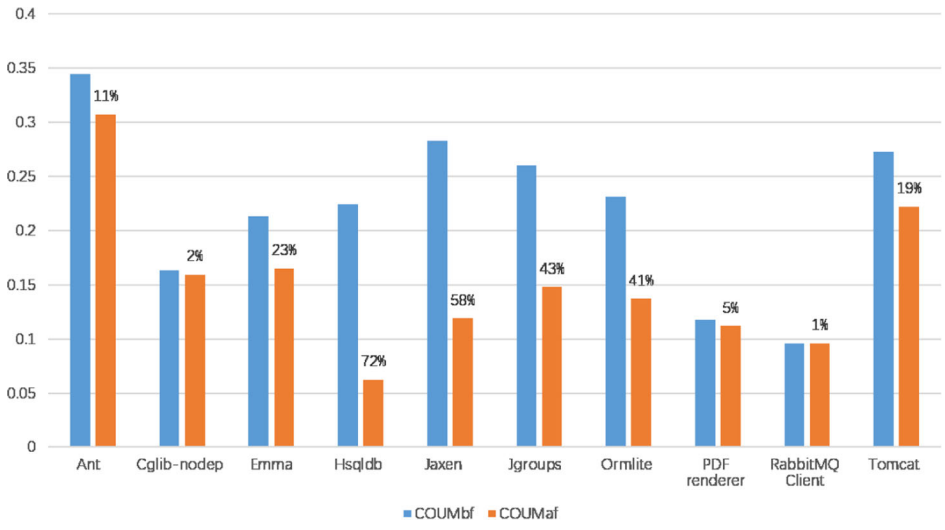

Fig. 4 The change of *COUM* after refactoring. *COUMbf* denotes the *COUM* value of the entire software before refactoring. *COUMaf* denotes the *COUM* value of the entire software after refactoring

Disturbing-recovering experiment and analysis

Several researches used to score the refactoring manually, which seems a little subjective. For reaching more objective comparison, we also design an experiment of disturbing and recovering to verify the correctness and efficiency of our metrics in guiding the software structure. Random disturbing for a package is that some classes in this package are randomly selected and placed into other packages. Recovering means that the disturbed classes can be recovered back to the original packages through refactoring algorithm.

Since software is an artifact developed carefully by programmers with professional skills, we deem the original structures of software systems as "*PERFECT*" structures. When we randomly disturb a package, the "*PERFECT*" structure is broken into chaos. Then we can use the refactoring algorithms to optimize the disturbed software systems. After refactoring, the more classes can be correctly recovered, the better the refactoring algorithm is. The precision rate P of recovering is calculated as

$$P = \frac{N_{Recovered}}{N_{disturbed}},$$

Table 3 Comparison of refactoring algorithms

name	*Ours*		*Mi's*		*Pan's*	
	TC	*COUM*	*TC*	*COUM*	*TC*	*COUM*
Ant	121	0.307	122	0.342	8353	0.389
Cglib-nodep	1	0.159	1	0.159	15	0.045
Emma	1	0.165	1	0.166	16	0.306
Hsqldb	44	0.062	43	0.054	3388	0.161
Jaxen	1	0.119	1	0.134	34	0.236
Jgroups	126	0.148	127	0.194	3254	0.302
Ormlite	1	0.137	1	0.139	20	0.235
PDF-Renderer	1	0.112	1	0.126	17	0.363
RabbitMQ Client	1	0.096	1	0.096	35	0.198
Tomcat	71	0.222	69	0.213	3811	0.280

TC: Time consumption (seconds)

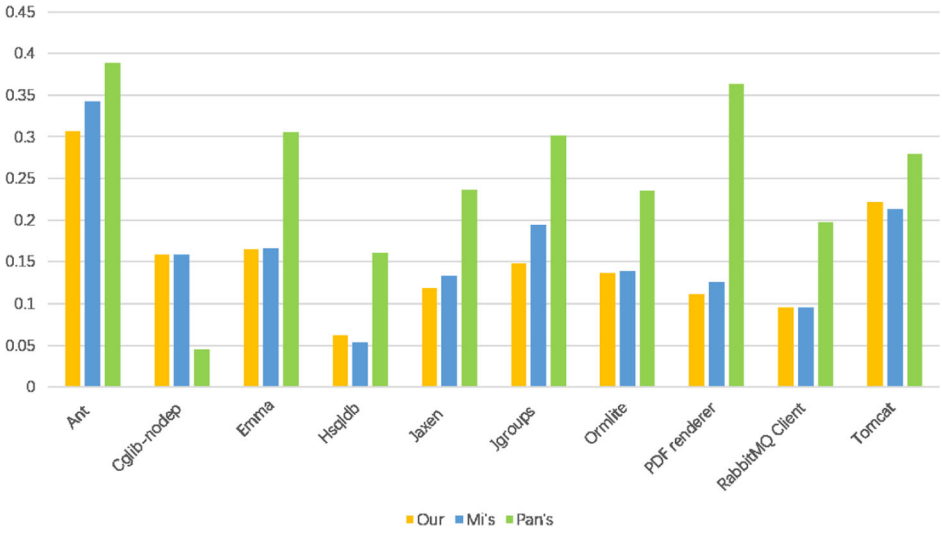

Fig. 5 The value of COUM

where $N_{Recovered}$ represents the number of disturbed classes recovered by the algorithm, and $N_{disturbed}$ denotes the total number of disturbed classes. We implement the disturbing-recovering experiment on ten Java open-source software systems. For each system, we repeatedly test 100 times and get their average. Then, we compare our refactoring algorithm with Mi's under the condition of different disturbing ratios.

The detail of disturbing-recovering experiment using our refactoring algorithm under 10% disturbing ratio is shown in Table 4. It can be found from the result that our refactoring algorithm can recover the disturbed classes very well and most classes can be placed back. This explains that our metrics can optimize the software structure effectively.

We also compare the performance between our algorithm and Mi's under the condition of different disturbing ratio 6%, 10%, 14% respectively. The comparison result is shown in Table 5. We can see that under different disturbing ratios, the average of our recovery percentages are higher than Mi's in most cases, except only one software *Ant*.

Table 4 Results of disturbing-recovering experiment (10%)

name	*NDC*	*NRC*	*RP(%)*	*LRP(%)*	*URP(%)*
Ant	35	27	76.2	78.4	73.7
Cglib-nodep	8	6	74.8	77.5	71.0
Emma	11	10	86.4	88.6	81.8
Hsqldb	17	14	81.1	83.8	78.4
Jaxen	14	10	70.3	72.7	66.7
Jgroups	32	24	75.0	78.8	71.9
Ormlite	13	10	75.8	80.4	70.8
PDF renderer	11	9	81.8	84.7	78.9
RabbitMQ Client	14	11	81.3	83.1	70.8
Tomcat	30	25	85.1	86.8	83.8

NDC: Number of disturbed classes; *NRC*: Number of recovered classes
RP:Average precision of recovering; *LRP*: Minimal precision of recovering; *URP*: Maximal precision of recovering

Table 5 Disturbing-recovering comparison under different disturbing ratios

name	*6%*		*10%*		*14%*	
	RPO(%)	*RPM(%)*	*RPO(%)*	*RPM(%)*	*RPO(%)*	*RPM(%)*
Ant	79.2	82.4	76.2	80.0	75.6	78.4
Cglib-nodep	82.3	76.2	74.8	74.0	67.4	66.9
Emma	92.8	92.0	86.4	88.7	86.9	86.4
Hsqldb	88.1	86.3	81.1	80.9	79.8	78.9
Jaxen	76.0	74.5	70.3	69.5	64.8	64.6
Jgroups	75.7	76.8	75.0	74.0	65.8	66.1
Ormlite	78.2	72.8	75.8	67.2	66.7	64.3
PDF-Renderer	87.4	84.6	81.8	80.4	71.8	73.1
RabbitMQ Client	87.9	85.6	81.3	81.2	74.7	72.9
Tomcat	91.7	86.0	85.1	84.9	82.3	81.4

RPO: Average Recovering percentage of our algorithm
RPM: Average Recovering percentage of Mi's algorithm

A more intuitive visualization is also demonstrated in Fig. 6. Under different disturbing ratios, our algorithm gets a more steady performance. Therefore, the disturbing-recovering experiment shows our metrics are good for software measurement, and the refactoring algorithm can be used to improve software quality for avoiding the risk structure deviation.

Conclusion

Software is a well-designed artifact implemented by programmers with professional skills. In the long maintenance phase, software faces the risk of code quality degradation and architecture deviation caused by continuous code revision. It is urgently necessary to create metrics, methods and tools to assist programmers in a macroscopic view. In this paper, we utilize the community methods in complex network and propose two metrics of package cohesion and coupling for software measurement. These two metrics are proved to satisfy the properties proposed by Briand(Briand et al. 1996). Then, based on the new

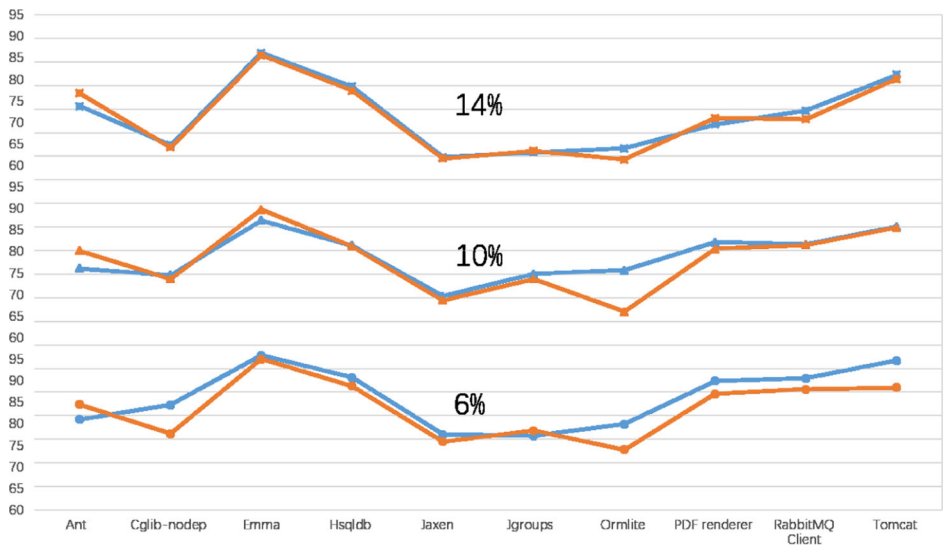

Fig. 6 The fluctuation of recovering precision[5]

metrics, we construct an evaluation model for package maturity, and propose a refactoring algorithm to make software structure better. Through several experiments on multiple open-source software systems, it is shown that our metrics are capable of improving package structure to fit the rule of high cohesion and low coupling, but also recovering the disturbed classes back to the correct place.

Abbreviations

CDG: the class dependency graph of the complex network; CN: the numbler of class in software; EN: the number of edge between classes; LRP: the minimal precision of recovering; NDC: the number of disturbed classes; NP: the number of neglected packages; NRC: the number of recovered classes; PN: the number of package in software; PRA: the number of other packages that depend on the package; PRE: the number of other packages that the package depends upon; RCN: the number of rest classes; RP: the average precision of recovering; RPM: the average recovering percentage of Mi's algorithm; RPO: the average recovering percentage of our algorithm; TC: time consumption of refactoring algorithms; URP: the maximal precision of recovering; WFR: the number of classes waiting for refactoring; WPAR: the number of incoming edges in a community; WPER: the number of outgoing edges in a community; WPR: the number of internal edges in a community

Acknowledgments

Not applicable.

Authors' contributions

YXZ proposed and theoretically demonstrated COUM metric and proposed a new refactoring algorithm. In addition, she completed the design and writing of the experiment. YRM proposed and theoretically demonstrated the COHM metric, and completed the parse of the software. YZ made a wide survey and wrote the related work. LC supervised the project and wrote the manuscript. All authors read and approved the final manuscript.

Funding

Not applicable.

Availability of data and materials

The ten software systems used in this paper, can be downloaded from their official websites, or requested from the corresponding author .

Competing interests

The authors declare that they have no competing interests.

Received: 13 March 2020 Accepted: 31 July 2020
Published online: 18 August 2020

References

Abdeen H, Ducasse S, Sahraoui H, Alloui I (2009) Automatic package coupling and cycle minimization. In: 2009 16th Working Conference on Reverse Engineering. IEEE, Lille. pp 103–112

Bieman JM, Kang B-K (1995) Cohesion and reuse in an object-oriented system. ACM SIGSOFT Softw Eng Notes 20(SI):259–262

Briand LC, Morasca S, Basili VR (1996) Property-based software engineering measurement. IEEE Trans Softw Eng 22(1):68–86

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Concas G, Marchesi M, Pinna S, Serra N (2007) Power-laws in a large object-oriented software system. IEEE Trans Softw Eng 33(10):687–708

Fortunato S (2010) Community detection in graphs. Phys Rep 486(3-5):75–174

Fowler M (1997) Refactoring: Improving the Design of Existing Code. In: Wells D, Williams L (eds). Extreme Programming and Agile Methods - XP/Agile Universe 2002. XP/Agile Universe 2002. Lecture Notes in Computer Science, vol 2418. Springer, Berlin

Girvan M, Newman ME (2002) Community structure in social and biological networks. Proc Natl Acad Sci 99(12):7821–7826

Gui G, Scott PD (2006) Coupling and cohesion measures for evaluation of component reusability. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. Association for Computing Machinery, New York. pp 18–21

Harrison R, Counsell SJ, Nithi RV (1998) An evaluation of the mood set of object-oriented software metrics. IEEE Trans Softw Eng 24(6):491–496

He K, Peng R, Liu J, He F, Liang P, Li B (2006) Design methodology of networked software evolution growth based on software patterns. J Syst Sci Complex 19(2):157–181

Kang D, Xu B, Lu J, Chu WC (2004) A complexity measure for ontology based on uml. In: 2004 10th IEEE International Workshop on Future Trends of Distributed Computing Systems. IEEE, Suzhou. pp 222–228

Lee Y (1995) Measuring the coupling and cohesion of an object-oriented program based on information flow. In: Proc. Int'l Conf. Software Quality, 1995, Austin

Li B, Wang H, Li Z-Y, He K-Q, Yu D-H (2006) Software complexity metrics based on complex networks. Dianzi Xuebao (Acta Electron Sin) 34(12):2371–2375

Li H, Zhao H, Cai W, Xu J-Q, Ai J (2013) A modular attachment mechanism for software network evolution. Phys A Stat Mech Appl 392(9):2025–2037
Martin RC (2002) Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, Upper Saddle River
Misic VB (2001) Cohesion is structural, coherence is functional: Different views, different measures. In: Proceedings Seventh International Software Metrics Symposium. IEEE. pp 135–144
Mi Y, Zhou Y, Chen L (2019) A new metric for package cohesion measurement based on complex network. In: International Conference on Complex Networks and Their Applications. Springer, Lisbon. pp 238–249
Myers CR (2003) Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Phys Rev E 68(4):046116
Newman ME (2006) Modularity and community structure in networks. Proc Natl Acad Sci 103(23):8577–8582
Pan W, Li B, Jiang B, Liu K (2014) Recode: software package refactoring via community detection in bipartite software networks. Adv Complex Syst 17(07n08):1450006
Pan W, Li B, Ma Y, Liu J (2011) Multi-granularity evolution analysis of software using complex network theory. J Syst Sci Complex 24(6):1068–1082
Pan W-F, Li B, Ma Y-T, Qin Y-Y, Zhou X-Y (2010) Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks. J Comput Sci Technol 25(6):1202–1213
Ping-ting S, Liang-yu C (2017) Complex network analysis in java application systems. J East China Normal Univ (Nat Sci) 2017(1):38
Potanin A, Noble J, Frean M, Biddle R (2005) Scale-free geometry in oo programs. Commun ACM 48(5):99–103
Sarkar S, Kak AC, Rama GM (2008) Metrics for measuring the quality of modularization of large-scale object-oriented software. IEEE Trans Softw Eng 34(5):700–720
Tom E, Aurum A, Vidgen R (2013) An exploration of technical debt. J Syst Softw 86(6):1498–1516
Valverde S, Solé RV (2005) Network motifs in computational graphs: A case study in software architecture. Phys Rev E 72(2):026107
Zheng X, Zeng D, Li H, Wang F (2008) Analyzing open-source software systems as complex networks. Phys A Stat Mech Appl 387(24):6190–6200

Publisher's Note