Pan WF, Li B, Ma YT *et al.* Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 25(6): 1202–1213 Nov. 2010. DOI 10.1007/s11390-010-1095-2

Measuring Structural Quality of Object-Oriented Softwares via Bug Propagation Analysis on Weighted Software Networks

Wei-Feng Pan[1] (潘伟丰), *Member*, *CCF*, Bing Li[1,2,3,*] (李　兵), *Senior Member*, *CCF*
Yu-Tao Ma[1,3] (马于涛), *Member*, *CCF*, *ACM*, Ye-Yi Qin[1] (覃叶宜), and Xiao-Yan Zhou[1] (周晓燕)

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

[2]*School of Computer, Wuhan University, Wuhan 430072, China*

[3]*Complex Networks Research Center, Wuhan University, Wuhan 430072, China*

E-mail: panweifeng1982@gmail.com; libing@sklse.org; yutaom@acm.org; qinyeyi2005@126.com; zhou0420@tom.com

Received July 15, 2009; revised May 17, 2010.

Abstract The quality of a software system is partially determined by its structure (topological structure), so the need to quantitatively analyze the quality of the structure has become eminent. In this paper a novel metric called *software quality of structure* (*SQoS*) is presented for quantitatively measuring the structural quality of object-oriented (OO) softwares via bug propagation analysis on weighted software networks (WSNs). First, the software systems are modeled as a WSN, weighted class dependency network (WCDN), in which classes are nodes and the interaction between every pair of classes if any is a directed edge with a weight indicating the probability that a bug in one class will propagate to the other. Then we analyze the bug propagation process in the WCDN together with the bug proneness of each class, and based on this, a metric (*SQoS*) to measure the structural quality of OO softwares as a whole is developed. The approach is evaluated in two case studies on open source Java programs using different software structures (one employs design patterns and the other does not) for the same OO software. The results of the case studies validate the effectiveness of the proposed metric. The approach is fully automated by a tool written in Java.

Keywords bug propagation, design pattern, object-oriented (OO) software, software network, structural quality

1 Introduction

Object-oriented (OO) has been the most widely used development paradigm since the early 1990's. And there are a large number of open source object-oriented (OSOO) software systems in free software ecosystems such as Sourceforge[1] and Freshmeat[2]. Many of them have equivalent or overlapping functionalities. The lack of objective evaluation criteria makes the choice of the best one from these candidate OSOO software systems difficult. Furthermore, as the OSOO software systems have considerable economic impact, and make great inroads into the mission-critical or life-critical real-world applications, many organizations would like to have object measures regarding the quality of the software products[3]. So there is a great need for some objective metrics to quantify the quality of OSOO software systems.

Software structure design explicates the structure of the software in terms of software components and interactions among them. Especially, in OO software systems, it describes methods, attributes, classes, packages, etc., and their interactions. With the increase of the complexity of software systems, the overall structure of the system is becoming more and more complicated, making the software structure become one of the most important factors that influences the quality of the final software products[4]. So the need to quantitatively analyze the quality of the structure has become eminent[5].

In recent years, researchers in the field of statistical physics and complex system used complex software networks (hereafter, software networks) to represent software systems by taking software components, such

Regular Paper

This work is supported by the National Basic Research 973 Program of China under Grant No. 2007CB310801, the National Natural Science Foundation of China under Grant Nos. 60873083, 60803025, 60703009 and 60703018, the Natural Science Foundation of Hubei Province under Grant No. 2008ABA379, the Natural Science Foundation of Hubei Province for Distinguished Young Scholars under Grant No. 2008CDB351, the Research Fund for the Doctoral Program of Higher Education of China under Grant Nos. 20070486065 and 20090141120022, and the Fundamental Research Funds for the Central Universities of China under Grant No. 6082005.

*Corresponding Author

as methods, classes and packages, as nodes and their interactions as edges[6]. It provides us a new way to study complex software systems. And the research interests are mainly involved in discovering the shared topological properties of software networks, the evolution mechanisms of software networks and the metrics for evaluating complexity of software networks. [7] gives a detailed review of the research in this new field.

But to the best of our knowledge, quantitative studies of software quality from the perspective of software networks (i.e., to quantify the structural quality of software networks) are very scarce, and the assumptions they are based on cannot meet the practice to a certain degree. Considering the defects of the existing methodologies, in this paper we propose modelling OO software systems at class level using a weighted software network (WSN), weighted class dependency network (WCDN), which is built on the details of the features (i.e., methods and attributes) and their interactions in the specific OO software system. And then, we present a new metric, software quality of structure (*SQoS*), to measure the structural quality of OO softwares by analyzing the bug propagation process in WCDN and the bug proneness of each class. We test our approach against two case studies on open source Java programs, each of which has two versions (one employs design patterns and the other does not). The results of the case studies validate the effectiveness of *SQoS*. The approach is fully automated by a tool written in Java.

The rest of this paper is organized as follows. Section 2 contains a brief summary of the related work. Section 3 describes our approach in detail. Section 4 presents the results of two case studies conducted on two open source case studies. In Section 5 a software tool that has been developed to automate the proposed approach is briefly described. The limitations and future work of our research are mentioned in Section 6. And we conclude the paper in Section 7.

2 Related Work

Let us brief the researches on software structural quality measurement first, and then detail some research work from the perspective of software networks.

In [8], Alan MacCormack *et al.* adopt design structure matrices (DSMs) to represent the software networks at the source file level, and introduce *change cost* to measure the average influence of components on the whole system. Basically, a software system should keep the *change cost* as low as possible, i.e., the influence of any performed change should be limited to a range as small as possible. However, it assumed that all files have the same probability that they can be changed, and that a change in one file will definitely propagate to other files that point to it directly and indirectly in software networks. This may not meet the practice.

In [9], Danmien Challet *et al.* propose a metric called *failure propagation basin* to study the bug propagation in function call graphs and package dependence graph. The *failure propagation basin* measures the potential influenced nodes caused by a faulty node (contains a bug) and it is defined as the number of nodes that point to the faulty node directly or indirectly in software networks. They calculated the *failure propagation basin* of each node, and studied the distribution curves of the size of failure propagation basins. They, however, neither take into account the significance of the attributes in bug propagation nor propose a metric to characterize the structural quality of the software systems as a whole.

In our preliminary work[10], we introduced an efficient statistical measure, called *average propagation ratio*, to characterize the structural quality of general complex software networks at the granularity level of class. And several representative real-world complex software networks were analyzed using average propagation ratio. However, average propagation ration also assumed that all classes have the same probability that they may be changed or may be faulty (contains a bug), and that a change or bug in one class will definitely propagate to other classes that point to it directly and indirectly in software networks, which may not meet the practice.

3 The Approach

In the previous researches as we talked in Section 2, people always make the following two assumptions: 1) the change or bug in one software component such as a file or a class will definitely propagate to other components that point to the changed or faulty node directly or indirectly in software networks; and 2) all software components, such as classes, and files, have the same probability that they may be changed or may be faulty. But these two assumptions may sometimes not meet the practice. The rationale is twofold.

1) In OO software systems, a class always contains many attributes and methods. We treat a class as a faulty one if it contains at least one bug. Similarly, the attributes and methods of another class depending on the faulty class do not all link to the faulty attributes or methods directly or indirectly in it.

See Fig.1, class X is composed of three methods $(b(), c(), d())$ and one attribute (a), and class Y is composed of two methods ($e()$ and $f()$). So if any attribute or method of class X is faulty, class X will be viewed as faulty. In previous work, people all think the bug in class X will definitely propagate to class Y, for the latter depends on the former. However it is not always the truth. For instance, if the bug exists in method $c()$, the bug will propagate to class Y, for $f()$ in class Y depends on $c()$ in X. However if the bug exists in other attributes or methods but $c()$, it will not propagate to class Y, for $f()$ and $e()$ in class Y do not directly or indirectly depend on the faulty attributes or methods in class X.

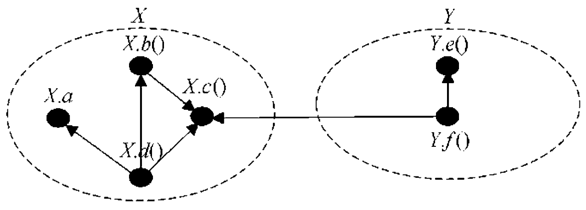


Fig.1. A simple example.

2) The probability that a bug arises in a class is not always the same. Many prior researches have revealed that this probability is closely related to many kinds of information such as complexity of requirement implementation[11] and source code complexity[12]. To make the estimation of software structural quality more precise, the property a bug arises probably will vary across classes should also be taken into consideration.

Compared with the prior researches, the main contributions of this paper are summarized in the following:

1) assume a class depending on a faulty class itself becomes faulty with probability p rather than 1;

2) introduce a new concept, *bug proneness index of classes* (*BPIC*), to represent the probability a bug arises in a class;

3) finally present a novel approach to measure the structural quality of OO softwares.

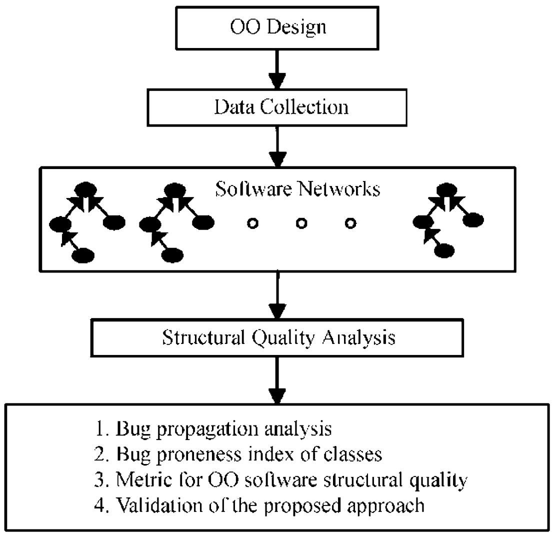

OO Design
Data Collection
Software Networks
Structural Quality Analysis
1. Bug propagation analysis
2. Bug proneness index of classes
3. Metric for OO software structural quality
4. Validation of the proposed approach

Fig.2. Workflow of the proposed approach.

In the following subsections, we will first detail the way to model OO software systems as weighted class dependency networks with the bug propagation probability p being the weight of the direct edge between two classes. And then we analyze the bug propagation process in such a software network, introduce the *bug proneness index of classes*, and finally propose a new metric to quantitatively measure the structural quality of OO softwares. Fig.2 gives a short overview of the workflow of the proposed approach.

3.1 OO Design

In this paper we mainly focus on the OO domains, and take the OSOO software systems as our research subjects. The rationale is threefold[13]:

1) OO has become the most widely used development paradigm since 1990's. And there are a lot of OSOO software systems on the Web which can be easily got for our research objectives.

2) OSOO software systems are developed under the OO paradigm. They have clear structures and the software components such as attributes, methods, classes, packages, and their dependencies are amenable to extraction and analysis.

3) many software design principles[14] can be easily used to improve the quality of OO software systems. And the research on OO designs can reveal the interrelationships between design principles and software quality.

3.2 Data Collection

Data collection refers to the process to extract software components such as attributes, methods, classes and their dependencies. We have developed a tool that can be used to analyze compiled Java codes (files with .class and .jar extension) to get needed data. In this paper, we only take into consideration two kinds of dependencies, method accessing attribute dependency and method call dependency, and the dependencies between classes are obtained according to these two kinds of dependencies.

3.3 Weighted Software Networks

After data collection, two kinds of weighted software networks can be built, weighted feature dependency network (WFDN) and weighted class dependency network (WCDN). And WFDN is mainly used to build WCDN. We use the term feature, a concept borrowed from Dependency Finder[15], to designate attributes and methods. The dependencies between two features as we talked in Subsection 3.2 are treated as the same dependency. In this subsection, we will first give the formal description of WFDN and WCDN.

Definition 1 (Weighted Feature Dependency Network (WFDN)). *In WFDN the nodes represent features (attributes or methods) of a specific OO software system. And each feature is represented by only one node. Directed edge between two nodes denotes one feature uses another feature, i.e., if feature A uses feature B, there will be an edge from the node denoting A to the node denoting B. And here we only consider the presence of dependency and neglect the multiplicity of dependencies such as A depends three times on B. And the weight of each directed edge denotes the probability that a bug in B will propagate to A. See Fig.*3 *for example. Therefore WFDN can be described as*:

$$WFDN = (N^f, E^f, \boldsymbol{M_P^f}), \tag{1}$$

where N^f is the set of all features of the specific OO software system; E^f is the set of directed edges denoting all relationships among features; $\boldsymbol{M_P^f}$ is a matrix storing the bug propagation probabilities among all pairs of nodes if they are linked by a directed edge in WFDN, i.e., if node j links to node i, the entry $\boldsymbol{M_P^f}(i,j)$ of $\boldsymbol{M_P^f}$ stores the probability that if node i is faulty (contains a bug), the bug will propagate to node j with probability $\boldsymbol{M_P^f}(i,j)$.

In WFDN we assume that the bug in one feature will propagate to features directly depending on it with probability 1. And for features that do not depend on it directly, the bug propagation probabilities will be 0.

```
public class X                    public class Y
{  private int a;                 {
   public void b() {c();}            public void e() {f();}
   public void c() {}                public void f() {
   public void d() {                   X x = new X();
     a++; b(); c();                    x.c();
   }                                 }
}                                 }
```

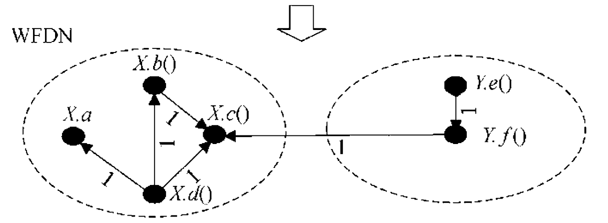


Fig.3. Illustration of WFDN.

Fig.3 shows a simple source code segment and its corresponding WFDN.

Definition 2 (Weighted Class Dependency Network (WCDN)). *In WCDN the nodes represent the classes of the specific OO software systems. And each class is represented by only one node in the whole WCDN. Directed edges between two nodes denote the interactions between the corresponding classes. And such interactions are obtained from the corresponding WFDN, i.e., if the features in class X use the features in class Y, there will be an edge from the node denoting class X to the node denoting class Y, and vice versa. Here we only consider the presence of dependency and neglect the multiplicity of dependencies such as class X depends three times on class Y. And the weight of each directed edge denotes the probability that a bug in class Y will propagate to class X. See Fig.*4 *for example. Therefore WCDN can be described as*:

$$WCDN = (N^c, E^c, \boldsymbol{M_P^c}). \tag{2}$$

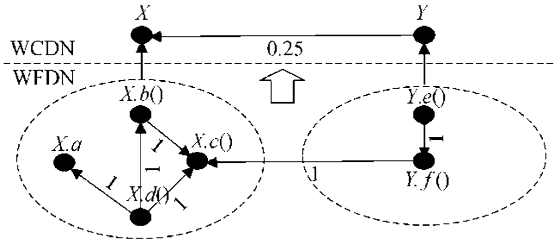


Fig.4. Illustration of WCDN.

The meanings of the notations of N^c, E^c and $\boldsymbol{M_P^c}$ are very similar to that we talked in WFDN. Only are they now applied to WCDN. So we will not go further into details about these notations here. Please refer to those in WFDN.

In the following, we will detail the way to calculate the probability matrix $\boldsymbol{M_P^c}$ clearly. But before that, some primary definitions should be given first.

Definition 3 (Extra-Class Method Reachability Set (ECMRS)). *Suppose there are two different classes, C^i and C^j ($C^i, C^j \in N^c$). And C^i depends on C^j directly (i.e., there is an edge from C^i to C^j in WCDN). Then we define the extra-class method reachability set of each method C_k^i in C^i, $ECMRS(C_k^i, C^j)$, as the set of features in C^j reachable from C_k^i. A feature w in C^j is reachable from C_k^i if in WFDN there is a directed path from C_k^i to w. And the path is only composed of nodes denoting features in C^i or C^j, and it travels each node only once.*

Fig.4 shows the WFDN in Fig.3 and its corresponding WCDN. We can find that class Y depends on class X. So $ECMRS(Y.e(), X) = \{X.c()\}$, and $ECMRS(Y.f(), X) = \{X.c()\}$ (for there are two paths, $Y.e() \rightarrow Y.f() \rightarrow X.c()$ and $Y.f() \rightarrow X.c()$).

Obviously, the *ECMRS* of a specific method in one class is the set of features in another class that will have direct and indirect impact on this method especially when one of them is faulty. In order to obtain the *ECMRS* of each method in every pair of classes which are directly linked, we propose a queue-based level order traversal algorithm which is shown in Algorithm 1, where F^i is the set of features in class i, $|*|$ denotes the counting of the elements in set $*$. Hereafter, if the notations are not explained explicitly, the meanings are similar to that proposed first.

Algorithm 1. Queue-Based Level Order Traversal Algorithm

Input: WFDN and the edge set E^c of the corresponding WCDN.

Output: $ECMRS(C_k^i, C^j)$, $i \neq j$, $i, j = 1, 2, \ldots, |N^c|$; $k = 1, 2, \ldots, |F^i|$.

```
1 Prepare a queue.
2 Select a directed edge from $E^c$, and delete it from $E^c$.
3 Find the classes $i$ and $j$ on its two sides, and suppose
  the direction is from $i$ to $j$.
4 Travel through the nodes in WFDN one by one, **if** (node
  $k \in F^i$) **then** add node $k$ to queue.
5 Get a node from the queue and store it in a set *setF*,
  then add other nodes who are dependent by this node
  and belong to $F^i$ or $F^j$ to queue.
6 **Go to** step 5 until node is NULL.
7 Let $ECMRS(C_k^i, C^j) = setF \cap F^j$, and $setF = \varnothing$.
8 **Go to** step 2 until edge is NULL.
```

Definition 4 (Inter-Class Ripple Basin (ICRB)). *The inter-class ripple basin of class i on class j, ICRB(i, j), describes the total set of features in class j that are directly or indirectly used by all methods in class i. So, in fact, it is the union of ECMRS of all methods in class i. Therefore ICRB(i, j) can be described as*:

$$ICRB(i,j) = \sum_{C_k^i \in F^i} \cup ECMRS(C_k^i, C^j). \tag{3}$$

Considering that the *ECMRS* of the methods in class i, all should have indirect relationships with the features in another class via the critical methods (those directly linked to features of another class), we only calculate the *ECMRS* of the critical nodes. See Fig.4 for example. $Y.f()$ is a critical method. $Y.e()$ is an ordinary method. And $Y.e()$ uses $Y.f()$. So we only need to calculate $ECMRS(Y.f(), X)$ to get $ECMRS(Y.e(), X)$. By the same token, $ICRB(i, j)$ is the union of *ECMRS* of all critical methods in class i. Therefore we can take a more simple formula to calculate $ICRB(i, j)$ which can be described as:

$$ICRB(i,j) = \sum_{C_k^i \in S_c} \cup ECMRS(C_k^i, C^j), \tag{4}$$

where S_c is the set of critical features in class i. We can find that $ICRB(i, j)$ in fact is the set of features in class j whose bug will propagate to the methods in class i, finally making class i faulty. In Fig.4, $ICRB(Y, X) = \{x.c()\}$.

So given class i and class j of an OO software system, the bug propagation probability from class i to class j is denoted by $\boldsymbol{M_p^c}(i, j)$ and defined in this paper as the following formula:

$$\boldsymbol{M_p^c}(j,i) = \frac{|ICRB(i,j)|}{|F_j|}. \tag{5}$$

In Fig.4 $\boldsymbol{M_p^c}(X, Y) = |\{X.c()\}|/|\{X.a, X.b(), X.c(), X.d()\}| = 1/4 = 0.25$.

3.4 Multi-Step Effect of Bug Propagation

From the definition of WCDN we can find that the weight of each edge describes the effect a faulty class will have on the nearest class which directly depends on it. It in fact describes the single-step effect of a faulty class. However, the multi-step effect should also need to be taken into consideration when we want to evaluate the effect of a faulty class on other classes.

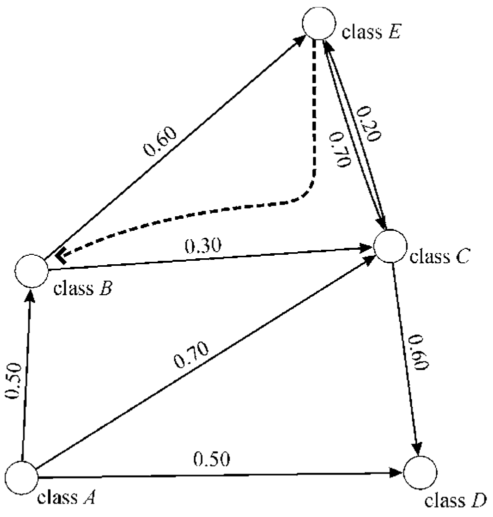


Fig.5. Illustration of a simple WCDN.

In Fig.5 the bug propagation probability that class B will be faulty if class E is faulty is not simply 0.6, for if class E is faulty, the bug will have 0.2 probability to propagate to class C, which in turn will propagate its bug to class B with probability 0.3 (along with the dashed arrow). So we can suppose that a bug in class E may have indirect effect on the class B via class C. Thus the bug propagation probability that class B will be faulty if class E is faulty is more than 0.6. Similarly, a bug in class C will have more than 0.7 probability to propagate to class A. And a bug in class E will have an indirect effect on class A via class B and class C though there is no direct dependency between the two classes. Obviously, it is the multi-step effect of a faulty class. So when evaluate the effect of a faulty class on other classes, we should take into consideration both the single-step and multi-step effect of a faulty class by measuring all direct and indirect effect among all pairs of classes.

After taking into consideration the multi-step effect of a faulty class, the bug propagation probability between every pair of classes is composed of two parts, single-step effect and multi-step effect. So we should update the bug propagation probability matrix $\boldsymbol{M_p^c}$. And the updating formula for the element of $\boldsymbol{M_p^c}$ can be written as:

$$\boldsymbol{M_p^c}(i,j)_{\text{updated}} = \boldsymbol{M_p^c}(i,j)_{\text{single-step}} + \boldsymbol{M_p^c}(i,j)_{\text{multi-step}}, \quad (6)$$

where $\boldsymbol{M_p^c}(i,j)_{\text{updated}}$ is the updated bug propagation probability between classes i and j. $\boldsymbol{M_p^c}(i,j)_{\text{single-step}}$ is the single-step bug propagation probability which is equal to the $\boldsymbol{M_p^c}(i,j)$ defined in WCDN. $\boldsymbol{M_p^c}(i,j)_{\text{multi-step}}$ is the multi-step bug propagation probability between classes i and j. And let $\boldsymbol{M_p^c}(i,j)^n_{\text{multi-step}}$ be the n-step (n is an integer and $n > 1$) bug propagation probability, i.e., $\boldsymbol{M_p^c}(i,j)^n_{\text{multi-step}}$ is the probability that the bug in class i propagates to class j by n steps. For simplicity, the n-step bug propagation probability from class i to class j can be calculated by the following formula:

$$\begin{aligned}\boldsymbol{M_p^c}(i,j)^n_{\text{multi-step}} =& \boldsymbol{M_p^c}(i,k_1)\boldsymbol{M_p^c}(k_{n-1},j) \\ &\cdot \prod_{l=1}^{n-2} \boldsymbol{M_p^c}(k_l,k_{l+1}),\end{aligned} \quad (7)$$

where k_i, $i = 1, 2, \ldots, n-1$ are the $n-1$ classes on the directed path from class j to class i with n steps.

So the multi-step bug propagation probability from class i to class j can be calculated by the following formula (where D is the diameter of the WCDN[16]):

$$\begin{aligned}\boldsymbol{M_p^c}(i,j)_{\text{multi-step}} &= \sum_{n=2}^{\infty} \boldsymbol{M_p^c}(i,j)^n_{\text{multi-step}} \\ &= \sum_{n=2}^{D} \boldsymbol{M_p^c}(i,j)^n_{\text{multi-step}}.\end{aligned} \quad (8)$$

Because $\boldsymbol{M_p^c}(i,j)^n_{\text{multi-step}}$ can be described as a recursive form $\boldsymbol{M_p^c}(i,j)^n_{\text{multi-step}} = \boldsymbol{M_p^c}(i, k_{n-1})^{n-1}_{\text{multi-step}}\boldsymbol{M_p^c}(k_{n-1},j)$, where k_{n-1} is the precursor class of class j. In order to calculate the updated bug propagation probability $\boldsymbol{M_p^c}(i,j)_{\text{updated}}$ between classes i and j, we propose a recursive algorithm described in Algorithm 2 to calculate the $\boldsymbol{M_p^c}(i,j)_{\text{multi-step}}$, where *bVisited*[$i$] is an array with type Bool denoting whether class i has been visited or not.

Algorithm 2. Calculation of the Multi-Step Bug Propagation Probability

```
Input: WCDN, $\boldsymbol{M^c_{p\,\text{multi-step}}}$ with all $\boldsymbol{M_p^c}(i,j)_{\text{multi-step}}$ = 0, class $i$ and class $j$ you want to calculate the multi-step bug propagation probability, and step $stp$ = 1.
Output: multi-step bug propagation probability $\boldsymbol{M_p^c}(i, j)_{\text{multi-step}}$.
1 Find class $j$.
2 If bVisited[$j$] == false, then bVisited[$j$] = true.
3 Find the first precursor of class $j$, class $k$ which is
    depended by class $j$ and bVisited[$k$] = false; then let
    bVisited[$k$] = true.
4 If class $k$ $\neq$ class $i$, then let $j$ = $k$,
    stp++, and call the algorithm itself; else store
    the path, calculate the stp-step bug propagation
    probability $\boldsymbol{M_p^c}(i,j)^{stp\text{-step}}_{\text{multi-step}}$ according to (7), let
    $\boldsymbol{M_p^c}(i,j)_{\text{multi-step}} += \boldsymbol{M_p^c}(i,j)^{stp\text{-step}}_{\text{multi-step}}$.
5 Find the next precursor class $k$ which is dependent on
    class $j$. If there is no more precursor, go to step 6, else
    bVisited[$k$] = false, and go to step 4.
6 Let bVisited[$j$] = false, stp = 1 and output
    $\boldsymbol{M_p^c}(i,j)_{\text{multi-step}}$.
```

The multi-step bug propagation probability matrix $\boldsymbol{M^c_p}_{\text{multi-step}}$ can be obtained by iteratively running Algorithm 2 for every pair of classes, and then we can get $\boldsymbol{M_p^c}(i,j)_{\text{updated}}$ according to (6). After we get $\boldsymbol{M^c_p}_{\text{updated}}$, we can measure the effect of every faulty class on other classes by the metric *class influence* (*CI*):

$$CI(i) = \sum_{j=1, j\neq i}^{|N^c|} \boldsymbol{M_p^c}(i,j)_{\text{updated}}, \quad (9)$$

where $CI(i)$ is the class influence of class i.

3.5 Bug Proneness Index of Classes

As we talked above, some kinds of information can be used to estimate the bug proneness of classes (i.e., the probability that a class contains a bug) such as complexity of requirement implementation and source code complexity. However, it is impossible for us to obtain the requirements of OSOO software systems. So in this paper, we only focus on using the source code complexity metrics to estimate the bug proneness of classes.

CK metrics suite, first presented by Chidamber and Kemerer in [17], is one of the most widely used metrics to evaluate complexities of OO softwares from inheritance (*DIT*, *NOC*), coupling between classes (*RFC*, *CBO*) and complexity within each class (*WMC*, *LCOM*). Many researches have sought to analyze the ability of CK metrics suite in estimating the bug proneness of classes[12,18-20]. Results show that most of the metrics in CK metrics suite are good indicators to predict bug prone classes, but there is a little difference across different researches. In [21], the author made a systematic comparison of the results of existing researches, and found that *SLOC*, *WMC*, *CBO*, and *RFC* are widely accepted as useful indictors for predicting the bug proneness of classes. We also use these four metrics to calculate the bug proneness index of classes proposed in this paper. We summarize them as follows. For details, please refer to [17, 22].

1) *Source Lines of Code* (*SLOC*). The *SLOC* is the number of code lines in a given class. Comments and empty lines are not counted.

2) *Weighted Methods per Class* (*WMC*). The *WMC* is the sum of complexities of every method in a given class. In this paper, we employ its simple form, where the complexity of each method is unity, i.e., *WMC* is the number of methods in a given class.

3) *Coupling Between Object* (*CBO*). The *CBO* is a count of the number of classes to which a given class is coupled.

4) *Response For a Class* (*RFC*). The *RFC* is a count of the number of methods that can potentially be executed in response to a message being received by an object of that class.

Based on what we have talked above, we propose a new metric, *bug proneness index of classes* (*BPIC*), to quantify the bug proneness of classes, and it can be computed according to (10).

$$\begin{aligned}BPIC_i = {}&\alpha\frac{SLOC_i}{SLOC_{\text{sum}}} + \beta\frac{WMC_i}{WMC_{\text{sum}}} + \\ &\gamma\frac{CBO_i}{CBO_{\text{sum}}} + \varphi\frac{RFC_i}{RFC_{\text{sum}}},\end{aligned} \tag{10}$$

where $BPIC_i$ is the *BPIC* of the class i. $SLOC_i$, WMC_i, CBO_i, and RFC_i are the *SLOC*, *WMC*, *CBO*, *RFC* of the class i, respectively. $SLOC_{\text{sum}}$, WMC_{sum}, CBO_{sum}, and RFC_{sum} are the sum of *SLOC*, *WMC*, *CBO*, *RFC* of all classes, respectively. α, β, γ and φ are the weights for the corresponding metrics and meet $\alpha+\beta+\gamma+\varphi = 1$. They are determined according to the effectiveness of that metric in bug proneness of classes estimation, i.e., the better the metric in bug proneness estimation, the larger value the metric can be set to. In this paper, we give each metric equal weight, i.e., $\alpha = \beta = \gamma = \varphi = 1/4$.

3.6 Metric for Structural Quality of OO Softwares

Then, based on *CI* and *BPIC*, a novel metric for measuring the structural quality of OO softwares, named *software quality of structure* (*SQoS*), can be produced, which can be computed according to (11).

$$SQoS = \frac{\sum_{i=1}^{|N^c|} CI(i) \times BPIC_i}{N^2 - N}, \tag{11}$$

where $|N^c|$ is the number of classes in the OO software systems. Obviously *SQoS* is a scalar whose value is not smaller than 0. *SQoS* will be 0 only when $\boldsymbol{M_p^c}(i,i)_{\text{updated}}$ are 1s and $\boldsymbol{M_p^c}(i,j)_{\text{updated}}$ $(i \neq j)$ are 0s. It is an ideal situation that the bug in any class will not propagate to other classes. So *SQoS* can be used to describe how far it is away from the best situation. So a lower *SQoS* indicates a software structure with a better quality, and bugs cannot easily propagate between its classes.

4 Experiment and Data Analysis

Design patterns are generally defined as descriptions of communicating classes that form a common solution to a type of design problem. They are widely accepted as a proven way to improve software quality[23-24]. Such an improvement in software quality should be qualitatively captured by the applied metrics to evaluate the software quality.

In order to investigate the applicability of the methodology proposed in this paper, two open source Java applications have been examined. Both applications have two versions with different software structures: one employs design patterns and the other does not. These two Java applications are selected for analysis because they satisfy the following criteria: 1) access to the full source code is possible since they are both open source applications; 2) they are written in Java which can be supported by our analysis tool; and 3) the two versions for each Java application have the same functionality, and the only difference is the software structure. So it can be used to evaluate the improvement made in software structure.

4.1 Case Study 1

The bridge design pattern is a design pattern used in software engineering which is meant to decouple an abstraction from its implementation so that the two can vary independently[25]. Our first case tests the proposed approach against the bridge design pattern. It is a simple Java application. There are two versions: one version does not employ bridge design pattern, while the other is implemented using bridge design pattern. The source code can be downloaded from [26].

To analyze their structural quality, we model them by WFDN and WCDN, using our own developed analysis tool SSQAT (that will be detailed in Section

5). Fig.A1 (see Appendix) shows the WFDN and the corresponding WCDN of case study 1. Table 1 and Table 2 show the values of metrics we talked above, such as *SQoS*, *CI* and *SLOC*, for software structure before and after using the bridge design pattern, respectively. They are all computed automatically by SSQAT. As for the calculation of CK metrics, we also can use some open source tools such as CKJM[27] and Eclipse metrics plug-in[28].

Based on our assumption that a lower value of *SQoS* implies a better software structure, the OO Java application after employing the bridge design pattern is better than that before, with their *SQoS* being 0.020963 and 0.037174, respectively.

Table 1. Case Study 1 before Employing the Bridge Design Pattern ($SQoS = 0.037174$)

Class Name	*CI*	*SLOC*	*WMC*	*CBO*	*RFC*	*BPIC*
BridgeDisc	0.000000	33	2	4	13	0.251084754
StackArray	3.427080	14	6	3	6	0.186029800
StackFIFO	0.666667	13	4	2	8	0.155260231
StackHanoi	0.750000	10	5	2	7	0.152205575
StackList	0.833333	17	6	2	12	0.208239725
Node	1.833330	5	1	1	1	0.047179915

Table 2. Case Study 1 after Employing the Bridge Design Pattern ($SQoS = 0.020963$)

Class Name	*CI*	*SLOC*	*WMC*	*CBO*	*RFC*	*BPIC*
BridgeDisc	0.000000	23	2	3	9	0.133596
Stack	1.787500	13	7	6	14	0.195333
StackFIFO	0.500000	16	5	4	13	0.162834
StackHanoi	0.600000	13	6	2	9	0.125183
Node	1.783330	5	1	1	1	0.033248
StackArray	1.200000	14	6	1	6	0.105287
StackList	0.783335	17	6	3	12	0.156762
StackImpl	4.100000	6	5	2	5	0.087757

4.2 Case Study 2

The decorator design pattern allows new or additional behavior to be added to an existing class dynamically[23]. The second case tests the proposed approach against the decorator design pattern. It is also a simple Java application, with two versions: one version does not employ the decorator design pattern, and the other is implemented using the decorator design pattern. The source code can be downloaded from [26].

Fig.A2 (see Appendix) shows the WFDN and WCDN of case study 2. Tables 3 and 4 show the values of metrics such as *SQoS*, *CI* and *SLOC*, for software structure before and after using the decorator design pattern, respectively.

Table 3. Case Study 2 before Employing the Decorator Design Pattern ($SQoS = 0.055269$)

Class Name	*CI*	*SLOC*	*WMC*	*CBO*	*RFC*	*BPIC*
Decorator Before	0.000000	8	2	4	6	0.159238
Decorator Before$A	10.694400	3	2	4	2	0.102449
Decorator Before$ AwithX	3.750000	4	3	4	5	0.143378
Decorator Before$ AwithXY	0.666667	8	2	3	6	0.148821
Decorator Before$ AwithXYZ	0.750000	11	2	4	8	0.190609
Decorator Before$ AwithY	2.277780	4	3	3	5	0.132961
Decorator Before$ AwithZ	1.166670	4	3	2	5	0.122544

Table 4. Case Study 2 after Employing the Decorator Design Pattern ($SQoS = 0.028587$)

Class Name	*CI*	*SLOC*	*WMC*	*CBO*	*RFC*	*BPIC*
DecoratorAfter	0.000000	10	2	5	7	0.230762
DecoratorAfter$A	1.000000	3	2	1	2	0.083266
DecoratorAfter$I	6.000000	2	1	2	1	0.065845
DecoratorAfter$X	0.333333	6	3	2	5	0.159836
DecoratorAfter$Y	0.333333	6	3	2	5	0.159836
DecoratorAfter$Z	0.333333	6	3	2	5	0.159836
DecoratorAfter$D	4.000000	4	2	4	3	0.140618

The *SQoS* for software structure before and after using the decorator design pattern are 0.055269 and 0.028587, respectively. So the OO Java application employing the decorator design pattern is better in structural quality than the one without the pattern.

5 Implementation

We have developed a Java program called Software Structural Quality Analysis Tool (SSQAT), which mainly consists of four parts: 1) a bytecode parser, 2) a NET generator and parser, 3) a Chidamber and Kemerer (CK) Java metrics calculator, and 4) an *SQoS* calculator.

The bytecode parser which uses that in Dependency Finder as its core component can parse the complied Java code (files with .class and .jar extension) to reveal the static structure.

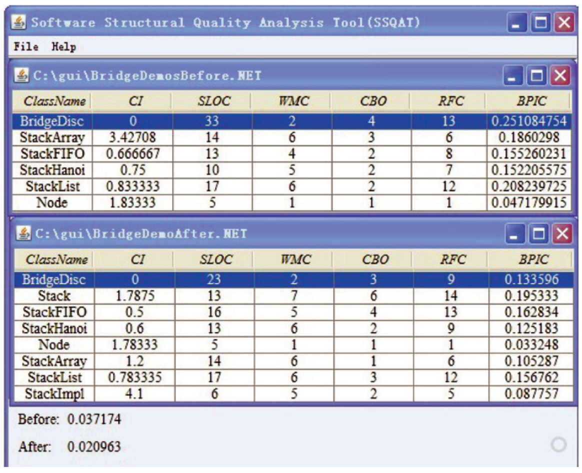
The NET generator, after the complied Java code has been parsed, produces two NET files, denoting

WFDN and WCDN, respectively. The NET files contain the information about the names of the software components (features or classes), their dependencies and the bug propagation probability of each directed edge. It has the same format as that used in Pajek[29]. So you can also use Pajek to give an illustration of the software networks. For the limitation of space, here we will not go further into details about the format of the NET file.

The Chidamber and Kemerer (CK) Java metrics calculator calculates Chidamber and Kemerer OO metrics by processing the bytecode of compiled Java files.

The SQoS calculator applies the aforementioned approach to calculate $\boldsymbol{M}_{\boldsymbol{p}}^{c}(i, j)$ for each pair of classes, *CI* for each class, and *SQoS* for the OO software system as a whole.

A sample screenshot of SSQAT with the concerned *CI*, CK metrics, and *SQoS* for case study 1 is shown in Fig.6.



C:\gui\BridgeDemosBefore.NET

ClassName	CI	SLOC	WMC	CBO	RFC	BPIC
BridgeDisc	0	33	2	4	13	0.251084754
StackArray	3.42708	14	6	3	6	0.1860298
StackFIFO	0.666667	13	4	2	8	0.155260231
StackHanoi	0.75	10	5	2	7	0.152205575
StackList	0.833333	17	6	2	12	0.208239725
Node	1.83333	5	1	1	1	0.047179915

C:\gui\BridgeDemoAfter.NET

ClassName	CI	SLOC	WMC	CBO	RFC	BPIC
BridgeDisc	0	23	2	3	9	0.133596
Stack	1.7875	13	7	6	14	0.195333
StackFIFO	0.5	16	5	4	13	0.162834
StackHanoi	0.6	13	6	2	9	0.125183
Node	1.78333	5	1	1	1	0.033248
StackArray	1.2	14	6	1	6	0.105287
StackList	0.783335	17	6	3	12	0.156762
StackImpl	4.1	6	5	2	5	0.087757

Before: 0.037174

After: 0.020963

Fig.6. Sample screenshot for SSQAT.

6 Limitations and Future Work

Although our approach shows some feasibilities in measuring the structural quality of the sample Java applications, the broad validity of our approach demands further demonstration. Moreover, when constructing WFDN, we suppose that the bug in one feature will propagate to all possible target features pointing to it directly and indirectly definitely, i.e., our analysis in WFDN can be described as a worst-case analysis. But to the best of our knowledge, there have been no prior researches on this topic. So in this paper, we have not addressed this problem.

Thus, the future work includes:

1) validating the approach using more other open source software systems written in Java and other programming languages (e.g., C++, C#);

2) presenting a more realistic approach which takes into consideration the non-trivial probability (not simply zero or one) in WFDN;

3) using the proposed approach for test case prioritization for regression testing.

7 Conclusion

Acknowledging the importance of software structure (topological structure) on the software quality, in this paper we use the weighted class dependency network (WCDN) to model the topological structure of OO software system, examined the bug propagation process in WCDN together with the bug proneness of classes, and finally proposed a metric *SQoS*. The goal is to use *SQoS* to measure the structural quality of OO software systems. The rationale behind this approach is that in a high quality software system, bugs arising in classes should be limited to a range as small as possible, i.e., *SQoS* should be kept as small as possible.

Case studies have shown the effectiveness of *SQoS* in software structural quality measurement. The proposed approach improves the accuracy of existing methodologies, for *SQoS* is produced considering both the software structure information at feature (method and attribute) and class levels as well as the complexity characteristics of the software source code.

The proposed approach has been automated by a tool written in Java and can be applied to measure the *SQoS* of any OO software system written in Java.

Acknowledgments We thank Jin-Hu Lv of Complex Networks Research Center at Wuhan University for invaluable advice, and thank anonymous reviewers for their helpful comments. We also thank Jing Liu, Xi Pu, Shan Li, and Ting-Ting Hou of State Key Laboratory of Software Engineering at Wuhan University for discussions. The data for the case studies is available for download from [30].

References

[1] Sourceforge. http://sourceforge.net, May 15, 2009.

[2] Freshmeat. http://freshmeat.net, May 15, 2009.

[3] Spinellis D, Gousios G, Karakoidas V, Louridas P, Adams P J, Samoladas I, Stamelos I. Evaluating the quality of open source software. *Electronic Notes in Theoretical Computer Science*, 2009, 233: 5-28.

[4] Fenton N E, Pfleeger S L. Software Metrics: A Rigorous and Practical Approach, 2nd Edition, London: International Thomson Computer Press, 1996.

[5] Abdelmoez W, Shereshevsky M, Gunnalan R, Ammar H H, Yu B, Bogazzi S, Korkmaz M, Mili A. Quantifying software architectures: An analysis of change propagation probabilities. In *Proc. the 3rd ACS/IEEE International Conference on Computer Systems and Applications*, Cairo, Egypt, Jan. 3-6, 2005, pp.687-694.

[6] Myers C R. Software systems as complex networks: Structure function, and evolvability of software collaboration graphs. *Physical Review E*, 2003, 68(4): 046116.

[7] Li B, Ma Y, Liu J, Ding Q. Advances in the studies on complex networks of software systems. *Advances in Mechanics*, 2008, 38(6): 805-814. (In Chinese)

[8] MacCormack A, Rusnak J, Bald Win C Y. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 2006, 52(7): 1015-1030.

[9] Challet D, Lombardoni A. Bug propagation and debugging in asymmetric software structures. *Physical Review E*, 2004, 70(4): 1015-1030.

[10] Liu J, Lu J, He K, Li B, TSE C K. Characterizing the structural quality of general complex software networks. *International Journal of Bifurcation and Chaos*, 2008, 18(4): 605-613.

[11] Srikanth H, Williams L, Osborne J. System test case prioritization of new and regression test cases. In *Proc. International Symposium on Empirical Software Engineering* (*ISESE 2005*), Queensland, Australia, Nov. 17-18, 2005, pp.64-73.

[12] Subramanyan R, Krishnan M S. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 2003, 29(10): 297-310.

[13] Pan W, Li B, Ma Y, Liu J, Qin Y. Class structure refactoring of object-oriented softwares using community detection in dependency networks. *Frontiers of Computer Science in China*, 2009, 3(3): 396-404.

[14] Martin R. Design principles and design patterns. http://www.objectmentor.com, May 20, 2009.

[15] Dependency finder. http://sourceforge.net/projects/depfind/files/, Jun. 3, 2009.

[16] Valverde S, Sole R V. Hierarchical small worlds in software architecture. Working Paper, SFI/03-07-044, SanteFe Insitute, 2003.

[17] Chidamber S R, Kemerer C F. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 1994, 20(6): 476-493.

[18] Basili V R, Briand L C, Melo W L. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 1996, 22(10): 751-761.

[19] Emam K EI, Benlarbi S, Goel N. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 2001, 27(6): 630-650.

[20] Gyimóthy T, Ferenc R, Siket I. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 2003, 31(10): 897-910.

[21] Xu J, Ho D, Capretz L F. An empirical validation of object-oriented design metrics for fault prediction. *Journal of Computer Science*, 2008, 4(7): 571-577.

[22] Basili V R, Perricone B. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 1984, 27(1): 42-52.

[23] Prechelt L, Unger B, Philippsen M, Tichy W F. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*, 2002, 28(6): 595-606.

[24] Tsantalis N, Chatzigeorgiou E, Stephanides G, Halkidis S T. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 2006, 32(11): 896-909.

[25] Gamma E, Helm R, Johnson R, Vlissides J M. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley Professional, Indiana, 1998.

[26] Open Source Java Applications. http://www.vincehuston.org/dp/, Jun. 3, 2009.

[27] CKJM. http://www.spinellis.gr/sw/ckjm/, Jun. 3, 2009.

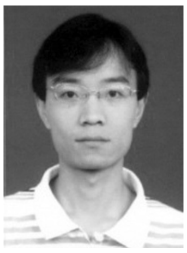
[28] Eclipse metrics plug-in. http://metrics.sourceforge.net/, Jun. 3, 2009.

[29] Pajek. http://pajek.imfm.si/doku.php, Jun. 3, 2009.

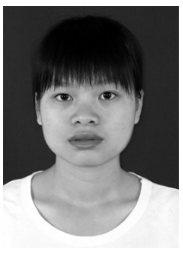
[30] Data for the case studies. http://blog.sina.com.cn/breezepan, Jun. 3, 2009.

Wei-Feng Pan is now a Ph.D. candidate of State Key Laboratory of Software Engineering (SKLSE) at Wuhan University. He is a CCF member. His current research interests include software metrics, software evolution, and interdisciplinary research between software engineering and complex networks.

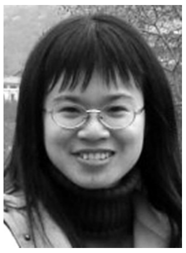
Bing Li, CCF senior member, is now a professor and Ph.D. supervisor of SKLSE at Wuhan University. He worked as a postdoctoral researcher in SKLSE at Wuhan University from 2003 to 2005. And he received his Ph.D., M.S. and B.A. degrees from Huazhong University of Science and Technology (HUST) in 2003, 1997 and 1990 respectively, all in computer science. His main research interests include requirements engineering, complex network, and semantic Web service.



Yu-Tao Ma is now a lecturer of SKLSE at Wuhan University as well as a post-doctor researcher of Institute of Electronic System Engineering. He is both CCF and ACM member. He received the Ph.D. degree from Wuhan University in 2007. His current research interests include software metrics, software evolution and the interdisciplinary research between software engineering and complex networks.



Ye-Yi Qin is an M.S. candidate of SKLSE at Wuhan University. Her research interests include software engineering and complex networks.



Xiao-Yan Zhou is an M.S. candidate of SKLSE at Wuhan University. Her research interests include software engineering and complex networks.

Appendix

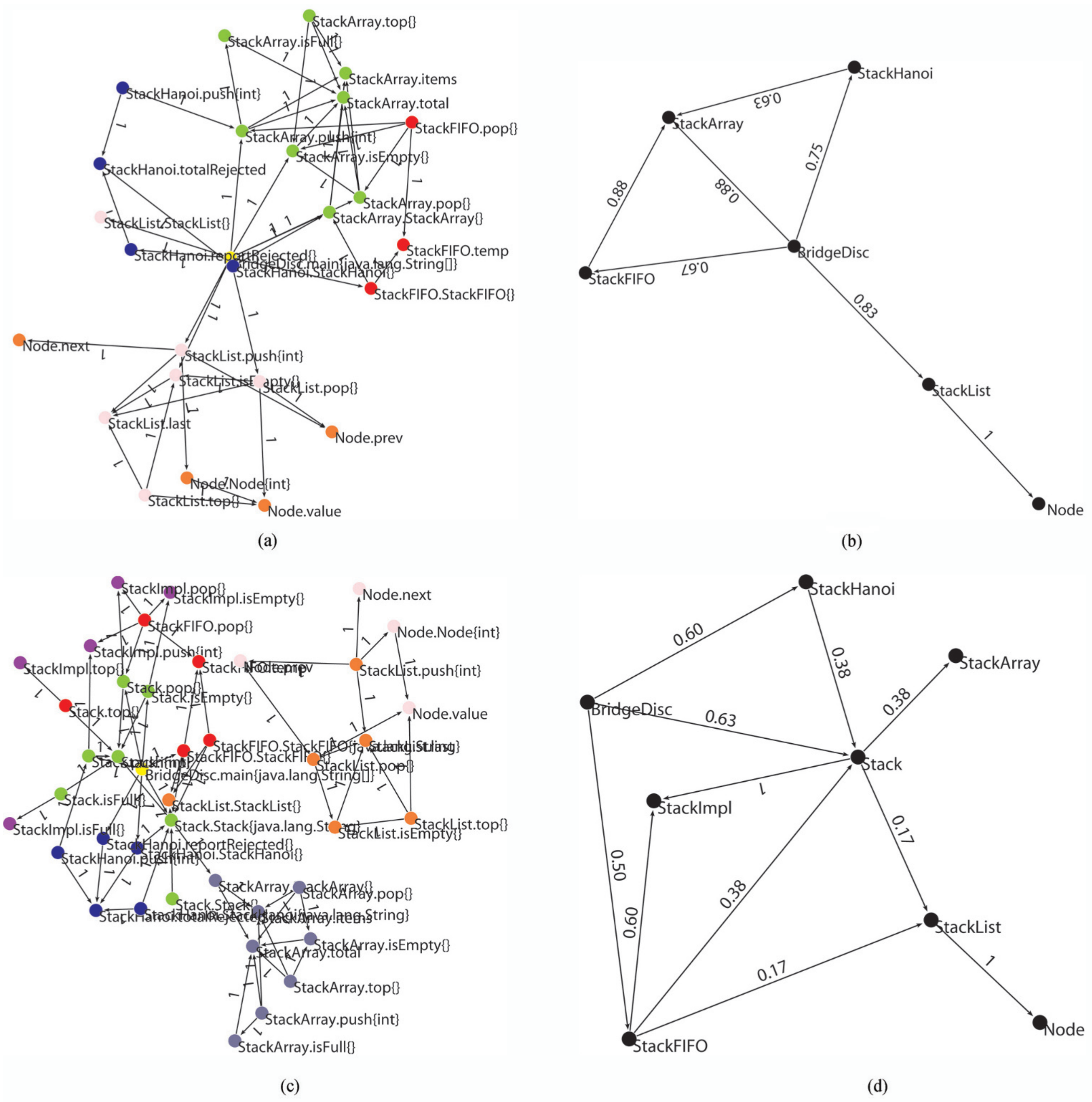


Fig.A1. The WFDN and WCDN of case study 1 before and after using the bridge design pattern. In (a) and (c) the nodes (features) with the same color belong to the same class. The notes beside the nodes are their names, and that beside the edges are the bug propagation probabilities. (a) WFDN before. (b) WCDN before. (c) WFDN after. (d) WCDN after.

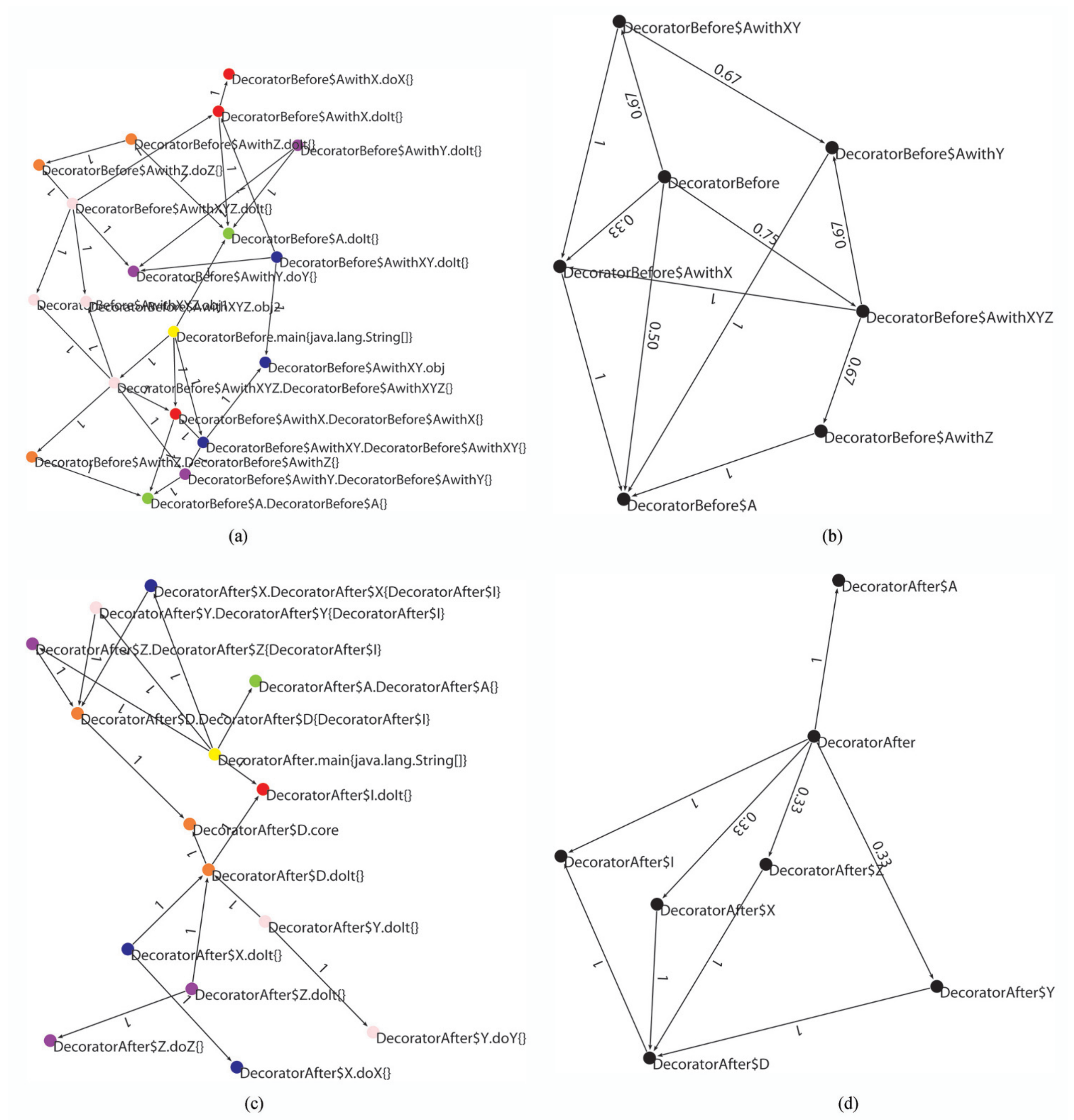

DecoratorBefore$AwithX.doX{}
DecoratorBefore$AwithX.doIt{}
DecoratorBefore$AwithZ.doIt{}
DecoratorBefore$AwithY.doIt{}
DecoratorBefore$AwithZ.doZ{}
DecoratorBefore$AwithXYZ.doIt{}
DecoratorBefore$A.doIt{}
DecoratorBefore$AwithXY.doIt{}
DecoratorBefore$AwithY.doY{}
DecoratorBefore$AwithXYZ.obj
DecoratorBefore$AwithXYZ.obj2
DecoratorBefore.main{java.lang.String[]}
DecoratorBefore$AwithXY.obj
DecoratorBefore$AwithXYZ.DecoratorBefore$AwithXYZ{}
DecoratorBefore$AwithX.DecoratorBefore$AwithX{}
DecoratorBefore$AwithXY.DecoratorBefore$AwithXY{}
DecoratorBefore$AwithZ.DecoratorBefore$AwithZ{}
DecoratorBefore$AwithY.DecoratorBefore$AwithY{}
DecoratorBefore$A.DecoratorBefore$A{}
(a)
DecoratorBefore$AwithXY
0.67
0.67
1
DecoratorBefore$AwithY
DecoratorBefore
0.33
0.75
0.67
DecoratorBefore$AwithX
1
DecoratorBefore$AwithXYZ
0.50
1
1
0.67
DecoratorBefore$AwithZ
1
DecoratorBefore$A
(b)
DecoratorAfter$X.DecoratorAfter$X{DecoratorAfter$I}
DecoratorAfter$Y.DecoratorAfter$Y{DecoratorAfter$I}
DecoratorAfter$Z.DecoratorAfter$Z{DecoratorAfter$I}
DecoratorAfter$A.DecoratorAfter$A{}
DecoratorAfter$D.DecoratorAfter$D{DecoratorAfter$I}
DecoratorAfter.main{java.lang.String[]}
DecoratorAfter$I.doIt{}
DecoratorAfter$D.core
DecoratorAfter$D.doIt{}
DecoratorAfter$Y.doIt{}
DecoratorAfter$X.doIt{}
DecoratorAfter$Z.doIt{}
DecoratorAfter$Y.doY{}
DecoratorAfter$Z.doZ{}
DecoratorAfter$X.doX{}
(c)
DecoratorAfter$A
1
DecoratorAfter
1
0.33
0.33
0.33
DecoratorAfter$I
DecoratorAfter$Z
DecoratorAfter$X
1
1
1
DecoratorAfter$Y
1
DecoratorAfter$D
(d)

Fig.A2. The WFDN and WCDN of case study 2 before and after using the decorator design pattern. The notes and colors denote the similar meanings to that in Fig.A1. (a) WFDN before. (b) WCDN before. (c) WFDN after. (d) WCDN after.