Cluster Comput (2019) 22:S2589–S2598
https://doi.org/10.1007/s10586-017-1353-y

Measuring software stability based on complex networks in software

Weifeng Pan[1] · **Chunlai Chai**[1]

Received: 20 September 2017 / Revised: 25 October 2017 / Accepted: 6 November 2017 / Published online: 30 November 2017

Abstract Software maintenance is regarded as an activity of high cost. Developing meaningful metrics to assess the quality characteristics of software has become one of the most effective ways to reduce the cost. In this paper, we propose metrics to quantify the software stability from a complex network perspective. First, the topological structure of software at the class level is represented by a Class Coupling Network (CCN). Second, based on the CCN, we further propose a Node Influence Network (NIN) which considers both the directed and indirected (transitive) coupling strength between classes. Finally, based on NIN, we propose a metric to quantify the class stability and further propose a metric to quantify the stability of software as a whole. The proposed metrics are validated theoretically using widely accepted Weyuker's criteria and empirically using Java programs. The theoretical evaluation shows the proposed metrics satisfy most of Weyuker's properties, and the empirical evaluation shows the effectiveness of our proposed metrics as indicators of the external software qualities such as scalability and change proneness.

Keywords Software stability · Complex networks · Software metrics · Object-oriented software · Software maintenance

✉ Chunlai Chai
ccl@mail.zjgsu.edu.cn

Weifeng Pan
wfpan@mail.zjgsu.edu.cn

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

1 Introduction

Software maintenance has been widely regarded as an activity of high cost, with typical estimates ranging from 60 to 80% of the total software cost [1]. In software maintenance process, there are two key activities, performing changes and change impact analysis, which account for more than 40% of the total software maintenance cost [2]. It has become an urgent as well as tough problem to control the change-related cost and further reduce the total maintenance cost. There basically exist two ways to control the cost [3]. The first way is to provide effective techniques or tools to ease the maintenance tasks. The second way is to develop or utilize meaningful metrics to assess the quality characteristics that may affect the cost. In this paper, we follow the second way to reduce the maintenance cost.

During the life cycle, software should be changed to correct faults, improve performance, or adapt to a changed environment [4]. Evolution has become an intrinsic property of software. However, the effect of a change may not be local to the change, and it may affect other parts of the software. Such kind of ripple effects are largely affected by the software stability [3,5]. Software stability is defined as the resistance of a piece of software to the amplification of changes in the software [3]. If the software stability is poor, the impact of any change of the software may be large, and the maintenance cost will be high. But how to measure the software stability is still a problem faced by many people.

The topological structure of a software system explicates the structure of the software in terms of software entities such as methods/attributes, classes/interfaces, and packages, and their couplings. With the increase of the complexity, software structure has become one of the key factors greatly influencing the software quality [2]. Complex network theory provides an effective tool to study the software structure and its dynamics. In recent years, a few researchers introduced the complex network theory to software engineering domain by representing the software structure as a complex network (namely software network), and many shared physics-like laws of software systems have been revealed [6–9], which could later be applied in practice. The interdisciplinary research between complex networks and software engineering provides us a new way to study complex software systems, and it also provides a promising way to quantify the software stability.

The objective of this paper is to propose metrics to quantify the software stability from a complex network perspective, by using a software network representation of the software structure. To fulfill this task, the topological structure of software at the class level of granularity is first represented by a weighted directed software network named Class Coupling Network (CCN), in which classes are nodes, and their couplings are directed links which are annotated with weights corresponding to the direct coupling strength between classes. Second, by considering all the directed paths between every pair of class nodes in the software network, we compute the coupling strength between all pairs of classes by taking into consideration both the directed and indirected (transitive) coupling strength between classes, and a Node Influence Network (NIN) is built correspondingly to formally represent all pairs of classes and their couplings. Finally, based on the NIN, we propose a metric to quantify the class stability and further propose a metric to quantify the stability of software as a whole. The proposed two metrics are validated theoretically using widely accepted Weyuker's criteria [10]. Moreover, the proposed metric for software stability as a whole is validated empirically using five Java programs, and the results show it is an effective indicator of the software scalability. The usefulness of the proposed metric for class stability is evaluated empirically by correlation analysis with the change proneness of classes.

The main contributions of this paper can be summarized as follows:

- We propose a more accurate software network model (i.e., CCN) to represent software structure at the class level which takes into consideration both the coupling direction and strength.
- We propose a NIN network which considers both the directed and indirected coupling strength between classes. Based on the NIN, we develop metrics to quantify class stability and stability of software as a whole, respectively.
- The proposed metrics are validated theoretically using widely accepted evaluation criteria, and they are also evaluated empirically using open source Java programs.

The rest of this paper is organized as follows. Section 2 contains a brief and incomplete summary of the related work. Section 3 describes our approach in detail, with focus on the definitions of the software network models and software stability metrics. Sections 4 and 5 present the theoretical and empirical validation of our proposed metrics, respectively. We conclude in Sect. 6.

2 Related work

This paper can be categorized as the software metric research. In this section, we give a brief overview of the related work. It falls into two categories: (i) traditional software metrics, and (ii) software metrics based on complex networks.

2.1 Traditional software metrics

The traditional software metrics mainly contain the metrics for Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP).

Till now, lots of software metrics for POP have been proposed in the literature. Wolverton proposed *LOC* (line of code) metric to measure the productivity and efficiency of the programmers [11]. McCabe proposed the *Cyclomatic Complexity* to indicate the software complexity by measuring the number of linearly independent paths through the source code of software [12]. Halstead proposed the *Halstead* metric to identify measurable properties of software, and the relations between them [13]. Yin and Winchester proposed primary metrics and secondary metrics to evaluate the software design [14]. McClure proposed a complexity metric to measure the control structures [15]. Woodfield proposed a metric to measure the complexity of the component [16]. Henry and Kafura proposed a new set of information flow based metrics [17]. Tai proposed a data flow based metric to quantify the software complexity [18].

Moreover, a significant number of software metrics for OOP have also been proposed in the literature. Chidamber and Kemerer proposed the CK metrics to evaluate the software complexity from inheritance (*DIT* and *NOC*), class coupling (*RFC* and *CBO*), and in-class complexity (*WMC* and *LCOM*) [19]. Abreu et al. proposed the MOOD (metrics for object oriented design) metrics [20], which reflect a set of basic properties of the object-oriented (OO) paradigm including encapsulation (*MHF* and *AHF*), inheritance (*MIF* and *AIF*), polymorphism (*POF*), and message passing (*COF*). There also exist many other software metrics such as the metrics proposed by Abreu and Carapuca [21], the metrics proposed by Li and Henry [22], and the metrics proposed by Lorenz and Kidd [23].

2.2 Software metrics based on complex networks

In recent years, many researchers began to measure the software complexity using complex network theory, and many metrics has been proposed. Ma et al. proposed the *structural entropy* which is based on the degree of nodes to measure the order of a specific software structure [24]. They further examined the relationships between *structural entropy* and software robustness, and between *structural entropy* and the efficiency of the communication. Their work laid the basis for the software optimization [24]. Girolamo et al. proposed some betweenness-based metrics to identify problematic classes at different levels of granularity [25]. Ma et al. proposed a hierarchical metric suit and used it to analyze the software complexity from a macro-meso-micro perspective [26]. Jenkins et al. proposed a I_{cc} metric to measure the evolution stability of software across successive versions based on the classes and their couplings [27]. Vasa et al. proposed some metrics to quantify the evolution stability of some software properties, and some laws have been found, such as the size and complexity of class changed little over time, and classes with a large degree are tend to be modified [28]. Ma et al. applied network motif to explore the software structure [29], and found that sub-graphs with high statistical importance are hard to form rings and tend to be more stable. Gu et al. proposed metrics to quantify the class cohesion from a complex network perspective [30].

3 The proposed approach

Our approach to measure the software stability works as follows. First, we analyze the source code files of a specific software system to collect classes (interfaces) and their couplings which will be further represented by a CCN. Second, we further propose a NIN which considers both the directed and indirected (transitive) coupling strength between classes. Finally, we propose two metrics based on the NIN to quantify the class stability and software stability as a whole, respectively. Figure 1 shows an overview of our proposed approach. In the following subsections, we will discuss the main parts of our approach in detail.

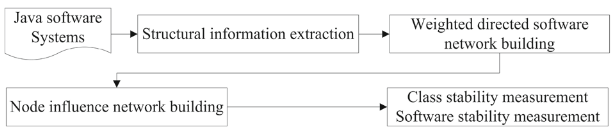

Java software Systems
Structural information extraction
Weighted directed software network building
Node influence network building
Class stability measurement
Software stability measurement

Fig. 1 An overview of the proposed approach

3.1 Software networks

In this paper, we focus our work on analyzing software systems coded in Java. It is mainly because Java has been one of the most widely used OO programming languages, and software systems developed by Java have a relatively clear internal structures which are amenable to extraction and analysis. However, it should be noted that, although we focus on Java software systems, our approach can be easily extended to software systems developed by other OO languages such as C++, C# and VB.NET.

A typical Java software system is usually composed of many software entities at different levels of granularity such as packages, classes/interfaces, and methods/attributes, which are interacting reciprocally by different types of couplings. We perform static analysis of the source code files of a Java software system to extract the structural information at the class level of granularity, i.e., we extract classes, interfaces, and the couplings between them. Note that we only consider classes/interfaces that are actually defined in the code, neglecting those that were only referenced to the imported package [31].

In our approach, the obtained static structural information will be formally represented by a CCN. Based on the CCN, we further build a NIN to represent class and their couplings.

3.1.1 Class coupling network

Definition 1 CCN is a weighted directed graph which represents the classes/interfaces and the couplings between them in a software system. Specifically, all classes and interfaces of a software system are modeled as nodes in the graph. The coupling between every pair of classes, every pair of interfaces, or every pair of class and interface is denoted by a directed link between the nodes. For example, if class i implements interface j, there exists a directed link $\langle n_i, n_j \rangle$ $(n_i \rightarrow n_j)$ in the graph, where n_i is the node denoting class i, and n_j is the node denoting interface j. We do not differentiate the class and interface and will treat them the same from here on. Every link in the graph is assigned a weight to signify the coupling strength between the two classes. The graph can be formally represented as $G = (N, L)$, where N is the node set and L is the link set.

Note that, for Java software systems, the link $\langle n_i, n_j \rangle$ can be defined under the following seven circumstances [32,33]:

– *Inheritance relation (INR)* If class i inherits from another class j via keyword *extends*.
– *Implements relation (IMR)* If class i realizes interface j via keyword *implements*.
– *Parameter relation (PAR)* If one of class i's methods has at least one parameter with type of class j.

- *Global variable relation (GVR)* If class i has at least one attribute with type of class j.
- *Method call relation (MCR)* If one of class i's methods calls a method on an object of class j.
- *Local variable relation (LVR)* If a local variable with type of class j is declared in one of class i's methods.
- *Return type relation (RTR)* If one of class i's methods has the return type class j.

The graph G is associated with an adjacency matrix Ψ to encode the coupling between every pair of classes in CCN. Its entry ψ_{ij} signifies the coupling between any pair of classes i and j:

$$\psi_{ij} = \begin{cases} w_{ij} \langle n_i, n_j \rangle \in L \\ 0 \quad otherwise \end{cases}, \tag{1}$$

Ψ is a $|N| \times |N|$ matrix. w_{ij} is the weight on the link $\langle n_i, n_j \rangle$, which denotes the coupling strength between n_i and n_j. Generally, a small value of w_{ij} indicates a low coupling strength between the classes that n_i and n_j denote.

The weights on the links of G allow us to consider the coupling strength between classes, which provides us a more accurate representation of the software structure. However, how to determine the weights on the links becomes a new problem that should be resolved. As mentioned above, there are many different types of couplings that can exist between two classes such as INR, IMR, and RTR. It is reported that the coupling strengths vary with the coupling types [32]. Moreover, the coupling frequency also affects the coupling strength between classes [32]. So, to measure the coupling strength between every pair of classes, we should consider both the coupling types and frequencies.

The coupling frequency can be obtained by simply counting their occurrences in the source code. However, estimating the coupling strength of different coupling types is a subject of empirical estimation [34]. There are many different frameworks that can be used to estimate it [32]. Here we will use the conceptual ideal proposed by Kang et al. [35–38] to assign the weights for different coupling types simply for its effectiveness has been demonstrated [36–38]. Kang et al. use an ordinal scale to quantify the relative coupling strength of different coupling types which is shown in Table 1. The weights, H_1–H_{10}, are arranged in an ascending order. Actually, the absolute values of the weights are not important but their relative ratios [35,38]. So in the current work, arbitrary value of 1–10 are respectively set to H_1–H_{10} as that of [35,38] do. Based on these values, we can compare the strengths between different coupling types.

As mentioned above, different types of coupling may exist between classes. Note that even a pair of classes, i and j, may have several coupling types at the same time. For example, as

Table 1 Ranking of strength of different coupling types between classes

No.	Coupling types	Strength
1	Dependency	H_1
2	Common association	H_2
3	Qualified association	H_3
4	Association class	H_4
5	Aggregation association	H_5
6	Composition association	H_6
7	Generalization (concrete parent)	H_7
8	Binding	H_8
9	Generalization (abstract parent)	H_9
10	Realize	H_{10}

the simple example shown in Fig. 2, class "Adoptor" has one local attribute with type class "Dog", and at the same time it has a method "getDog()" with return type class "Dog". So, when computing the final coupling strength between classes i and j, we should sum up the weights of different coupling types.

Once the coupling types and their corresponding coupling frequencies are obtained for any pair of classes, i and j, we can compute the final coupling strength w_{ij} on the link $n_i \to n_j$ as

$$w_{ij} = \sum_{k=1}^{10} f_{ij}^k \times H_k = \sum_{k=1}^{10} f_{ij}^k \times k, \tag{2}$$

where f_{ij}^k is the frequency of the kth coupling type from class i to class j, and H_k is the strength of the k-th coupling type in Table 1.

Due to the fact that we extract the structural information from the source code using a static analysis based approach, we are unable to recover coupling types between classes with a very fine granularity. But we managed to identify "Dependency" (identified by "method call", "parameter", "local variable", and "return type" relations defined in Definition 1), "Aggregation/Composition association" (identified by "global variable" relation defined in Definition 1), "Generalization" (identified by "inheritance" relation defined in Definition 1), and "Realize" (identified by "implements" relation defined in Definition 1) which respectively correspond to H_1, H_5/H_6, H_7/H_9, and H_{10} in Table 1. But we are also unable to differentiate "Aggregation" from "Composition" for they are usually distinguished semantically. The best way to recover them from the source code is to gain sufficient knowledge of the architecture of the software analyzed. It is a very hard work for those not involved in the development of the software. So, in the current work, we will treat "Aggregation" and "Composition" as the same relationship,

```
public interface Animal {
    public abstract void animalMethod()
}
class Mammal implements Animal {
    public void animalMethod() {
        System.out.println("Mammal");
    }
}
class Dog extends Mammal {
    public void animalMethod() {
        System.out.println("Dog");
    }
}
class Zoom {
    public void say() {
        System.out.println("Zoom");
    }
}
class Adoptor {
    private Zoom myZoom;
    public void setZoom(Zoom newZoom) {
        myZoom = newZoom;
    }
    public Dog getDog() {
        Dog myDog = new Dog();
        return myDog;
    }
    public void say() {
        myZoom.say();
    }
}
```

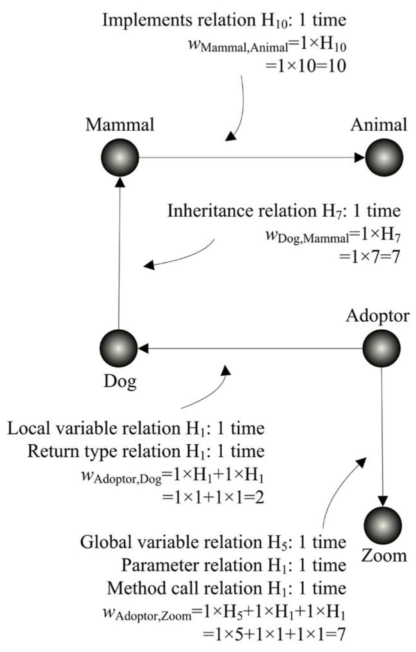

Implements relation H10: 1 time
wMammal,Animal=1×H10 =1×10=10
Mammal
Animal
Inheritance relation H7: 1 time
wDog,Mammal=1×H7 =1×7=7
Adoptor
Dog
Local variable relation H1: 1 time
Return type relation H1: 1 time
wAdoptor,Dog=1×H1+1×H1 =1×1+1×1=2
Global variable relation H5: 1 time
Parameter relation H1: 1 time
Method call relation H1: 1 time
wAdoptor,Zoom=1×H5+1×H1+1×H1 =1×5+1×1+1×1=7
Zoom

Fig. 2 A simple code segment (left) and its corresponding CCN (right)

and we use the strength of “Aggregation” to represent both scenarios.

To illustrate the idea to build CCN from the source code, we give a simple example in Fig. 2. However, due to the limitation of space, we only take Mammal → Animal as an example to show how to establish a link and compute the corresponding weight $w_{\text{Mammal,Animal}}$. Since class “Mammal” implements interface “Animal”, there is a link Mammal → Animal in CCN. At the same time, the frequency of this “implements” relationship is 1, and the strength of the “implements” relationship (i.e. “Realize” in Table 1) is H_{10}. So $w_{\text{Mammal,Animal}} = 1 \times \text{H}_{10}$. Other links and weights in Fig. 2 can be similarly established. The notes beside the link show the coupling types, coupling frequencies, and weights assigned to the link.

3.1.2 Node influence network

Based on the CCN, we further define a NIN to represent all classes of a software system and the coupling between all pairs of classes.

Definition 2 NIN is a weighted directed graph whose nodes represent the classes/interfaces of a software system, and links represent the direct and indirect couplings between nodes. It can formally be defined as $G' = (N, L')$, where N is same as that of CCN. L' is adapted from L of CCN, i.e., if there is no link between nodes n_i and n_j in CCN, but there is a directed path $n_i \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow n_j$ between nodes n_i and n_j, then we will add a link $\langle n_i, n_j \rangle$ between the nodes n_i and n_j in G'. The weight assigned to $\langle n_i, n_j \rangle$ is computed over all the directed paths between the two nodes n_i and n_j by

$$w'(i,j) = \sum_{r^k \in GIR(i,j)} (w(i,t_1^k) w(t_n^k, j) \prod_{a=0}^{n-2} w(t_{n-a-1}^k, t_{n-a}^k)), \tag{3}$$

where $w'(i,j)$ is the weight of $\langle n_i, n_j \rangle$, $GIR(i,j)$ returns all the directed paths between nodes n_i and n_j (indirected coupling), r^k denotes the k-th directed path, and t_l^k is the l-th node of the r^k.

If in CCN, there exist not only a link between nodes n_i and n_j, but directed paths $n_i \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow n_j$, then we will add a link $\langle n_i, n_j \rangle$ between the nodes n_i and n_j in G'. The weight assigned to $\langle n_i, n_j \rangle$ is computed over all the directed paths (indirected coupling) and the link (directed coupling) between the two nodes n_i and n_j by

$$w'(i,j) = \sum_{r^k \in GIR(i,j)} (w(i,t_1^k) w(t_n^k, j) \prod_{a=0}^{n-2} w(t_{n-a-1}^k, t_{n-a}^k)) + w(i,j), \tag{4}$$

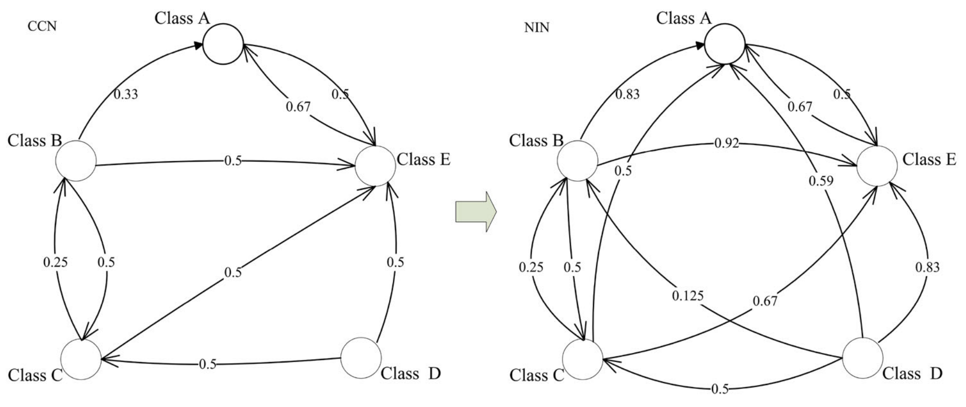


Fig. 3 A simple example to build NIN from CCN

To illustrate how to build the NIN from CCN, we give a simple example in Fig. 3, where the left part is the CCN and the right part is its corresponding NIN. As shown in the CCN of Fig. 3, there is no link between node "Class A" and "Class C", but there are three directed paths between them, i.e., "Class C" → "Class B" → "Class A", "Class C" → "Class B" → "Class E" → "Class A", and "Class C" → "Class E" → "Class A". So there is a link ⟨ClassC, ClassA⟩ in NIN, and according to Eq. (3), w'(ClassC, ClassA) = 0.25 × 0.33 + 0.25 × 0.5 × 0.67 + 0.5 × 0.67 = 0.08 + 0.335 + 0.08375 = 0.5. As shown in the CCN of Fig. 3, there is a link between node "Class B" and "Class A", and there exist two directed paths, i.e., "Class B" → "Class E" → "Class A" and "Class B" → "Class C" → "Class E" → "Class A". So, according to Eq. (4), w'(ClassB, ClassA) = 0.33 + 0.5 × 0.67 + 0.5 × 0.5 × 0.67 = 0.33 + 0.335 + 0.1675 = 0.8325 ≈ 0.83. Other links and weights in NIN of Fig. 3 can be similarly established.

3.2 Definitions of metrics

According to the stable design principle (SDP) [39], packages should only depend upon packages that are more stable than themselves. The structural stability of a package can be computed by counting the number of links that enter and leave the package. Generally, a package is more stable when its in-degree is high and its out-degree is low. In the current work, we applied SDP to a class node in NIN and calculated the class stability S_c of node n_i, S_c^i, in NIN by

$$S_c^i = \frac{wi^i}{wi^i + wo^i}, \tag{5}$$

where wi^i and wo^i denote the weighted in-degree and out-degree of node n_i, respectively. wi^i is the sum of the weights of the links with its head ends being adjacent to n_i. wo^i is the sum of the weights of the links with its tail ends being adjacent to n_i. For example, as shown in Fig. 3, the stability of "Class A" can be calculated as $S_c^{\text{ClassA}} = \frac{0.25+0.125}{0.25+0.125+0.5+0.92+0.83} = 0.14285714$. S_c^i has the range [0, 1], where $S_c^i = 0$ when $wi^i = 0$ indicates a maximally unstable class, and $S_c^i = 1$ when $wo^i = 0$ indicates a maximally stable class. Under this definition, n_i is more stable when its value of S_c^i is greater. Let $S_c^i = 0$ when $wi^i = 0$ and $wo^i = 0$.

Based on S_c, the stability of the software as a whole, S_s, can be defined as

$$S_s = \frac{1}{|N|}\sum_{i=1}^{|N|} S_c^i, \tag{6}$$

where $|N|$ is the number of nodes in NIN. Obviously, S_s is the average of S_c over all the nodes in NIN.

4 Theoretical validation

Our proposed metrics belong to the category of software complexity metrics. It can be used to measure the structural complexity of OO software systems. In previous literatures, Weyuker has proposed a set of properties for evaluating the usefulness of software metrics [10]. Although some researchers offered critique on these properties especially on the Property 9 [40,41], these properties do provide formal criteria for evaluating the behavior of a metric and are therefore widely adopted [19,36,42]. Our proposed metrics are evaluated against Weyuker's nine properties, which are paraphrased as follows (M denotes any software complexity metric).

Property 1 A complexity measure should not rate all programs as equally complex. That is, for two given programs P and Q, it can always be found such that: $M(P) \neq M(Q)$. Obviously, for two different Java software systems P and Q, they may have different sets of classes and internal structures, which results in different NINs for the two systems and different values for S_c and S_s. Therefore, our proposed metrics do adhere to Property 1.

Property 2 There are only finitely many programs of a given complexity. That is, if c is a non-negative number, there are only finitely many programs P with $M(P) = c$. Since the universe of discourse deals with at most a finite set of applications. There are only a finitely many programs with the same NIN and measurement values. Therefore, Property 2 is satisfied by our proposed metrics.

Property 3 There exist two different programs of the same complexity. That is, for two distinct programs P and Q, it can be found such that: $M(P) = M(Q)$. It is reasonable to assume there are two different programs having the same structure and measurement values. Therefore, Property 3 is satisfied by our proposed metrics.

Property 4 Two different programs with the same functionality need not have the same complexity. That is, there are functionally equivalent programs P and Q satisfying $M(P) \neq M(Q)$. The same set of functionalities can be designed or implemented in different ways, leading to different structures of NINs and measurement values. Therefore, Property 4 is satisfied.

Property 5 The complexity of a program segment should be less than or equal to the complexity of the whole program. Formally, for all programs P and Q, the following must hold: $M(P) \leq M(P+Q)$ and $M(Q) \leq M(P+Q)$. Because this property is only suitable for metrics that measure software size, and the metrics proposed in this paper is only used to measure software stability (not size related metrics), it is not suitable for evaluating our metrics.

Property 6 The resulting complexity of the composition of two program P and R is not necessarily the same as the composition of program Q and R, even though P and Q have the same complexity. Formally, there exist programs P, Q and R such that $M(P) = M(Q)$ and $M(P + R) \neq M(Q + R)$. Let P and Q be two different software systems with $M(P) = M(Q)$, there exists another software R that can be combined with P and Q. The combination may result in different NINs. Therefore, $M(P + R) \neq M(Q + R)$ and Property 6 is satisfied.

Property 7 If the statements within a program are permutated, the complexity of the resulting program is not necessarily equal to the complexity of the original program. That is, for two programs P and Q (Q is formed by permuting the order of the statements of P), it can be found such that: $M(P) \neq M(Q)$. This property is meaningful in traditional programming languages, while in Java software, changing the order of the statements does not affect the structure of NIN and the value of metrics. Therefore, Property 7 is not satisfied by our proposed metrics as it is not applicable to Java software systems.

Property 8 Renaming the attributes or methods has no effect on the measure. Formally, if P is a renaming of Q then $M(P) = M(Q)$. As our proposed metrics are independent of the name of the attributes or methods, our metrics satisfy Property 8.

Property 9 The complexity of the composition of two programs may be greater than the sum of the complexities of the two taken separately. Formally, there exist programs P and Q such that $M(P) + M(Q) < M(P + Q)$. Considering an extreme example where there is only one node in the NIN of P and Q, it is obvious that $M(P) = M(Q) = 0$. After composing P and Q, the resulting NIN may have a link connecting the two nodes with $M(P + Q) > 0$. Therefore, $M(P) + M(Q) < M(P + Q)$, satisfying Property 9.

As discussed above, our proposed metrics satisfy the majority of the properties proposed by Weyuker, only with two exceptions, Properties 5 and 7. By Property 5 it implies that the complexity should be increased monotonically. In our study, our proposed metrics do not satisfy this property. It may because our metrics are calculated in term of structure, instead of size. Such exception has also been observed in other work. Property 7 is not applicable to our metrics for they are initially produced for traditional languages rather than OO languages like Java.

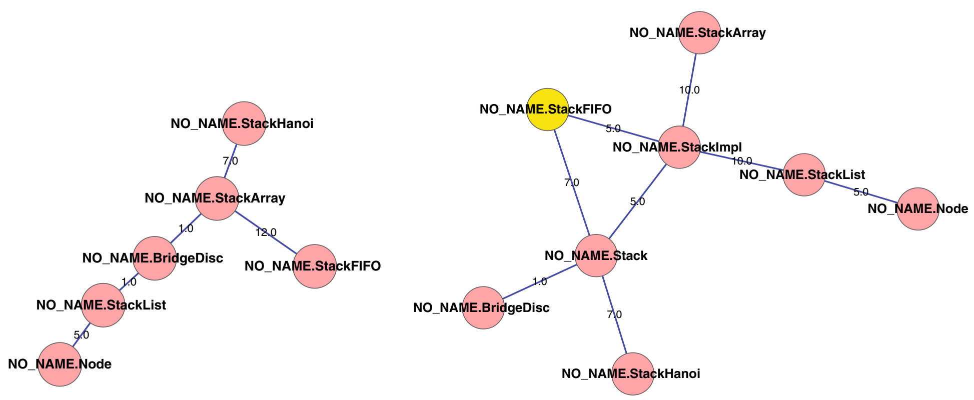
5 Empirical evaluation

In this section, we perform empirical studies to assess whether the proposed metrics are useful indicators of the external software qualities such as scalability and change proneness. We have applied the proposed metrics to measure a set of Java software system collected from online sources, and we briefly discussed some observations obtained from the empirical analysis. Our experiments were carried out on a PC at 2.6 GHz with 8 GB of RAM.

5.1 Research questions

In order to investigate the effectiveness of our proposed metrics as the predictors of external software qualities, we focus on the following two research questions (RQ):

- *RQ1 Is our proposed metric S_s a good indicator of software scalability?* As an effective indicator for software scalability, it should have the ability to identify the software with a better scalability from two software systems with the same functionalities. We wish to know whether our proposed metric S_s has such an ability.
- *RQ2 Is the proposed metric S_c a good indicator of the change proneness of classes?* As an effective indicator for



(a) before **(b)** after

Fig. 4 CCNs built from the programs before using bridge design patterns (the left part) and after using bridge design patterns (the right part). NO_NAME in the class name denotes the corresponding class is defined in an unnamed package, and the color of the nodes does not carry special meanings. See online versions for colors (Color figure online)

change proneness, it should also have the ability to indicate the change proneness of classes. We wish to know whether our proposed metric S_c has such an ability.

5.2 Answer to RQ1

Using design patterns in software development is widely accepted as an effective way to improve software scalability [43]. Such kind of software quality improvement should be captured by our proposed metric S_s. In order to answer RQ1, five Java programs have been examined, each of which has two versions. The two versions for each program have the same set of functionalities. Their only difference is one employs design patterns and one does not. It is estimated that the program developed by using design patterns should be easy to be extended, i.e., it has more extended points (classes whose $S_c < 1$), and the S_s should be smaller.

For illustration purpose, we shown in Fig. 4 the CCNs built from the programs before using bridge design pattern (the left part) and after using bridge design patterns (the right part). Enlarging the corresponding CCN can give you the details such as the name of the class each node denotes, the link between every pair of classes if it exists, and the weight on the link. These figures are automatically produced by our own developed analysis tool SNAP using the Spring layout algorithm [9].

The results of our proposed metrics applied to the subject programs are shown in Table 2, where #c is the number of classes whose $S_c < 1$.

As we can see from Table 2, in all the five programs, #c of the version employing design patterns is equal to or larger than that of the version which does not use design patterns, and S_s of the version employing design patterns is smaller than that of the version which does not use design patterns. The results are in line with our expectations, indicating our proposed S_s is an effective metric for software scalability.

Table 2 Results obtained from the five Java programs

Design pattern	Before		After	
	#c	S_s	#c	S_s
Bridge	4	0.36	6	0.34
Composite	1	0.67	3	0.38
Iterator	1	0.5	3	0.45
Sate	1	0.5	7	0.27
Decorator	6	0.38	6	0.22

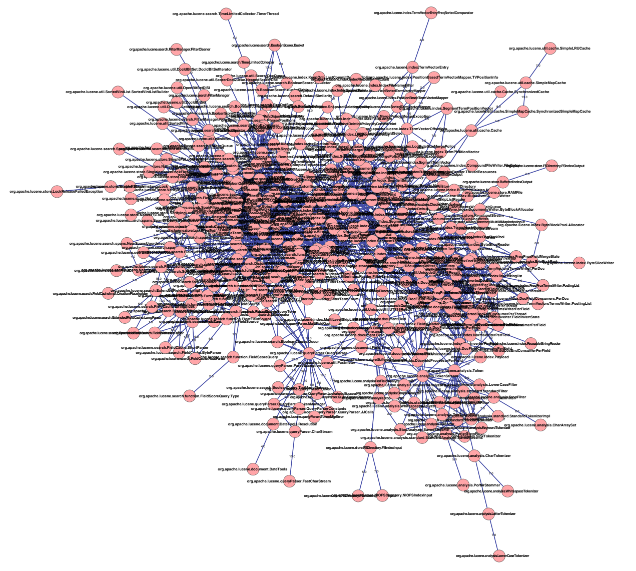


Fig. 5 The CCN built from lucene-2.4.0. The color of the nodes does not carry special meanings. The source file used to produce the Figure can be downloaded from http://pan.baidu.com/s/1pLn2FOn. See the online version for colors (Color figure online)

5.3 Answer to RQ2

It is not an reasonable design that all the classes of a software system are maximally stable. If this were the case, then the software system would be impossible to be extended. So,

Table 3 Pearson correlation analysis between ten metrics with the change proneness of classes

Software	*WMC*	*DIT*	*NOC*	*CBO*	*RFC*	*LCOM*	*NPM*	*LCOM3*	*LOC*	S_c
lucene-2.4	0.174**	0.265**	– 0.118*	0.048	0.261**	0.072	0.121*	– 0.310**	0.129*	– 0.430**
xalan-2.6	0.196**	0.258**	0.077**	0.310**	0.310**	0.107**	0.188**	– 0.045	– 0.237**	– 0.445**

**Correlation is significant at the 0.01 level (2-tailed)
*Correlation is significant at the 0.05 level (2-tailed)

S_c can be an indicator of the change proneness of classes. In order to answer RQ2, two open source software systems, lucene-2.4 and xalan-2.6, are selected as objects of study. We applied our approach to quantify the S_c of all the classes in the two systems, and analyzed the correlation between S_c of each class with their change proneness using Pearson correlation analysis. It is estimated that there exists a strong negative correlation between class stability and their change proneness. We also compare S_c with other nine OO metrics such as *WMC*, *DIT*, and *NOC* [44]. Note that the change proneness data of classes are computed using the approach proposed in [45].

For illustration purpose, we show in Fig. 5 the CCN built from the subject system lucene-2.4.0. Enlarging the CCN can give you the details such as the name of the class each node denotes, the link between every pair of classes if it exists, and the weight on the link.

Table 3 shows the pearson correlation analysis between 10 metrics and the change proneness of classes. The measurement data of the two subject systems can be downloaded from http://pan.baidu.com/s/1b1khbg. Obviously, we can see that, in the two subject systems, there exists a strong negative correlation between S_c and the change proneness of classes. The results are in line with our expectations, indicating our proposed S_c is an effective indicator for the change proneness of classes. Moreover, compared with other nine OO metrics, S_c is more effective, with its absolute value being much more closer to 1.

6 Conclusions

In this paper, we proposed two metrics, class stability (S_c) and software stability as a whole (S_s), to quantify the software stability. Our metrics proposed are based on two software networks, i.e., class coupling network (CCN) and node influence network (NIN). CCN is a weighted directed software network at the class level of granularity, where nodes represent classes and links represent their couplings. NIN is also a weighted directed software network, which is adapted from CCN by taking directed and indirected couplings into consideration.

We evaluated our metrics theoretically using widely accepted Weyuker's criteria and empirically using Java programs. The theoretically evaluation shows that the proposed S_c and S_s metrics satisfy most of Weyuker's properties, and empirical evaluation shows the effectiveness of our proposed metrics as indicators of the external software qualities such as scalability and change proneness. Moreover, our proposed S_c metric is better than other nine OO metrics in predicting the change proneness of classes.

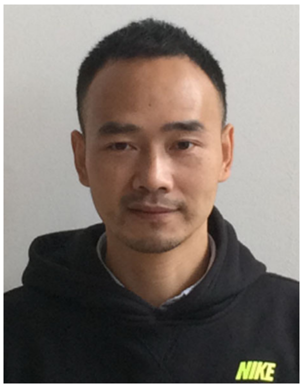
Acknowledgements This work was supported by the Zhejiang Provincial Nature Science Foundation of China (No. LY15F020004), the National Natural Science Foundation of China (Nos. 61402406 and 61603163), and the Commonweal Project of Science and Technology Department of Zhejiang Province (No. 2015C33091).

References

1. Pressman, R.S.: Software Engineering: A Practitioner's Approach, pp. 1–10. McGraw-Hill, London (1992)
2. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, pp. 5–10. International Thomson Computer Press, London (1996)
3. Yau, S., Collofello, J.S.: Design stability measures for software maintenance. IEEE Trans. Softw. Eng. **11**, 849–856 (1985)
4. IEEE Std. 610.12, Standard Glossary of Software Engineering Terminology, 10-12. IEEE Computer Society Press, CA (1990)
5. Yau, S., Collofello, J.S.: Some stability measures for software maintenance. IEEE Trans. Softw. Eng. **SE–6**, 545–552 (1980)
6. Myers, C.R.: Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. Phys. Rev. E **68**, 046116 (2003)
7. Potanin, A., Noble, J., Frean, M., Biddle, R.: Scale-free geometry in OO programs. Commun. ACM **48**, 99–103 (2005)
8. Concas, G., Marchesi, M., Pinna, S., Serra, N.: Power-laws in a large object-oriented software system. IEEE Trans. Softw. Eng. **33**, 687–708 (2007)
9. Pan, W.F., Li, B., Ma, Y.T., Liu, J.: Multi-granularity evolution analysis of software using complex network theory. J. Syst. Sci. Complex. **24**, 1068–1082 (2011)
10. Weyuker, E.J.: Evaluating software complexity measures. IEEE Trans. Softw. Eng. **14**, 1357–1365 (1988)
11. Wolverton, R.W.: The cost of developing large-scale software. IEEE Trans. Comput. **C–23**, 615–636 (1974)
12. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **SE–2**, 308–320 (1976)
13. Halstead, M.H.: Elements of software science. Oper. Program. Syst. **7**, 128–128 (1977)
14. Yin, B.H., Winchester, J.W.: The establishment and use of measures to evaluate the quality of software designs. ACM Sigmetrics Perform. Eval. Rev. **7**, 45–52 (1978)
15. McClure, C.L.: A model for program complexity analysis. In: Proceedings of the 3rd International Conference on Software Engineering, pp. 149–157 (1978)

16. Woodfield, N.: Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors, 6–8. Purdue University, West-Lafayette (1980)
17. Henry, S., Kafura, D.: Software structure metrics based on information flow. IEEE Trans. Softw. Eng. **SE–7**, 510–518 (1981)
18. Tai, K.C.: A program complexity metric based on data flow information in control graphs. In: Proceedings of the 7th International Conference on Software Engineering, pp. 239–248 (1984)
19. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object-oriented design. IEEE Trans. Softw. Eng. **20**, 476–493 (1994)
20. Abreu, B.F.: The MOOD metrics set. In: Proceedings of the ECOOP'95 Workshop on Metrics, pp. 150–152 (1995)
21. Abreu, B.F., Carapuca, R.: Candidate metrics for object-oriented software within a taxonomy framework. J. Syst. Softw. **26**, 87–96 (1994)
22. Li, W., Henry, S.: Object-oriented metrics that predict maintainability. J. Syst. Softw. **23**, 111–122 (1993)
23. Lorenz, M., Kidd, J.: Object-Oriented Software Metrics: A Practical Guide. Prentice Hall PTR, Englewood Cliffs (1994)
24. Ma, Y.T., He, K.Q., Du, D.H.: A qualitative method for measuring the structural complexity of software systems based on complex networks. In: Proceedings of the 12th Asia-Pacific Software Engineering Conference, pp. 257–263 (2005)
25. Girolamo, A., Newman, L.I., Rao, R.: The structure and behavior of class networks in object-oriented software design (2005). www.eecs.umich.edu/leenewm/documents/classnetworks.pdf
26. Ma, Y.T., He, K.Q., Du, D.H.: A complexity metrics set for large-scale object-oriented software systems. In: Proceedings of the 6th International Conference on Computer and Information Technology, pp. 257–263 (2006)
27. Jenkins, S., Kirk, S.R.: Software architecture graphs as complex networks: a novel partion scheme to measure stability and evolution. Inf. Sci. **177**, 2587–2601 (2007)
28. Vasa, R., Schneider, J.G., Nierstrasz, O.: The inevitable stability of software change. In: Proceedings of the 23rd IEEE International Conference on Software Maintenance, pp. 4–13 (2007)
29. Ma, Y., He, K.Q., Liu, J.: Network motifs in object-oriented software systems. Dyn. Contin., Discret. Impuls. Syst. Ser. B **16**, 166–172 (2007)
30. Gu, A., Zhou, X., Li, Z., Li, Q., Li, L.: Measuring object-oriented class cohesion based on complex. Networks **42**, 3551–3561 (2017)
31. Pan, W.F., Li, B., Ma, Y.T., Qin, Y.Y., Zhou, X.Y.: Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks. J. Comput. Sci. Technol. **25**, 1202–1213 (2010)
32. Briand, L.C., Daly, J.W., Wüst, J.K.: A unified framework for coupling measurement in object-oriented systems. IEEE Trans. Softw. Eng. **25**, 91–121 (1999)
33. Pan, W.F., Li, B., Liu, J., Ma, Y.T., Hu, B.: Analyzing the structure of Java software systems by weighted *k*-core decomposition, Future Generation Computer Systems (2017). https://doi.org/10.1016/j.future.2017.09.039
34. Briand, L., Devanbu, P., Melo, W.: An investigation into coupling measures for C++. In: Proceedings of the 19th International Conference Software Engineering, pp. 412–421 (1997)
35. Kang, D.Z., Xu, B.W., Lu, J.J., Chu, W.C.: A complexity measure for ontology based on UML. In: Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, pp. 222–228 (2004)
36. Zhang, H.Y., Li, Y.F., Tan, H.B.K.: Measuring design complexity of semantic web ontologies. J. Syst. Softw. **83**, 803–814 (2010)
37. Chong, C.Y., Lee, S.P., Ling, T.C.: Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach. Inf. Softw. Technol. **55**, 1994–2012 (2013)
38. Chong, C.Y., Lee, S.P.: Analyzing maintainability and reliability of object-oriented software using weighted complex network. J. Syst. Softw. **110**, 28–53 (2015)
39. Martin, R.C.: Agile Software Development, Principles, Patterns and Practices, pp. 50–57. Prentice Hall, Upper Saddle River (2002)
40. Cherniavsky, J.C., Smith, C.H.: On Weyuker's axioms for software complexity measures. IEEE Trans. Softw. Eng. **17**, 636–638 (1991)
41. Roy, G.: On the applicability of Weyuker Property 9 to object-oriented structural inheritance complexity metrics. IEEE Trans. Softw. Eng. **27**, 381–384 (2001)
42. Harrison, W.: OAn entropy-based measure of software complexity. IEEE Trans. Softw. Eng. **18**, 1025–1029 (1992)
43. Tsantalis, N., Chatzigeorgous, E., Stephanides, G., Halkidis, S.T.: Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. **32**, 896–909 (2006)
44. Madeyski, L., Jureczko, M.: Which process metrics can significantly improve defect prediction models? Empir. Study, Softw. Qual. J. **23**, 393–422 (2015)
45. Tsantalis, N., Chatzigeorgous, E., Stephanides, G.: Predicting the probability of change in object-oriented systems. IEEE Trans. Softw. Eng. **30**, 601–614 (2005)

Weifeng Pan received his Ph.D. degree from School of Computer at Wuhan University, China, in 2011. He is presently an associate professor in School of Computer Science and Information Engineering at Zhejiang Gongshang University. He is also a member of China Computer Federation (CCF) and ACM. His current research interests include software engineering, service computing, complex networks, and intelligent computation.



Chunlai Chai received his M.S. degree from School of Computer at Hohai University, China, in 2003. He is presently an associate professor in School of Computer Science and Information Engineering at Zhejiang Gongshang University. He is also a member of China Computer Federation (CCF) and ACM. His current research interests include software engineering and complex networks.