Research Article
Characterizing Software Stability via Change Propagation Simulation

Weifeng Pan,1 Haibo Jiang,1 Hua Ming,2 Chunlai Chai,1 Bi Chen,1 and Hao Li3

1School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, China
2School of Engineering and Computer Science, Oakland University, Rochester, MI 48309, USA
3Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA

Correspondence should be addressed to Weifeng Pan; panweifeng1982@gmail.com

Received 26 February 2019; Accepted 18 July 2019; Published 29 August 2019

Copyright © 2019 Weifeng Pan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Software stability means the resistance to the amplification of changes in software. It has become one of the most important attributes that affect maintenance cost. To control the maintenance cost, many approaches have been proposed to measure software stability. However, it is still a very difficult task to evaluate the software stability especially when software becomes very large and complex. In this paper, we propose to characterize software stability via change propagation simulation. First, we propose a class coupling network (CCN) to model software structure at the class level. Then, we analyze the change propagation process in the CCN by using a simulation way, and by doing so, we develop a novel metric, SS (software stability), to measure software stability. Our SS metric is validated theoretically using the widely accepted Weyuker’s properties and empirically using a set of open source Java software systems. The theoretical results show that our SS metric satisfies most of Weyuker’s properties with only two exceptions, and the empirical results show that our metric is an effective indicator for software quality improvement and class importance. Empirical results also show that our approach has the ability to be applied to large software systems.

1. Introduction

Software maintenance is widely regarded as the most costly and difficult phase in the life cycle of a piece of software [1]. It mainly consists of four phases, i.e., analyzing software, generating modification proposals, ripple effect analysis, and testing the modified software [2, 3]. It is estimated that software maintenance costs account for more than 60% of the total life cycle cost, and the third phase (ripple effect analysis) usually accounts for about 40% of the software maintenance cost [4]. Worse still, this maintenance cost shows no sign of declining. The cost being so high makes how to reduce it become a problem facing many people.

Generally, there are two ways to control the maintenance cost [2]. One is to provide effective tools or techniques to help maintenance personnel ease their maintenance tasks and improve their maintenance productivity. The other is to propose meaningful metrics to assess the quality characteristics of software and help maintenance personnel (or developers) identify potential problems in software. In this paper, we resort to the second way (i.e., developing some new metrics) to control the maintenance cost.

As mentioned above, ripple effect analysis is a key activity in software maintenance. Ripple effect can be defined as a phenomena that changes to one part of software will potentially affect other parts of the software [2, 3]. It is reported that software stability which means the resistance to the amplification of changes in software, is a primary attribute that affects ripple effect [2, 3]. Thus, we can control the maintenance cost by proposing meaningful metrics to measure software stability. But it is still a difficult task to measure software stability quantitatively.

Software structure is usually defined as the organization of software elements (i.e., methods/attributes, classes/interfaces, and packages) within a piece of software [5]. It includes a set of software elements and a set of connections between every pair of software elements. With the increase of software complexity, software structure has become one of the most important factors that affect software stability. Changes in one part of software can propagate to...
other parts of software along with the connections between software elements. Thus, we can analyze software stability from the perspective of software structure.

Software elements and the connections between them naturally form a network (or a graph) structure. Thus, software can be analyzed under the framework of complex network theory [6]. Many researchers applied complex network theory to analyze software systems and their dynamics by representing software structure as a software network, where nodes denote software elements and edges (or links) denote connections [6–9]. They found that software networks shared some physics-like laws such as “scale-free” and “small-world” [6–9]. It opens up a new path to study the existing research problems in software engineering from a complex network perspective. Until now, complex networks have been applied in many different domains of software engineering, such as software refactoring [10], software element ranking [11], and software testing [12]. Such novel interdisciplinary research also provides us a promising way to study software stability.

In this paper, we aim to characterize software stability via change propagation simulation. To fulfill this task, we propose a class coupling network (CCN) to model software structure at the class level, where nodes are classes (in this work, if not mentioned, the term “class” designates both class and interface from here on), and directed links between nodes are couplings between classes. Each link in the CCN is assigned a weight to denote the probability that a change in one class (the tail end of the link) will propagate to another class (the head end of the link) connecting to it. Then, we analyze the change propagation process in the CCN by using a simulation way, and by doing so, we develop a novel metric, SS (software stability), to measure software stability. Our SS metric is validated theoretically using the widely accepted evaluation criteria, and empirically using open source Java software systems. The theoretical results show that our metric satisfies most of Weyuker’s properties only with two exceptions, and the empirical results show that SS is an effective indicator for software quality improvement and class importance.

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

(i) We propose a novel metric, SS (software stability), to measure software stability at the class level. Our metric is based on a network representation of software structure and a change propagation simulation in the network.

(ii) Our class coupling network uses weights to denote the probability that a change in one class will propagate to other classes that link to it. Such a weight allows us to consider the influence of probability on the change propagation, which has been neglected in the existing literature.

(iii) We propose a simulation way to analyze the change propagation process in the CCN. The simulation way can also be applied to calculate the value of SS.

(iv) The proposed metric is validated theoretically using widely accepted evaluation criteria, and empirically using open source Java software systems. The data set and software used in this work are all published online to support further replication.

The rest of the paper is organized as follows. Section 2 reviews the related work on software change propagation and stability. Section 3 describes our approach in detail step-by-step. Section 4 contains the theoretical and empirical evaluation of our approach. We conclude the paper and discuss the future work in Section 5.

2. Related Work

Much effort has been made to analyze software change propagation and stability. Kung et al. [14] defined a concept, “class firewall”, to denote classes that may be affected by changes in a given class and used it to explore the change impact analysis in the class diagram of the software. Li et al. [15] proposed a composite metric, SDI, to characterize software design instability by capturing the evolutionary change in an object-oriented (OO) design. Grosser et al. [16] proposed a case-based reasoning approach for predicting the stability of software items from relevant metric data and results about the stress. Ratiu et al. [17] proposed a metric, Stab, to measure the stability of classes by characterizing the difference between two successive versions. MacCormack et al. [18] applied design structure matrices (DSMs) to map dependencies between elements of a design and defined a concept, “change cost”, to quantify the average influence of an element on the other parts of software. Shaik et al. [19] conducted experiments to show the role and use of change propagation to assess the design quality of software architectures. Liu et al. [20] proposed an unweighted directed software network at class level to represent classes and their couplings and introduced a metric named “average propagation ratio” to analyze the change propagation process in the software network. Li et al. and Zhang et al. [21, 22] proposed a change propagation model and defined a set of change impact metrics. They applied a simulation technique to simplify the calculation of the metrics. Pan [23] proposed a SOS metric to measure the stability of OO software systems at the method/attribute level. Alshayeb et al. [24] identified several factors that affect class stability and then used these factors to characterize class stability. Ampatzoglou et al. [25] conducted a case study to examine the stability of classes that participate in instances/occurrences of GoF design patterns. Badri et al. [26] presented several impact rules which are based on Java language constructs and further proposed a new change impact analysis model for Java programs. Abuhammad and Alshayeb [27] proposed a suite of software stability metrics for three UML diagrams at the model level, i.e., class, use case, and sequence. Pan and Chai [28] proposed a Node Influence Network (NIN) to represent classes and their couplings and further proposed several metrics to quantify class stability and software stability.

References [15–17, 24] measured design stability (or class stability) by characterizing the difference between two
successive versions. However, other existing work quantified design stability by a worst-case analysis of the software change propagation process. They usually assumed changes in one class definitely affect other classes linking to it directly and indirectly. They do not take into consideration the frequency of different couplings between classes. Our approach is very different from the existing work. We consider seven kinds of couplings between classes and also take into consideration their frequency. Furthermore, we take a simulation approach to analyze change propagation process and compute stability-related metrics. By using a simulation way, we consider the influence of the weight on edges to the change propagation and software stability.

3. The Approach

This paper proposes to measure software stability from a network perspective (CCN) by using a simulation way. It mainly contains four parts. First, we analyze the source code of software to extract meaningful information. Second, the extracted information will be formally represented by a CCN. Third, we propose a change propagation model and some metrics to analyze the change propagation process in the CCN. Finally, we propose a change propagation algorithm to calculate these metrics. The overall framework of our approach is illustrated in Figure 1, which will be discussed in detail one-by-one in the following subsections.

3.1. Java Software. We focus on the analysis of software coded in Java. The rationale is threefold: First, Java is the most widely used programming language according to TIOBE Index (TIOBE Index. Available at: https://www.tiobe.com/tiobe-index/), and analyzing Java software can give implications to a wider range of software engineering practitioners. Second, there are a large number of open source software systems that can be used as research subjects, ensuring the replication of research work. Third, our own developed analysis platform, SNAP [11], now can only analyze software coded in Java. However, the approach proposed in this paper is general enough, and it can be used in software coded in other languages such as C++, C#, and VB.NET, once these systems are represented by CCNs.

3.2. Software Information Extraction. Software elements are usually hierarchically organized. Upper level elements are composed of lower level ones. For example, packages are composed of classes, which are in turn composed of methods and attributes. Software elements at different levels usually interact reciprocally by different types of couplings.

To represent software structure as software networks at the class level, we should extract the structure information of the software at the class level. Specifically, we perform static analysis of the source code of Java software to extract classes and the coupling between them such as “inheritance relation”, “implements relation”, and “parameter relation”. In the analysis process, we consider the direction and frequency of each coupling type.

3.3. CCN Definition. As mentioned above, the extracted information will be formally represented by CCN. Thus, in this section, we first give the definition of CCN.

Definition 1 (CCN). CCN is a weighted directed graph that represents the coupling relations between classes/interfaces in a specific software system. Specifically, all classes/interfaces in the software are modeled as graph nodes while the coupling relations are modeled as weighted directed links between nodes. For example, if class $i$’s methods call methods of class $j$, there exists a direct link $⟨n_i, n_j⟩ (n_i \rightarrow n_j)$ in the graph, where $n_i$ is the caller class, and $n_j$ is the callee class. Each link is assigned a weight to denote the probability that a change in one class (the tail end of the link) will propagate to another class (the head end of the link) connecting to it. In this work, we do not differentiate classes and interfaces, treating them as the same thing. Formally, CCN can be represented as $CCN = (N, L)$, where $N$ is the node set and $L$ is the link set.

Java software systems usually have the following seven possible coupling relations [11, 29]:

(i) Inheritance relation (INR): if class $i$ inherits from another class $j$ via keyword extends.

(ii) Implements relation (IMR): if class $i$ realizes interface $j$ via keyword implements.

(iii) Parameter relation (PAR): if one of class $i$’s methods has at least one parameter with type of class $j$. 

Figure 1: The overall framework of our approach.
(iv) Global variable relation (GVR): if class $i$ has at least one attribute with type of class $j$.

(v) Method call relation (MCR): if one of class $i$’s methods calls a method on an object of class $j$.

(vi) Local variable relation (LVR): if a local variable with type of class $j$ is declared in one of class $i$’s methods.

(vii) Return type relation (RTR): if one of class $i$’s methods has the return type class $j$.

Thus, the link $(n_i, n_j)$ of CCN can also be defined under the above seven circumstances, i.e., if there exists at least one of the seven coupling relations from class $i$ to class $j$, then there will be a link $(n_i, n_j)$. Weights on all the links of CCN can be encoded as an $|N| \times |N|$ adjacency matrix $\Psi$, whose entry $\psi_{ij}$ signifies the change propagation probability from classes $i$ to $j$, i.e.,

$$\psi_{ij} = \begin{cases} w_{ij} & (n_i, n_j) \in L \\ 0 & \text{otherwise} \end{cases},$$

where $w_{ij}$ denotes the weight (change propagation probability) on the link $(n_i, n_j)$.

Considering the weight of links provides us a more accurate representation of software. But how to assign weights to links is still a problem that should be resolved. As mentioned above, there are total seven types of possible couplings that might exist between classes. In other words, for a specific pair of classes, $i$ and $j$, there may exist several different types of couplings at the same time. For example, as the source code segment shown in Figure 2, class “Adoptor” has one global variable “myZoom” with type “Zoom”, one method “setZoom(Zoom newZoom)” with one parameter “newZoom” of type “Zoom”, and one method “say()” with a call to another method “say()” in class “Zoom”.

In Section 3.2, we obtain couplings and their frequencies between every pair of classes if they have connections. Thus, for a specific pair of classes, $i$ and $j$, the weight $w_{ij}$ on the link $n_i \rightarrow n_j$ can be computed as

$$w_{ij} = \frac{1}{w_{\text{max}}} \sum_{k \in T} f^k_{ij}$$

where $T = \{\text{INR, IMR, PAR, GVR, MCR, LVR, RTR}\}$, $f^k_{ij}$ is the frequency of the $k$ coupling type from class $i$ to class $j$, and $w_{\text{max}}$ is the maximum value of the weights on all the links, i.e., $w_{\text{max}} = \max_{i,j\in V}(w_{ij})$. As shown in Figure 2, the total coupling
frequency on the link from class “Adoptor” to class “Zoom” is 3, and the maximum value of weights on the links is also 3. Thus, $w_{\text{Adoptor,Zoom}} = (1 + 1 + 1)/3 = 1$. Weights on other links in Figure 2 can be similarly computed.

### 3.4. Change Propagation Model

In this section, we will give our change propagation model in detail, with focus on the atomic changes that we consider, the change propagation process, and a simple example to illustrate the change propagation process one-by-one.

#### 3.4.1. Catalog of Changes

To simulate the change propagation process in CCN, we should transform source code edits into a set of atomic changes. As we know, CCN consists of nodes and links. Thus, changes to CCN can be roughly classified into six groups, i.e., adding new nodes, adding new links, deleting old nodes, deleting old links, changing old nodes, and changing old links. Since we analyze the change propagation process based on the static structure of CCN, changes that reconstruct the topological structure of CCN will be ignored. That is, changes on adding new nodes, adding new links, deleting old nodes, and deleting old links are not taken into consideration.

In this work, the atomic changes that we consider for CCN are summarized in Table 1. Obviously, all these atomic changes are concern with the changing old nodes, i.e., changing the modifier or content (methods and attributes) of classes. We assume any source code change edits can be decomposed into a set of atomic changes defined in Table 1. Note that AEM, DEM, AA, and DA are considered in this work, simply for they will not affect the topological structure of CCN. Further, we assume the original and the changed versions of software systems are both syntactically correct and compilable.

### 3.4.2. Change Propagation Process

Software has an intrinsic property to evolve. Fault corrections, performance improvements, and adaption to new environments all will push software to make necessary changes. When one node of CCN changes, the change will propagate to other parts of CCN along links. In this section, we will describe the change propagation process in CCN in detail. But before that, we will first give some hypotheses to simplify the analysis process.

#### Hypothesis 2

When change requirements arrive, each node in CCN can be changed. Suppose the initial change probability of node $i$ is $p(i)$, $i \in [1, |N|]$ ($|N|$ is the number of nodes in CCN), then it should satisfy $\sum_{i=1}^{\infty} p(i) = 1$. Generally, the value of $p(i), i \in [1, |N|]$ may vary with nodes. Since the randomness of maintenance activities, we cannot predict when the next maintenance activity will occur [2]. That is, we cannot estimate the specific value of $p(i), i \in [1, |N|]$ for each node since we cannot estimate the future change requirement. Thus, in this work, we simply suppose each node has an equal probability to be changed, i.e., $p(i) = 1/|N|, i \in [1, |N|]$.

#### Hypothesis 3

As shown in Table 1, there are eight types of atomic changes. However, we cannot differentiate them from each other from a network perspective. Thus, in this work, we will be treating all the atomic changes as the same change operation, namely change, regardless of the nature of each atomic change.

#### Hypothesis 4

Generally, any change requirement may cause multiple nodes to change. That is, the number of initial changed nodes may be more than one. However, due to the random nature of maintenance activities, we cannot predict what each activity will consist of [2]. That is, we cannot estimate the specific number of initial changed nodes. For simplicity, we suppose only one node is changing at any one time, and any change requirement will be fulfilled by a series of such changes. That is, at one time, one of the initial changed nodes will be selected until there are no nodes left.

#### Hypothesis 5

If one node of CCN changes, the change propagation process begins. The change propagates from the end node of the link to the head node of the link with a specific probability (weight on the link), and further propagates to other parts of CCN. For example, any change in node $i$ will propagate to node $j$ following link $i \rightarrow j$. The propagation process will not stop until there are no new nodes that are affected by the initial changed nodes.

Based on the CCN and the hypotheses mentioned above, we can describe the change propagation process from a network perspective. The propagation process of changes in one node of CCN to other parts of CCN is shown as follows:

#### Step 1

Input the CCN of software, set all elements of array $b\text{Changed}[]$ as false, and prepare a queue $c\text{Queue}$.

#### Step 2

Select one node $i$ randomly, set its status as “changed” (i.e., $b\text{Changed}[i]=true$), and push node $i$ into $c\text{Queue}$.

#### Step 3

If there is no element in $c\text{Queue}$, then go to Step 5; otherwise, go to Step 4.

#### Step 4

Pop up the first node $j$ from $c\text{Queue}$, and generate a random decimal $r \in [0, 1]$; iterate the node $k$ in CCN one-by-one, if there is link from $k$ to $j$ (i.e., $j \rightarrow k$), $r \leq w_{kj}$ ($w_{kj}$ is the weight on the $j \rightarrow k$), and $b\text{Changed}[k]$ is false, then

| Abbreviation | Atomic Change Name |
|--------------|--------------------|
| CMC          | Change the modifier of a class |
| AEM          | Add an empty method |
| DEM          | Delete an empty method |
| CMB          | Change body of a method without introducing/deleting new/old couplings |
| CHM          | Change the header of a method |
| AA           | Add an attribute without introducing new couplings |
| DA           | Delete an attribute without deleting couplings |
| CMA          | Change the modifier of an attribute |

#### Table 1: Categories of atomic changes.
Step 3) and the second generated random number is again 0.6, we then push V2 into cQueue (i.e., cQueue=[V2]) and set bChanged[2]=true (See (2) of Figure 4).

(v) Since cQueue is not null (i.e., CQueue=[V3]), pop up the first node "V3" from cQueue. Suppose the generated random number is again 0.3. There are no nodes linking to "V3". cQueue will be null. Thus, the change propagation process stops. bChanged={true,true,true,false} signifies "V1", "V2", and "V3" are affected by the initial node "V1" in the propagation process (See (4) of Figure 4).

Note that the change propagation route (V1 — V2 — V3) showed above is just one of the possible change propagation routes. The change propagation route we obtain in a specific simulation is affected by the random decimal r. For example, we may get another change propagation route as V1 — {V2,V3} — V4, if the first generated r = 0.4 (at Step 3) and the second generated r = 0.3 (at Step 4).

3.5. Metric Definition. To characterize the software stability, we should quantify the change propagation quantitatively. Thus, we define a set of metrics as follows.

Definition 6 (propagation basin (PB)). PB(i) is defined as the set of nodes in CCN that are affected by changes in one randomly selected node in the i-th simulation. Thus, the number of changed nodes in the i-th simulation is denoted as pb(i) = |PB(i)|.

Definition 7 (ration of changed nodes (RCN)). RCN of the i-th simulation, RCN(i), is defined as the ratio that pb(i) accounts for the total number of nodes in CCN. It can be calculated as RCN(i) = (pb(i)/|N|) * 100%, where |N| is the number of nodes in CCN.

Note that, for two independent simulations, i and j, their PBs and RCNs are usually different, i.e., PB(i) ≠ PB(j) and RCN(i) ≠ RCN(j). The reason is that we randomly select the initial changed node and use a random decimal r in the simulation (see Section 3.4.2).

Definition 8 (average number of changed nodes (ANCN)). ANCN of the T independent simulations, ANCN(T), is defined as the average number of changed nodes over T independent simulations. ANCN(T) can be defined as ANCN(T) = (1/T) ∑_{i=1}^{T} pb(i). Suppose ARCN_n(T) is the ARCN(T) of any node n in N (i.e., the randomly selected initial node is fixed to node n).

Definition 9 (average ration of changed nodes (ARCN)). ARCN of the T independent simulations, ARCN(T), is defined as the ratio that ANCN(T) accounts for the total number of nodes in CCN. It can be calculated as ARCN(T) = (ANCN(T)/|N|) * 100%, where |N| is the number of nodes in CCN, and T is the number of simulations.

Since RCN varies with independent simulations, we cannot use it to quantify the change propagation process.
4. Evaluation

In this section, we theoretically and empirically evaluate the effectiveness of our proposed SS metric. Our aim is to check whether SS is an effective metric to assess the stability of a piece of software. Our theoretical evaluation is performed using widely used criteria, while our empirical evaluation is performed using a set of Java software systems. All the experiments were carried out on a PC at 2.6 GHz with 8 GB of RAM.

4.1. Research Questions. In the evaluation, we aimed to address the following five research questions (RQs):

(i) RQ1: Does SS converge to a stable value? As a practical metric to characterize software stability, the value of SS should be relatively stable. However, we use a simulation way to compute the value of SS, introducing some randomness. We wish to know whether the SS we compute for a specific piece of software converges to a relative stable value.

(ii) RQ2: Does SS satisfy Weyuker’s nine properties? Weyuker’s nine properties are widely used to evaluate the usefulness of any software metric. We wish to know whether SS also satisfies Weyuker’s nine properties.

(iii) RQ3: Is SS a good indicator for software quality improvement? There are many functionally equivalent software systems. We wish to know whether SS has the ability to identify high-quality software from two functionally equivalent software systems.

(iv) RQ4: Is ARCN_n(T) a good indicator for class importance? Important (or key) classes have a “controlling function” in a software system, providing services for a large amount of other classes. ARCN_n(T) can be used as a metric to quantify the influence of a specific class to other classes in the change propagation process. Thus, we wish to know whether ARCN_n(T) is a good indicator for class importance, with the hope to indirectly validate the effectiveness of SS.

(v) RQ5: Can SS be applied to large software systems? As a practical metric, SS may be applied to large software systems. We wish to know whether SS has the ability to be applied to large software systems.

4.2. Answers to Research Questions. We evaluate the effectiveness of the proposed SS metric theoretically and empirically to answer the research questions raised in Section 4.1.

4.2.1. RQ1: Does SS Converge to a Stable Value? As mentioned above, we use a simulation way to compute SS. Since the randomness of our approach owing to r and T, the SS we obtained for a specific piece of software may vary with independent simulations. However, as a metric that can be used to characterize software stability, its value should not change greatly, and have better convergence to a relative stable value. In this section, we perform experiments to examine whether SS converges to a relative stable value.

(i) Subject Systems. We choose a set of ten widely used Java software systems as our subject systems. Table 2 gives some...
Input:

- CNN, |N| (number of nodes in CCN), T (number of simulations).

Output:

- Stop the change propagation process and print metric values.

(1) \( t = 1, \text{sum} = 0; \)
(2) while \( t \leq T \) do
(3) \( pb = 0; \)
(4) for \( q = 1 \) to |N| do
(5) \( \text{bChanged}[q] = \text{false}; \)
(6) end for
(7) \( i = \text{rnd}(1,|N|); /\text{rnd}() \) is a function to return an integer in \([1,|N|]\)
(8) \( pb++; \)
(9) \( \text{bChanged}[i] = \text{true}; \)
(10) \( i \rightarrow \text{cQueue} // \text{push } i \text{ into queue cQueue} \)
(11) while cQueue is not null do
(12) \( j \leftarrow \text{cQueue} // \text{pop up the first node, } j, \text{ from cQueue} \)
(13) \( r = \text{random}(0,1); /\text{random()} \) is a function to return decimal in \([0,1]\)
(14) for \( q = 0 \) to |N| do
(15) if \( w(q,i) > 0 \&\& r \leq w(q,i) \) then
(16) if \( \text{bChanged}[q] \) equals false then
(17) \( q \rightarrow \text{cQueue} // \text{push } q \text{ into queue cQueue} \)
(18) \( \text{bChanged}[q] = \text{true}; \)
(19) \( pb++; \)
(20) end if
(21) end if
(22) end for
(23) end while
(24) print \( pb; // pb(i) \)
(25) print bChanged array; // PB(i)
(26) print RCN = \( pb/|N|; // \text{RCN}(i) \)
(27) \( \text{sum} = \text{sum} + pb; \)
(28) end while
(29) print ANCN = \( \text{sum}/T; // \text{ANCN}(T) \)
(30) print ARCN = \( \text{sum}/(T \times|N|); // \text{ARCN}(T) \)
(31) print SS = 1 - ARCN; // SS

Algorithm 1: Flow of the change propagation algorithm.

### Table 2: Descriptions of the subject software systems.

| Systems  | Version | Domain | Directory | KLOC   | #P  | #C  | #M   | #A     | URL                               |
|----------|---------|--------|-----------|--------|-----|-----|------|--------|-----------------------------------|
| jdk      | 1.3.0   | sdk    | src       | 238.606| 79  | 2833| 25134| 9999   | https://www.oracle.com/java/     |
| tomcat   | 6.0.20  | middleware | src  | 156.911| 103 | 1,395| 15136| 7182   | http://tomcat.apache.org        |
| ant      | 1.6.1   | parsers | src/main  | 81.515 | 67  | 900 | 7691 | 4167   | http://ant.apache.org           |
| jfreechart| 1.0.19  | tool   | source    | 97.880 | 40  | 648 | 9006 | 2940   | http://www.jfree.org/jfreechart/ |
| struts   | 2.5.2   | middleware | src/core | 48.347 | 68  | 633 | 3518 | 1571   | https://struts.apache.org       |
| mybatis  | 3       | middleware | src  | 20.385 | 54  | 317 | 2411 | 782    | http://www.mybatis.org         |

We also show some size-related metrics, i.e., thousand lines of code (KLOC), number of packages (#P), number of classes (#C), number of methods (#M), and number of attributes (#A). These values are computed from the Java code in the directory listed in the “Directory” column, not the whole distribution of the corresponding software.

(2) Metrics. To check whether SS is stable, we use the standard deviation of SS (stdSS for short) to quantify the difference between two independent calculations of SS. Specifically, stdSS is defined as

\[
\text{stdSS} = \sqrt{\frac{\sum_{i=1}^{U} (SS_i - \text{avgSS})^2}{U - 1}},
\]
(a) CCN for jdk
(b) CCN for tomcat
(c) CCN for ant
(d) CCN for jfreechart
(e) CCN for struts
(f) CCN for struts

Figure 5: CCNs for the six subject systems. The source files of these figures can be downloaded from https://www.icloud.com/iclouddrive/0sSM4U7GmiTLpLfp0kxcD5g#complexity94162supp

where $U$ denotes $U$ independent calculations of $SS$, $SS_i$ is the $SS$ value of the $i$-th calculation, and $avgSS = (1/U) \sum_{i=1}^{U} SS_i$.

Generally, if $SS$ is absolutely stable, $stdSS = 0$. However, since we use a simulation way to compute $SS$, $stdSS$ may not always exactly equal 0. But if $stdSS$ is very close to 0, we consider $SS$ is acceptably stable.

(3) Experiment Process and Results Analysis. We follow the steps shown in Figure 1 to parse the source code, extract information, and build CCNs for each subject system. Figures 5(a)–5(f) show the CCNs we build for the subject systems. Enlarging the corresponding figure can give you more information of the figure, such as the class that each node denotes, and the coupling between classes.

There is only one parameter (i.e., $T$) that should be set before executing Algorithm 1. To compute $stdSS$, we should set another parameter $U$ in formula (3). Thus, we analyze the convergence of $SS$ by different settings of $T$ and $U$. Specifically, we set $T$ from 1 to 10,000 at interval 100, and set $U$ from 1 to 100 at interval 10. Figures 6(a) and 6(b) show $SS$ versus $T$ and $stdSS$ versus $U$, respectively.

As shown in Figure 6(a), we can observe that, for all the examined systems, $SS$ fluctuates slightly at the early period of our simulation ($T < 2,000$ for the examined systems). But after that, its value is relatively stable. As shown in Figure 6(b), $stdSS < 1E - 4$ for all the examined systems, signifying $SS$ is very stable and can be used as a feasible metric. Further, we also observe that the quality of all the examined systems is high, with their $SS$ being larger than 0.9955.

4.2.2. RQ2: Does $SS$ Satisfy Weyuker’s Nine Properties? Weyuker proposed a set of nine properties to theoretically validate the efficiency and robustness for various software complexity measures [13]. These properties are designed to analyze whether a specific metric qualifies as an effective measure. Following the suggestion given by [30], we also evaluate our $SS$ metric against Weyuker’s nine properties with the goal to answer RQ2. The theoretical evaluation is paraphrased as follows, where $M$ denotes any software complexity metric (in this work, $M$ denotes $SS$).

Property II (noncoarseness). $(\exists P)(\exists Q)(M(P) \neq M(Q))$, where $P$ and $Q$ are two different programs.

For two different Java software systems $P$ and $Q$, their internal structures at the class level may be different. In other words, they may have a different set of classes or a different set of links. Thus, their CCNs are also different, which results in different values of $SS$ for the two different programs. Therefore, $SS$ does adhere to Property II.
Property 12 (granularity). Let $c$ be a nonnegative number, then there are only finitely many programs $P$ with $M(P) = c$.

The universe of discourse deals with a finite number of applications. Thus, there are only a finite number of programs having the same CCNs which satisfy $SS = c$. Therefore, Property 12 is satisfied by $SS$.

Property 13 (nonuniqueness). There are two different programs $P$ and $Q$ such that $M(P) = M(Q)$.

A lot of programs have been developed. There is a chance that two programs have same CCNs. Furthermore, it is easy to build two programs with same CCNs and SS values. Therefore, Property 13 is satisfied by $SS$.

Property 14 (design details are important). $(\exists P)(\exists Q)(P \equiv Q & M(P) \neq M(Q))$.

This property states that even if two programs have the same functionalities, they may differ in complexity. A same set of functionalities can be implemented in different ways. For example, software systems shown in Table 3 all have two versions. One applies design pattern, and the other does not apply design pattern. But the two versions have equivalent functionalities. From Table 4, we can see that the two versions have different $SS$ values. Therefore, Property 14 is satisfied by $SS$.

Property 15 (monotonicity). $(\exists P)(\exists Q)(M(P) \leq M(P + Q) & M(P) \leq M(P + Q))$.

Property 15 is proposed for metrics that are used to measure software size. $SS$ is not a size-related metric. Thus, Property 15 is not applicable to evaluate $SS$.

Property 16 (nonequivalence of Interaction). $(\exists P)(\exists Q)(\exists R)(M(P) = M(Q) & M(P + R) = M(Q + R))$.

$P$ and $Q$ are two different programs with $M(P) = M(Q)$. $R$ is another program that can be combined with $P$ and $Q$. The combination of $P$ and $R$ may produce a CCN that is different from that of the combination of $Q$ and $R$. Thus, Property 16 is satisfied by $SS$.

Property 17 (significance of permutation). 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 does not hold for any OO metric, since the alteration of the order of the statements does not affect the CCN and its corresponding $SS$. Thus, Property 17 is not satisfied by $SS$.

Property 18 (no change on renaming). If $P$ is a renaming of $Q$ then $M(P) = M(Q)$.

As CCN is independent of the name of program, $SS$ satisfies Property 18.

Property 19 (interaction increases complexity). $(\exists P)(\exists Q)(\exists R)(M(P) + M(Q) < M(P + R))$.

This property states complexity may increase with the interaction of classes. Suppose $P$ and $Q$ are two different programs only with one node in their CCNs, then we can obtain $M(P) = M(Q) = 0$. Combining $P$ and $Q$ may result in a new CCN with links connecting nodes in the CCN. Thus, $M(P + Q) > 0$ and $M(P) + M(Q) < M(P + Q)$. $SS$ satisfies Property 19.

As discussed above, $SS$ satisfies seven out of nine of Weyuker’s properties. The two exceptions are Properties 15 and 17. Property 15 is only applicable to size-related metrics, and Property 17 does not hold for any OO metric. Such exceptions have also been observed in other work [31–35]. Thus, $SS$ is a well-structured metric that can be used for computing software stability.
Table 3: Descriptions of the subject software systems.

| Design Pattern | Version | LOC | #P | #C | #M | #A |
|----------------|---------|-----|----|----|----|----|
| Bridge         | before  | 86  | 1  | 6  | 15 | 8  |
|                | after   | 102 | 1  | 8  | 31 | 9  |
| Builder        | before  | 130 | 1  | 4  | 8  | 3  |
|                | after   | 161 | 1  | 6  | 19 | 10 |
| Composite      | before  | 59  | 1  | 3  | 6  | 4  |
|                | after   | 60  | 1  | 4  | 7  | 4  |
| Decorator      | before  | 34  | 1  | 7  | 10 | 3  |
|                | after   | 39  | 1  | 7  | 14 | 1  |
| Iterator       | before  | 112 | 1  | 2  | 3  | 1  |
|                | after   | 174 | 1  | 3  | 8  | 4  |
| State          | before  | 61  | 1  | 2  | 4  | 1  |
|                | after   | 83  | 1  | 7  | 10 | 1  |

Table 4: Descriptions of the subject software systems.

| Design Pattern | SS before | SS after | Design Pattern | SS before | SS after |
|----------------|-----------|----------|----------------|-----------|----------|
| Bridge         | 0.776467  | 0.7921   | Decorator      | 0.707429  | 0.783514 |
| Builder        | 0.561775  | 0.773917 | Iterator       | 0.25345   | 0.4613   |
| Composite      | 0.491367  | 0.609525 | State          | 0.2503    | 0.760243 |

4.2.3. RQ3: Is SS a Good Indicator for Software Quality Improvement? It is a widely accepted agreement that design pattern is a best practice for design changes, and can improve the quality of software design [36]. SS is an internal quality indicator of software. It should have the ability to identify the higher-quality software from two functionally equivalent software systems.

(1) Subject Systems. We use six pairs of software systems (see Table 3). In each pair, there are two different versions with the same set of functionalities. The only difference between two versions is one does not apply any kind of design pattern (before version), the other applies one kind of design pattern. Thus, based on the six pairs of software systems, we can check whether the design pattern can improve the quality of software. Table 3 shows some simple descriptions of the examined software systems, including the design pattern they used, lines of code (LOC), number of packages (#P), number of classes (#C), number of methods (#M), and number of attributes (#A).

(2) Experiment Process and Results Analysis. We follow the steps shown in Figure 1 to parse the source code, extract information, and build CCNs for the six pairs of software systems. Figures 7(a)–7(l) show the CCNs we build for the subject systems. Enlarging the corresponding figure can give you more information of the figure, such as the class that each node denotes, the coupling between classes, and the weight on each edge. Note that weights on the edges denote the coupling frequency. They are not the weights that we mentioned in the CCN definition. To obtain the weight on the edges in each CCN, for example, in the CCN shown in Figure 7(a), weight on the edge between “NO_NAME.StackList” and “NO_NAME.BridgeDisc” is 1.0. The maximum weight in the CCN is 8 on the edge between “NO_NAME.StackFIFO” and “NO_NAME.StackArray”. Thus, the true weight on the edge between “NO_NAME.StackList” and “NO_NAME.BridgeDisc” in CCN should be 1/8.

Table 4 shows the SS that we compute for each pair of software systems. Obviously, the SS for the software that applies design patterns is larger than the software that does not apply design patterns. Thus, in the examined systems, we can find that SS can identify the higher-quality software from two functionally equivalent software systems.

4.2.4. RQ4: Is ARCN_n(T) a Good Indicator for Class Importance? Important classes (also called key classes) have a “controlling function” in a piece of software. These important classes usually work together tightly to provide services for many other classes in the system. ARCN_n(T) quantifies the scope that changes in node n will influence in CCN. It is an indirect measurement of the influence power of node n in the change propagation process. Thus, it is estimated that node n with a larger value of ARCN_n(T) should be much more important. There should be a positive correlation between the importance of a class and its ARCN_n(T). In this section, we treat ARCN_n(T) as the importance of class node n, and based on ARCN_n(T) of each node to identify the important classes in software. We also compare ARCN_n(T) with four baseline approaches on key class identification.

(1) Subject Systems. Two Java software systems have been chosen as our subject systems. One is ant, a library and
Figure 7: CCNs for the six pairs of software systems. NO_NAME is the package (anonymous package) where the corresponding class is defined. The source files of these figures can be downloaded from https://www.icloud.com/iclouddrive/0sSM4U7GmxITLpfp0kxcDs#complexity9414162supp2.

command-line tool for automating software build processes. The other is JMeter, an application for testing functional behavior of software and measuring its performance. The two systems have been widely used as benchmark systems for validating the effectiveness of approaches on identifying key classes. Table 5 gives some simple descriptions of the selected two subject systems, including their names and versions, the domain they belong to, the directory we analyze, and URLs to download these systems. We also show some size-related metrics as we have shown in Table 2. The data for ant shown in Table 5 are copied from Table 2.

(2) Baseline Approaches. The four approaches that we used as baseline approaches include $h$-index [37], $a$-index [37], SM_SW+HITS [38], and SM_SO+HITS [38]. These approaches use different versions of software networks at the
class level of granularity, and also use different metrics to quantify class importance. Specifically, these approaches are described briefly as follows:

(i) $h$-index [37] uses a weighted undirected software network at class level to represent classes and their couplings. Its software network is different from our CCN defined in Definition 1. It only considers four kinds of coupling between classes, and does not consider “IMR”, “MCR”, and “LVR”. It neglects the direction of edges, and summarizes the frequencies of different kinds of couplings between a pair of classes as the weight on that edge. Then, it computes the $h$-index of each class in the software network as the importance of classes. After that, it ranks the classes according to their $h$-index in a descending order. It treats the top-ranked classes as the key classes it identifies.

(ii) $a$-index [37] is similar to $h$-index. The only difference is it computes the $a$-index of each class in the software network as the importance of classes.

(iii) SM$_{SW}$+HITS [38] uses a weighted directed software network at class level to represent classes and their couplings. Its software network is also different from our CCN defined in Definition 1. It only considers one relation between class. That is “MCR”. It counts the method call between every pair of classes to assign weights to the edge between the two classes. It counts every occurrence of a call. Then, it applies HITS algorithm to compute the hubiness score of each class as their importance. After that, it ranks the classes according to their hubiness score in a descending order. It treats the top-ranked classes as the key classes it identifies.

(iv) SM$_{SO}$+HITS [38] is similar to SM$_{SW}$+HITS. The only difference is it applies a different counting mechanism of method calls. It counts every occurrence of a call only once.

(3) Metrics. As mentioned above, $ARCN_m(T)$ is also used as a measure for class importance. To compare the effectiveness of $ARCN_m(T)$ with the baseline approaches, we apply three external metrics. That is, *Precision*, *Recall*, and *F-Measure* (i.e., $F_1$ and $F_3$) [39]. To calculate these metrics, we should establish a standard set of key classes for each software system. In this work, we treat the classes explicitly mentioned in the design documentation by the original developers of software as the ground true for key classes [11]. The metrics used are defined as follows [39]:

(i) True Positive ($TP$). Classes identified by the original developers as key classes are also suggested by a specific approach as key classes.

(ii) False Positive ($FP$). Classes identified by a specific approach as key classes are not suggested by the original developers as key classes.

(iii) False Negative ($FN$). Classes identified by the original developers as key classes are not suggested by a specific approach as key classes.

(iv) True Negative ($TN$). Classes which are not identified by the original developers as key classes are also not suggested by a specific approach as key classes.

Then, we can define *Precision*, *Recall*, and *F-Measure* (i.e., $F_1$ and $F_3$) as follows:

\[
\text{Precision} = \frac{TP}{TP + FP} \times 100\%	ag{4}
\]

\[
\text{Recall} = \frac{TP}{TP + FN} \times 100\%	ag{5}
\]

\[
F_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \times 100\% \tag{6}
\]

\[
F_3 = \frac{26 \times \text{Precision} \times \text{Recall}}{25 \times \text{Precision} + \text{Recall}} \times 100\% \tag{7}
\]

Generally, a larger value of *Precision*, *Recall*, and *F-Measure* indicates the approach has a better effectiveness.

(4) Experiment Process and Results Analysis. We follow the steps shown in Figure 1 to parse the source code of the two software systems, extract information, and build CCNs. Figure 8 shows the CCN we build for the subject system JMeter. Enlarging the figure can give you more information about the figure, such as the class that each node denotes, the coupling between classes, and the weight on each edge. As we show in Figure 7, weights on the edges also denote the coupling frequency. The CCN for the subject system ant is shown in Figure 5(c).

In the related work, the top 15% classes in the ranked list of classes are regarded as key classes [37, 38]. Thus, we also examine the top 15% classes. Tables 6 and 7 show the key classes identified by $ARCN_m(T)$ and the baseline approaches for the two systems, respectively. In Tables 6 and 7, the “Key Classes” column lists the key classes in the ground truth. Other columns contain the key classes identified by each approach. We use “✓” to signify the corresponding class is identified by each approach. The last four rows of each table show the metric data for each approach.

### Table 5: Descriptions of the subject software systems.

| Systems | Version | Domain | Directory | KLOC | #P | #C | #M | #A | URL               |
|---------|---------|--------|-----------|------|----|----|----|----|-------------------|
| ant     | 1.6.1   | parsers| src/main  | 81.515 | 67 | 900| 7691| 4167| http://ant.apache.org |
| JMeter  | 2.0.1   | testing| src/core  | 22.701 | 42 | 258| 1956| 824 | http://wiki.apache.org/jmeter |
Note that, [38] only examined a small part of classes as we have examined in both systems by using a dynamic analysis. However, we cannot access the list of classes that [38] examined. Thus, the results for SM_SW+HITS and SM_SO+HITS shown in Tables 6 and 7 are directly copied from [38].

It can be observed from Table 6 that, when applied to ant, $ARCN_n(T)$ obtains a recall of 80%, a precision of 5.90%, an F1 of 10.99%, and an F5 of 53.94%, all better than the other four approaches in the baseline. It indicates that 80% of the key classes in the ground truth of ant can be retrieved by $ARCN_n(T)$. Similarly, we can observe from Table 7 that, when applied to JMeter, $ARCN_n(T)$ obtains a recall of 28.57%, a precision of 10.33%, an F1 of 15.17%, and an F5 of 26.75%, all better than the other four approaches in the baseline. It indicates 28.57% of the key classes in the ground truth of JMeter can be retrieved by $ARCN_n(T)$. Thus, we can conclude that $ARCN_n(T)$ perform best in all the approaches, signifying $ARCN_n(T)$ is indeed a good indicator for class importance.

4.2.5. RQ5: Can SS Be Applied to Large Software Systems? As a feasible metric, SS will be applied to software with different sizes. Thus, we evaluate the scalability of SS by tracking the execution time that is required to compute it. As mentioned in Figure 1, to compute SS, we mainly follow the following three steps:

(i) Parsing the source code of a piece of software to extract structural information at the class level of granularity.

![Figure 8: CCN for JMeter. The source file of the figure can be downloaded from https://www.icloud.com/iclouddrive/0sSM4U7-GmxTLpLpp0kxcDsg#complexity9414162supp_2.](image-url)
Table 6: Comparison of the results obtained by the approaches applied to ant.

| Key Classes         | SM SW+HITS | SM SO+HITS | $h$-index | $a$-index | $ARCN_{jT}$ |
|---------------------|------------|------------|-----------|-----------|-------------|
| Project             | √          | √          | √         | √         |             |
| UnknownElement      | √          | √          | √         | √         |             |
| Task                | √          |            | √         | √         |             |
| Main                | √          | √          |           |           |             |
| IntrospectionHelper |            |            |           |           |             |
| ProjectHelper       |            |            |           |           |             |
| RuntimeConfigurable |            |            |           |           |             |
| Target              |            |            |           |           |             |
| ElementHandler      |            |            |           |           |             |
| TaskContainer       |            |            |           |           |             |
| Recall              | 30%        | 30%        | 30%       | 20%       | 80%         |
| Precision           | 5%         | 5%         | 2.22%     | 1.48%     | 5.90%       |
| F1                  | 8.57%      | 8.57%      | 4.10%     | 2.76%     | 10.99%      |
| F5                  | 25.16%     | 25.16%     | 20.19%    | 13.50%    | 53.94%      |

Table 7: Comparison of the results obtained by the approaches applied to JMeter.

| Key Classes         | SM SW+HITS | SM SO+HITS | $h$-index | $a$-index | $ARCN_{jT}$ |
|---------------------|------------|------------|-----------|-----------|-------------|
| AbstractAction      |            | √          |           |           |             |
| JMeterEngine        |            |            |           |           |             |
| JMeterTreeModel     |            |            |           |           |             |
| JMeterThread        |            |            |           |           |             |
| JMeterGUIComponent  |            |            |           |           |             |
| PreCompiler         |            |            |           |           |             |
| Sampler             |            |            |           |           |             |
| SampleResult        |            |            |           |           |             |
| TestCompiler        |            |            |           |           |             |
| TestElement         |            |            |           |           |             |
| TestListener        |            |            |           |           |             |
| TestPlan            |            |            |           |           |             |
| TestPlanGui         |            |            |           |           |             |
| ThreadGroup         | √          | √          |           |           |             |
| Recall              | 7.14%      | 7.14%      | 21.43%    | 14.29%    | 28.57%      |
| Precision           | 1.4%       | 1.4%       | 7.75%     | 5.17%     | 10.33%      |
| F1                  | 2.34%      | 2.34%      | 11.38%    | 7.59%     | 15.17%      |
| F5                  | 6.17%      | 6.17%      | 20.07%    | 13.38%    | 26.75%      |

(ii) Representing the extracted structural information as CCN.

(iii) Applying Algorithm 1 to compute the value of related metrics.

Table 8 shows the CPU time required for each step when applied to subject software systems listed in Table 2. Obviously, it can be observed that there is a strong positive correlation between the KLOC of the software system and the CPU time for the (i) step, i.e., a larger KLOC of a piece of software indicates the more execution time that the (i) step takes. The CPU time for the (ii) and (iii) steps seems related to the number of nodes in CCNs. We can observe that although jdk and tomcat are very large in size (see Table 2), the execution time to compute $SS$ from scratch is less than 4.5 minutes. Thus, the answer to RQ5 is that $SS$ has a good scalability, and it can be applied to large software systems with an acceptable CPU time cost.

4.3. Threats to Validity. In the evaluation, some factors may affect the validity of our conclusions, i.e.,

Table 8: CPU time required for each step ($T = 10,000$ and $U = 1$ in Algorithm 1).

| Software | jdk | tomcat | ant | jfreechart | struts | mybatis |
|----------|-----|--------|-----|------------|-------|--------|
| i        | 234s| 53s    | 30s | 46s        | 8s    | 5s     |
| ii       | 3s  | 2s     | 2s  | 1s         | 1s    | 1s     |
| iii      | 10s | 6s     | 5s  | 4s         | 4s    | 4s     |
(i) A threat to the internal validity of our study is related to ground truth that we used to answer RQ4. Key classes in the ground truth are provided by the original developers of the software systems. However, these developers may be biased towards some parts of the software that they are familiar with. It will affect the values of Recall, Precision, F1, and F5. This threat has been partially mitigated by the fact that we use the benchmark software systems in the literature.

(ii) A threat to the external validity of our study is related to the generalization of our conclusions to other software systems. On one hand, our evaluation is performed on a small set of software systems when compared with the whole set of software in the world. On the other hand, our evaluation is performed on software coded in Java. There is no software coded in non-Java language. This threat has been partially mitigated by the fact that our subject software systems are chosen from different domains and have different sizes. Furthermore, our approach can be generalized to software systems coded in non-Java languages if they can be transformed to a CCN representation.

5. Conclusions and Future Work

In this paper, we proposed a novel approach to characterize software stability with the aim to control software maintenance costs. Our approach is based on a novel software network representation of software structure (i.e., class coupling network) and a simulation algorithm for change propagation. A novel metric, SS (software stability), is developed to measure software stability.

We evaluated our SS metric theoretically using the widely accepted Weyuker’s criteria, and empirically using a set of open source Java programs. The theoretical results show that our SS metric satisfies most of Weyuker’s properties only with two exceptions, and the empirical results show that our SS metric is an effective indicator for software quality improvement and class importance. Empirical results also show that our approach has the ability to be applied to large software systems.

Our future work includes (i) collecting a large set of benchmark Java software systems to validate the effectiveness of our approach and (ii) extending the current work to analyze software coded in non-Java languages.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (nos. 2017YFB1400602 and 2016YFB0800400), the National Natural Science Foundation of China (nos. 61572371 and 61702378), and the Commonwealth Project of Science and Technology Department of Zhejiang Province (no. LGF19F020007).

References

[1] R. S. Pressman, Software Engineering: A Practitioner’s Approach, McGraw-Hill, New York, NY, USA, 1992.
[2] S. Yau and J. Collofello, “Design stability measures for software maintenance,” IEEE Transactions on Software Engineering, vol. SE-11, no. 9, pp. 849–856, 1985.
[3] S. Yau and J. Collofello, “Some stability measures for software maintenance,” IEEE Transactions on Software Engineering, vol. SE-6, no. 6, pp. 545–552, 1980.
[4] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, International Thomson Computer Press, London, UK, 2nd edition, 1996.
[5] D. Darcy, C. Kemerer, S. Slaughter, and J. Tomayko, “The structural complexity of software an experimental test,” IEEE Transactions on Software Engineering, vol. 31, no. 11, pp. 982–995, 2005.
[6] C. R. Myers, “Software systems as complex networks: structure, function, and evolvability of software collaboration graphs,” Physical Review E: Statistical, Nonlinear, and Soft Matter Physics, vol. 68, no. 4, Article ID 046116, 2003.
[7] A. Potanin, J. Noble, M. Frean, and R. Biddle, “Scale-free geometry in OO programs,” Communications of the ACM, vol. 48, no. 5, pp. 99–103, 2005.
[8] G. Concass, M. Marchesi, S. Pinna, and N. Serra, “Power-laws in a large object-oriented software system,” IEEE Transactions on Software Engineering, vol. 33, no. 10, pp. 687–708, 2007.
[9] W. Pan, B. Li, Y. Ma, and J. Liu, “Multi-granularity evolution analysis of software using complex network theory,” Journal of Systems Science and Complexity, vol. 24, no. 6, pp. 1068–1082, 2011.
[10] W. Pan, B. Li, Y. Ma, J. Liu, and Y. Qin, “Class structure reactoring of object-oriented softwares using community detection in dependency networks,” Frontiers of Computer Science in China, vol. 3, no. 3, pp. 396–404, 2009.
[11] W. Pan, B. Song, K. Li, and K. Zhang, “Identifying key classes in object-oriented software using generalized k-core decomposition,” Future Generation Computer Systems, vol. 81, pp. 188–202, 2018.
[12] W.-F. Pan, B. Li, Y.-T. Ma, and J. Liu, “Test case prioritization based on complex software networks,” Acta Electronica Sinica, vol. 40, no. 12, pp. 2456–2465, 2012.
[13] E. J. Weyuker, “Evaluating software complexity measures,” Institute of Electrical and Electronics Engineers. Transactions on Software Engineering, vol. 14, no. 9, pp. 1357–1365, 1988.
[14] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, “Change impact identification in object-oriented software maintenance,” in Proceedings of the IEEE International Conference on Software Maintenance (ICSM ’94), pp. 202–211, IEEE, Victoria, Canada, 1994.
[15] W. Li, L. Etzkorn, C. Davis, and J. Talburt, “An empirical study of object-oriented system evolution,” Information and Software Technology, vol. 42, no. 6, pp. 373–381, 2000.
[16] D. Grosser, H. A. Sahraoui, and P. Valtchev, “An analogy-based approach for predicting design stability of Java classes,” in Proceedings of the 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (Metric ’03), pp. 252–262, Sydney, Australia, 2003.

[17] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu, “Using history information to improve design flaws detection,” in Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR ’04), pp. 223–232, Tampere, Finland, 2004.

[18] A. MacCormack, J. Rusnak, and C. Y. Baldwin, “Exploring the structure of complex software designs: an empirical study of open source and proprietary code,” Management Science, vol. 52, no. 7, pp. 1015–1030, 2006.

[19] I. Shaik, W. Abdelmoez, R. Gunnalan et al., “Using change propagation probabilities to assess quality attributes of software architectures,” in Proceedings of the IEEE International Conference on Computer Systems and Applications (ICCSA ’06), pp. 704–711, Dubai, UAE, March 2006.

[20] J. Liu, J. Lü, K. He, B. Li, and C. K. Tse, “Characterizing the structural quality of general complex software networks via statistical propagation dynamics,” International Journal of Bifurcation and Chaos, vol. 18, no. 02, pp. 605–613, 2008.

[21] L. Li, G. Qian, and L. Zhang, “Evaluation of software change propagation using simulation,” in Proceedings of the WRI World Congress on Software Engineering (WCSE ’09), pp. 28–33, Xiamen, China, May 2009.

[22] L. Zhang, G.-Q. Qian, and L. Li, “Software stability analysis based on change impact simulation,” Chinese Journal of Computers, vol. 33, no. 3, pp. 440–451, 2010.

[23] W. Pan, “Quantifying the stability of software systems via simulation in dependency networks,” International Journal of Computer and Systems Engineering, vol. 5, no. 12, pp. 1599–1606, 2011.

[24] M. Alshayeb, M. Naji, M. Elish, and J. Al-Ghamdi, “Towards measuring object-oriented class stability,” IET Software, vol. 5, no. 4, pp. 415–424, 2011.

[25] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avergiou, “The effect of GoF design patterns on stability: a case study,” IEEE Transactions on Software Engineering, vol. 41, no. 8, pp. 781–802, 2015.

[26] L. Badri, M. Badri, and N. Joly, “Towards a change impact analysis model for java programs: an empirical evaluation,” Journal of Software, vol. 10, no. 4, pp. 441–453, 2015.

[27] A. Abuhassan and M. Alshayeb, “A metrics suite for UML model stability,” Software and Systems Modeling, vol. 18, no. 1, pp. 557–583, 2019.

[28] W. Pan and C. Chai, “Measuring software stability based on complex networks in software,” in Cluster Computing, pp. 1–10, Springer, 2017.

[29] L. C. Briand, J. W. Daly, and J. K. Wüst, “A unified framework for coupling measurement in object-oriented systems,” IEEE Transactions on Software Engineering, vol. 25, no. 1, pp. 91–121, 1999.

[30] S. Misra, I. Akman, and R. Colomo-Palacios, “Framework for evaluation and validation of software complexity measures,” IET Software, vol. 6, no. 4, pp. 323–334, 2012.

[31] J. C. Cherniavsky and C. H. Smith, “On Weyuker axioms for software complexity measures,” IEEE Transaction on Software Engineering, vol. 17, no. 6, pp. 636–638, 1991.

[32] Gursaran and G. Roy, “On the applicability of Weyuker Property 9 to object-oriented structural inheritance complexity metrics,” IEEE Transactions on Software Engineering, vol. 27, no. 4, pp. 381–384, 2001.

[33] W. Harrison, “An entropy-based measure of software complexity,” IEEE Transactions on Software Engineering, vol. 18, no. 11, pp. 1025–1029, 1992.

[34] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476–493, 1994.

[35] H. Zhang, Y. Li, and H. B. Tan, “Measuring design complexity of semantic web ontologies,” The Journal of Systems and Software, vol. 83, no. 5, pp. 803–814, 2010.

[36] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, “Design pattern detection using similarity scoring,” IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 896–909, 2006.

[37] M. S. Wang, H. M. Lu, Y. M. Zhou, and B. W. Xu, “Identifying key classes using h-index and its variants,” Journal of Frontiers of Computer Science and Technology, vol. 5, no. 10, pp. 891–903, 2011.

[38] A. Zaidman and S. Demeyer, “Automatic identification of key classes in a software system using webmining techniques,” Journal of Software Maintenance and Evolution: Research and Practice, vol. 20, no. 6, pp. 387–417, 2008.

[39] W. Pan, B. Li, B. Jiang, and K. Liu, “Recode: software package refactoring via community detection in bipartite software networks,” Advances in Complex Systems, vol. 17, no. 7&8, Article ID 1450006, 2014.
