A Hybrid Set of Complexity Metrics for Large-Scale Object-Oriented Software Systems

Article *in* Journal of Computer Science and Technology · November 2010

DOI: 10.1007/s11390-010-9398-x · Source: DBLP

5 authors, including:

Yutao Ma
Wuhan University
96 PUBLICATIONS **1,368** CITATIONS

SEE PROFILE

Bing Li
Wuhan University
133 PUBLICATIONS **1,573** CITATIONS

SEE PROFILE

Jing Liu
East China Normal University
104 PUBLICATIONS **1,182** CITATIONS

SEE PROFILE

Ma YT, He KQ, Li B *et al.* A hybrid set of complexity metrics for large-scale object-oriented software systems. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 25(6): 1184–1201 Nov. 2010. DOI 10.1007/s11390-010-1094-3

A Hybrid Set of Complexity Metrics for Large-Scale Object-Oriented Software Systems*

Yu-Tao Ma[1,2,3] (马于涛), *Member*, *CCF*, *ACM*, Ke-Qing He[1,2] (何克清), *Senior Member*, *CCF*, *IEEE*
Bing Li[1,2] (李　兵), *Senior Member*, *CCF*, Jing Liu[1,2] (刘　婧), and Xiao-Yan Zhou[1] (周晓燕)

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

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

[3] *Institute of Electronic System Engineering, Beijing 100141, China*

E-mail: ytma@mail.whu.edu.cn; {hekeqing, jingliu}@sklse.org; bingli@whu.edu.cn; zhou0420@tom.com

Received July 15, 2009; revised February 2, 2010.

Abstract Large-scale object-oriented (OO) software systems have recently been found to share global network characteristics such as *small world* and *scale free*, which go beyond the scope of traditional software measurement and assessment methodologies. To measure the complexity at various levels of granularity, namely graph, class (and object) and source code, we propose a hierarchical set of metrics in terms of coupling and cohesion — the most important characteristics of software, and analyze a sample of 12 open-source OO software systems to empirically validate the set. Experimental results of the correlations between cross-level metrics indicate that the graph measures of our set complement traditional software metrics well from the viewpoint of *network thinking*, and provide more effective information about fault-prone classes in practice.

Keywords complexity metrics, quality analysis and evaluation, object-oriented programming, reverse engineering, complex networks

1 Introduction

It has been widely recognized that object-oriented programming (OOP) is one of the most successful techniques to design and implement applications and computer programs with the support of programming languages such as C++ and Java. Nowadays, as diversified application requirements continue to grow rapidly the Internet-based OO software systems are becoming more and more complex, fragile, and difficult to control. "You can't control what you can't measure."[1] To quantify and evaluate software quality (or some properties of a piece of software), software metrics deal with the measurement of a software product (or artifact) and the process by which it is developed[2], and provide a quantitative evaluation basis for modern OO software development.

Until now, many OO software metrics have been proposed and applied to practical software projects. It is well known that two of the widely accepted metrics sets are the Chidamber and Kemerer's Metrics Suite (CK)[3] and the metrics for object-oriented design set (MOOD)[4]. They have previously been shown to be good predictors of OO design complexity as well as software defects, and successful experiences from OO software development practices have proved their validity for years[5]. Hence, OO software metrics play an important role in ensuring the desired quality of software systems.

For a metrics suite such as CK and MOOD, it only offers a special assessment of some features of an OO software system[6]. For example, CK appears to give a class-level evaluation with respect to inheritance, coupling, cohesion and complexity[7], while MOOD tends to provide a system-wide assessment according to inheritance, coupling, polymorphism and encapsulation[8]. On the other hand, previous studies[9–12] imply that the empirical effects of these metrics suites on large-scale OO software systems are limited on account of design complexity, programming languages and implementation tools. So, designing an

Regular Paper

*An early version has been published in the proceedings of the 6th IEEE International Conference on Computer and Information Technology (CIT 2006).

Supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2007CB310800, the National Natural Science Foundation of China under Grant Nos. 60873083 and 60803025, the Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20090141120022, the Natural Science Foundation of Hubei Province of China under Grant Nos. 2008ABA379 and 2008CDB351, and the Fundamental Research Funds for the Central Universities of China under Grant No. 6082005.

appropriate metrics suite for complex OO software systems based on the Internet is more challenging.

With the popularity of the Unified Modeling Language (UML)[13] and the Model Driven Architecture (MDA)[14], the focus of modern software development shifts from implementation (coding) to design (modeling). A series of UML diagrams such as class diagram and collaboration diagram have been employed to model the solution to given requirements of a system with large numbers of interacting classes and objects. Among these diagrams of a UML model, class diagram is an important one, which describes classes (composed of attributes and methods) and the relationships between them. Owing to the independence of specific platforms and programming languages, it is easier for the metrics for UML class diagrams to represent the nature of OO software design.

It is generally accepted that the quality of class diagrams created at high-level design stage will have a significant impact on the system which is ultimately implemented[15]. Since 2000, a few new metrics have gradually been put forward to measure structural complexity[16-17], maintainability[18], understandability and modifiability[19] of UML class diagrams. Despite the perceived effectiveness for some special design features of OO software systems, they still have some shortcomings, e.g., no consensus has yet been reached on measuring structural complexity of UML class diagrams[20]. Therefore, it is difficult to make a sensible choice when software engineers evaluate the quality of their design artifacts.

Overly concentrating on the details of OOP hampers the effort to provide an insight into some important properties of OO software systems at a high level of abstraction[21] (such as graph or network model). Obviously, a UML class diagram can be mapped directly to a graph, where nodes represent classes and (directed) edges denote various types of relationships between classes. Since graphs conceal the details of OO context from developers, they are valuable in many ways to understand (and even quantify) the underlying properties of large-scale OO software[22].

Recently, researchers analyzed huge amounts of OO software systems described by means of class-level dependency (or collaboration) graphs, and found that most of them possess global statistical features[23-25] such as *small world* and *scale free*. The unexpected findings raised empirical and theoretical questions about those traditional metrics under the context of OOP. Graph theory has long been used in several fields of software engineering, but very little research concerning this interdisciplinary field has so far been conducted[26-27]. Even so, we argue that *network thinking* for large-scale OO software systems based on the Internet should be encouraged so as to supply a novel insight into the above-mentioned problems that go beyond the limit of OO context for software developers.

By understanding the fundamental idea that views OO software in terms of network or graph, we can recognize that graph-level measures for behavioral and structural properties of software would be a necessary complement to traditional software metrics[6]. Furthermore, as indicators for defect-prone classes, these measures may be more useful than OO software metrics when system-wide interactions among elements are taken into account. Then, the contributions of this paper are described as follows.

1) We proposed a hybrid set of complexity metrics, which has a hierarchical structure with 3 layers, namely graph, class and code. It can provide enough metrics for software engineers to evaluate OO software complexity at various levels and from different perspectives.

2) Based on the set, we performed experiments with open-source OO software systems, and discovered some significant correlations between graph-level measures and class-level metrics. The results will help developers to identify high-level software structural defects more effectively.

The rest of this paper is organized as follows. Section 2 introduces the background and related work. Section 3 presents detailedly the hybrid set which is composed of 3-layered software measurements. Section 4 offers a case study of applying the set to measuring a medium-sized Java system. Section 5 continues to analyze some significant correlations between graph-level measures and class-level metrics to identify defect-prone classes. In the end, Section 6 concludes this paper.

2 Background and Related Work

2.1 Complex Networks

In the context of graph theory, Wikipedia defines a *complex network* as "a network (graph) with non-trivial topological features — features that do not occur in simple networks such as lattices or random graphs."[28] Such features often refer to a heavy tail in the degree distribution, a high clustering coefficient, community structure, etc. However, many mathematical models of networks that have been studied in the past, such as random graphs, do not exhibit these features.

Ten years ago, *small world* effect[29] and *scale free* property[30] were discovered successively in real-world systems such as the Internet, World Wide Web (WWW) and protein interaction networks. The former is known for short path length and high clustering (this phenomenon is also popularly known as *six degrees of separation*[31]), while the latter is famous for power-law degree distribution. Now, they have been deemed as

the most important statistical characteristics of complex networks that are different from regular graphs such as lattice and random graphs.

The epoch-making finding attracts attention of many famous scientists from diverse domains, which contributes significantly to the prosperous study of complex networks. Since the early 2000's, these nontrivial observations and increasingly popular researches on real-world networks have gradually spawned a new research field called *complex networks*, whose purpose is to seek to discover general principles, algorithms and tools that govern real-world network behaviors[32].

2.2 Software Systems as Complex Networks

Software is built up out of large amounts of interacting components at various levels of granularity, e.g., package, class and method. The dependencies and interactions between those pieces can be used to define network models or graphs that form a skeletal description of software systems[21]. A few physicists first analyzed class-level dependency (or collaboration) graphs derived from source code by using some statistical approaches, and found that most of them share global statistical features[33-35] such as *small world* and *scale free*. So, Myers argued that software systems represent another important class of complex networks[33].

The interesting result not only validated the universality of *small world* effect and *scale free* property, but also attracted scholars from the software engineering community to participate actively in the research. OO software systems present novel perspectives to the study of complex networks[33], so the interest has been increasing in software's geometric structure, system dynamics and evolutionary mechanisms across the disciplines from physics to computer science.

A number of *software networks*, which were defined as dependency, collaboration or coupling graph (or network) in previous literature [33, 35-36], were generated with the help of reverse engineering tools. A comprehensive analysis on structural features of these networks showed that most of them exhibit approximate power-law degree distribution and high clustering across different levels of granularity (viz., package, class and method)[23-25,37-44], which proved that the topological structure of OO software based on the Internet possesses distinct characteristics of complex networks. Based on the ample empirical evidence, *software as a complex network* (SaaCN) has gradually been recognized within the software engineering community[24-25,45-46].

2.3 Metrics for Structural Properties

It was known long ago that the structure of software has a significant impact on its functions and quality attributes[47]. Recently, in order to develop better and more robust large-scale software, some researchers began to investigate the structural properties of OO software systems in terms of complex network. Up to now, they have proposed several graph-level metrics to quantify the specific properties such as complexity and stability independent of OOP context. In general, these metrics are useful to evaluate the quality of software design, and help developers to identify problematic dependency structures and defect-prone nodes in software networks.

According to structure entropy, Ma *et al.* proposed a qualitative method[21] to measure structural complexity of software systems. Their experiments revealed that the heterogeneous topology with *scale free* property has a strong positive correlation with structural complexity. From the perspective of social network analysis (SNA) related to graph theory[48], Zhao *et al.* put forward a suite of metrics[49] for static structural complexity, which overcomes the limitations of traditional OO software metrics. In addition to complexity metrics, complex network-based measures for stability and evolvability (associated with complexity) have also been presented in recent literature [50-52] to facilitate the maintainability and continuous refactoring of OO software. However, an appropriate metrics set that can comprehensively evaluate a large-scale OO software system's complexity at various levels of granularity is still under discussion.

Other work about defect identification has been carried out in [27, 36, 53]. Liu *et al.*[36] defined the weight of a node in software networks in terms of CK's WMC (the Weighted Methods per Class) metric, and found that there is a clear positive correlation between the weight and the out-degree of nodes in four open-source software systems. Further experiments indicated that the similar correlation also exists between CK's LCOM (the Lack of Cohesion in Methods) metric and out-degree. So, they suggested that out-degree may be an indicator for defect-prone dependency structures. Zimmermann *et al.*[27] proposed a suite of SNA-based network measures on dependency graphs to identify central program units that are more likely to face defects. Empirical data showed that these measures are able not only to identify critical binaries of a complex software system that are missed by traditional complexity metrics, but also to predict the number of defects. Based on Zimmermann and Nagappan's work, Tosun *et al.*[53] conducted additional experiments on public data from open-source software systems to validate their results[27]. Experimental results demonstrated that network measures are important predicators of defective modules for large and complex systems rather

than small-scale projects. Hence, the multi-granularity defect analysis method for ever-increasing complexities of the Internet-based OO software systems is an urgent need for software engineering research.

3 Hybrid Set of Complexity Metrics

3.1 Summary of Traditional Metrics

Early complexity metrics for software programs such as SLOC (the Source Lines of Code) and McCabe Cyclomatic Complexity (MCC) metric[54] have been proved to be inadequate for modern OO software. It is well known that the main mechanisms of OO paradigm, namely inheritance, encapsulation and polymorphism, are the key to achieving efficient reuse and easier maintainability. Considering the quantification of these particular features of OOP, CK and MOOD initiated a new era of OO software measurement. However, there are still some acknowledged limitations of the above-mentioned metrics suits.

On one hand, obviously, it is difficult to measure the macroscopic properties independent of OOP (such as the shortest path length) in complex software systems with OO software metrics[6] or even class diagram-scope metrics[15,20]. For instance, there is a lack of metrics for the influence of a class or a relationship on the whole system. That is to say, if a random class fails, developers will need a metric to evaluate its impact on the global structure of the system.

On the other hand, the absence of graph-level metrics may affect a developer's decisions on choosing an appropriate metric or measure in the practical software development. Hence, in practice developers have to go through OO metrics all in order to decide which metric

Table 1. Summary of Traditional Software Metrics

Metric	Category	Description	CK	MOOD	Genero's Metrics
FP	Size	The number of function points			
SLOC	Size	The number of code lines			
NC	Size	The number of classes			NC
DIT	Inheritance	The depth of a class in the inheritance hierarchy	DIT		
NoC	Inheritance	The number of the children of a class	NOC		
NGenH	Inheritance	The total number of generalization hierarchies within a class diagram			NGenH
MaxDIT	Inheritance	The maximum DIT value obtained for each class within a class diagram			MaxDIT
MIF	Inheritance	The number of inherited methods as a proportion of the total number of methods		MIF	
MAF	Inheritance	The number of inherited attributes as a proportion of the total number of attributes		AIF	
NID	Coupling	The number of classes that depend on or associate with a class	CBO	CF	NAssocC, NDepIN
NOD	Coupling	The number of classes on/with which a class depends/associates			NAssocC, NDepOUT
RFC	Collaboration	The number of methods that can be executed in response to a message to the class	RFC		
NAgg	Aggregation	The total number of aggregation relationships within a class diagram			NAgg
NAggH	Aggregation	The total number of aggregation hierarchies within a class diagram			NAggH
HAgg	Aggregation	The length of the longest path from the class to leaves within an aggregation hierarchy			HAgg
MaxHAgg	Aggregation	The maximum HAgg value obtained for each class within a class diagram			MaxHAgg
MAgg	Aggregation	The number of direct "whole" classes within an aggregation hierarchy			MAgg
LCOM	Cohesion	The degree of similarity of methods by data inputs or the instance variables of a class	LCOM		
CC	Complexity	The complexity of a class	WMC		NODP, NP, NW
CCD	Complexity	The complexity of a UML class diagram			NAssoc, NDep, NAgg, NGen
ED	Encapsulation	The degree of information hiding		MHF AHF	
PD	Polymorphism	A measure of polymorphism potential		PF	
SC	Complexity	A measure of structural complexity of an object-oriented system			?
ID	Complexity	The influence of a class or a relationship on the whole system			?
⋮	⋮	Metrics for macroscopic properties of large-scale software systems			?

is most suitable for their project and adjust the metrics set to get the right benefits according to the size and complexity of the project.

These problems lead to an embarrassing situation where we do not have adequate software metrics to measure the overall complexity of a large and complex software system. For example, as we know, Windows XP has a more complex structure than Windows 98, but we cannot yet say it is x times more complex, where x is some number, because the measurement of their structural complexity still remains challengeable for software engineers[55].

Then, in Table 1 we present the summary of traditional software metrics to show the general properties they measure and implicit limitations. The details of CK, MOOD and Genero's metrics please refer to [3, 4, 56], respectively.

It is apparent from Table 1 that Genero's metrics for UML class diagrams have overcome some limitations of CK and MOOD by virtue of abstraction. For example, a few novel metrics were designed to measure other kinds of relationships between classes such as aggregation; moreover, the class diagram-scope metrics such as NGenH were also proposed to assess some external quality attributes, e.g., the maintainability related to complexity. Although they can be applied not only to a single class, but also to a class diagram, little emphasis has been put on measuring quality aspects of a class diagram as a whole[15]. The study of complex networks has led to an ongoing development and refinement of network models and graph theoretical analysis techniques with which to characterize and understand complexity, so we argue that graph-level metrics may provide an insight into the complexity of software structures derived from source code or class diagrams.

3.2 Generic Graph-Level Metrics

From the perspectives of complexity science and software engineering, the following subsections will introduce the basic graph theoretical measures (viz. the so-called global topological measures of complex networks) that characterize the topology of a software network at multiple scales.

3.2.1 Degree Distribution

In the context of graph theory, the degree of a node in a graph is defined as the number of edges the node has against other nodes. For a directed graph, its nodes have two different degrees, namely in-degree and out-degree. Degree distributions, summarizing the connectivity of each node in a graph, indicate the probability of finding a node with a specified degree k[33]. Sometimes, a cumulative degree distribution[35], the fraction of nodes with degree greater than or equal to k, represents another form of the same information.

Now, many software networks derived from source code have been found to possess a *scale free* degree distribution, implying that $P(k)$ approximately obeys a power law over an extended range of degrees. By and large, the existence of heavy-tailed out-degree distributions suggests a broad spectrum of complexity, and the existence of heavy-tailed in-degree distributions implies a broad spectrum of reuse[33]. Hence, the metric ($P(k)$) may be used not only to characterize the connectivity of a software network, but also to estimate its overall structural complexity in terms of structure entropy[21].

3.2.2 Average Shortest Path Length

As an important concept of network topology, average shortest path length is also known as average path length in graph theory. It is defined as the average number of steps along the shortest paths between every pair of nodes through a network. If there is no path connecting the two nodes, their distance is conventionally defined as an infinite. Physicists often make use of it in practice to measure the efficiency of information or mass transport on a network.

Many software systems have recently been found that their average shortest path lengths are short and change proportionally to $\log N$[35], where N is the number of nodes in the network. For example, the average shortest path length of JDK1.5 is around 6[35], implying the so-called *six degrees of separation* phenomenon[31] which has been found in social networks, the Internet, WWW, etc. In general, the metric (d) has remarkable impacts on the efficiency of messaging (or information transfer) among classes as well as on the overall response capability of OO software.

3.2.3 Clustering Coefficient

In a network clustering indicates the tendency for a node's neighbors to cluster themselves. So, clustering coefficient is used to assess the degree to which nodes tend to cluster together. For a single node in a network, the metric quantifies how close its neighbors are to being a clique (complete graph)[29]; for the whole network, the metric is the average of clustering coefficients for each node.

Besides short average shortest path length, high (average) clustering coefficient is also a significant characteristic of small-world networks. The average clustering coefficients of many software networks are much higher than those of random graphs constructed on the same node set, possibly suggesting a high degree of cohesion among components of these systems. Furthermore, Myers observed that the clustering coefficients of nodes

with k edges follow the scaling law $C(k) \sim k^{-1}$, which implies a hierarchical organization of modularity[57] in experimental software systems[33]. Hence, the metric (C) could be employed to characterize modularity and to estimate the cohesion of community (or modular) structures within large-scale software systems.

3.2.4 Betweenness Centrality

Betweenness is a measure of the centrality of a node in a network, which is normally calculated as the fraction of shortest paths between node pairs that pass through the node of interest[58]. It determines the "traffic" passing through the chosen node along all shortest paths between all node pairs in a network. So, a node that occurs on many shortest paths between other nodes has higher betweenness value than those that do not. Similarly, one can define the betweenness of an edge[58]. Both measures of betweenness centrality give some sense of the relative linking and/or traffic-directing capability of a node or an edge in a software network.

Betweenness reveals the importance of a node or an edge in the overall connectivity of a software network. So, the metric (B) enables us to analyze the significance of a node or an edge through the whole network, which may facilitate our understandings on the robustness for structural optimization of a given OO software system.

3.2.5 Degree Correlations

The correlations between the in- and out- degrees of different nodes in a network have been found to play an important role in many structural and dynamical network properties[59]. They measure the linear correlation of degrees over all edges of a graph, and reflect the tendency of nodes with similar degrees to be connected to one another.

An interesting observation is that essentially all social networks measured appear to be assortative, suggesting that nodes with high degree tend to connect with other similar nodes, but other types of networks (such as information network and technological network) appear to be disassortative[33,58-59]. Hence, these correlations not only can help us to measure the collaboration among classes with different degrees, but also provide a useful way to analyze the hierarchy of software functions.

3.2.6 Summary

Although new measures such as the resilience of complex networks were proposed one after another in recent years, the above-mentioned metrics have been recognized as the basis to study complex networks. As the generic graph-level metrics for software networks, they are capable of expressing the most relevant topological features of large-scale OO software systems based on the Internet. Then, a brief review of them is presented in Table 2.

3.3 Integration of Different Levels of Metrics

3.3.1 Hierarchy of the Metrics Set

A software system is composed of large numbers of software components. Considering their diverse levels of granularity, e.g., procedure, class and package, software complexity should be analyzed from the viewpoint of *system thinking* with different dimensions or aspects, mainly including source code, OO context and network characteristics.

Hence, the metrics of our hybrid set could be classified into three categories (see Fig.1): statistical metric (at the graph level regardless of OO context), system & component metric (at the OO context level) and code metric (at the implementation level of program blocks written in programming languages).

1) By virtue of network analysis techniques, statistical metrics are mostly used to measure the global features of a large-scale OO software system independent of specific design techniques so as to provide an overview or a general picture of the system for software engineers.

Table 2. Brief Review of Generic Graph-Level Metrics

Metric	Symbol	Implications for Software Engineering
Number of nodes	N	
Number of edges	L	
Average degree	$\langle k \rangle$	The metric for the average coupling of a software system
Average shortest path length	d	The metric for the efficiency of information transport, communication cost among classes and the overall response capability of a software system
Degree distribution	$P(k)$	The metric for the overall complexity of a software system
Clustering coefficient	C	The metric for the degree of interior cohesion of a software module
Betweenness	$B(v)/B(e)$	The metric for the impact of a class or a relationship on the whole system
Correlation coefficient	$corr(k_i^{\text{in/out}}, k_j^{\text{in/out}})$	The metric for the hierarchy of software functions and the collaboration among classes with different degrees
	$corr(k_i, C_i)$	The metric for a hierarchical organization of modularity in a software system

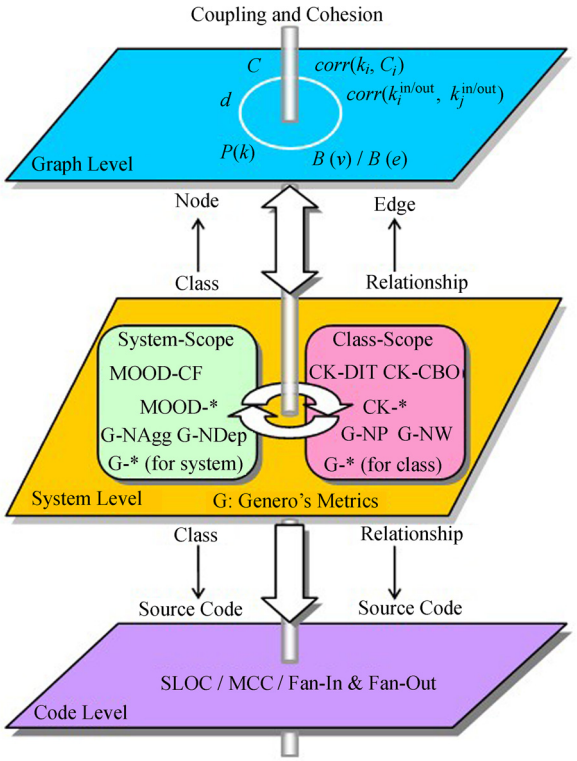


Fig.1. Hierarchy of the metrics set.

2) In the context of OOP, system & component metrics are mainly employed to quantify the specific properties of a system as well as a class in terms of the main mechanisms of OOP so as to evaluate the quality of OO software design.

3) Code metrics represent the measurement of code complexity of program blocks such as class or module, which is directly related to the maintainability and testability of source code, so they can offer detailed guidance for programmers to improve code quality.

3.3.2 *Different Levels of Metrics*

In Table 3, all metrics at different levels are listed according to the properties that are measured. Note that the symbol (x, y) represents a composite metric that consists of two single metrics x and y.

As shown in Fig.1, statistical metrics are positioned towards the top layer of the hierarchy. Since an OO software system can be treated as a directed network irrespective of OOP context, they measure global structural features of a software network in terms of coupling (degree) and cohesion (clustering coefficient), which are the most important characteristics of software. Thus, these graph-level metrics can provide the measurement of properties that are necessary but not sufficient for software networks derived from source code or UML class diagrams. In addition to complexity, size, etc., other important properties that go beyond the reach of traditional metrics (e.g., modularity and efficiency of information transport) are also measured at this level.

OO software metrics have long been the focus of software engineering. The previous work lays solid foundation for the design of system metrics under the context of OOP. As we know, software structural complexity stems from complex relationships among a very large number of software entities[21]. Compared with traditional OO metrics suites such as CK and MOOD, the metrics for UML class diagrams take a wider range of relationships besides inheritance and dependency into account, facilitating a comprehensive assessment of system-wide structural complexity. Combining the

Table 3. Metrics of the Set

Purpose	Statistical Metric	System & Component Metric		Code Metric
		System-Scope	Class-Scope	
Coupling	k, $\langle k \rangle$	CF	CBO (NAssocC, NDepIN, NDepOUT)	Fan-in & Fan-out
Cohesion	C		LCOM	
Information hiding		(MHF, AHF)		
Polymorphism		PF		
Inheritance		(MIF, AIF) (NGen, NGenH, MaxDIT)	DIT, NOC	
Abstractness		A: the ratio of abstract classes		
Size	N	NC	(NODP, NP, NW)	SLOC
Efficiency of information transport	d			
Complexity	$P(k)$	(NAssoc, NDep, NAgg, NGen)	WMC, RFC	MCC
Significance Relationship	$B(e)$			
Significance Component	$(B(v), k)$			
Hierarchical modularity	$corr(k_i, C_i)$			
Function hierarchy	$corr(k_i^{\text{in/out}}, k_j^{\text{in/out}})$			$corr(i_{\text{Fan-in/out}}, j_{\text{Fan-in/out}})$

primary advantages of CK, MOOD and Genero's metrics, two kinds of metrics at the level of OO context, namely system-scope metric and class-scope metric, are defined to measure the main features of OOP and the related external quality attributes.

All design artifacts will be implemented by modern programming languages that support OOP in the end. Hence, code-level complexity metrics such as SLOC and MCC are applied to the modules, methods and classes within a software program to monitor code quality. Moreover, they are also used to provide a visual estimation of effort or software cost for programmers.

Software metrics are often validated by using property-based approach[60] and measurement theory-based approach[61]. On one hand, the hybrid set of complexity metrics covers not only the minimal property set of OO metrics, including abstractness, inheritance, size, coupling, cohesion, polymorphism and encapsulation, but also some significant global features of software networks such as degree distribution and clustering coefficient. On the other hand, it is easy to judge the constructive validity of these metrics and to prove that they are characterized by ratio scales. Thus, the metrics of our set can be theoretically validated. A case study of empirical validation will be presented in Section 4.

3.3.3 Indicator for Structural Defects

Our hybrid set aims at measuring the specific properties of large-scale OO software systems at various levels so as to explore the probability of defect detection in terms of these metrics. Software engineers always seek to improve software quality by identifying fault-prone classes (or modules) with software metrics. So, the underlying goal of such a study is to better predict fault-prone classes by means of statistical metrics, OO design metrics and the significant correlations between them.

Based on the empirical results, Zimmermann *et al.* argued that network measures have advantages over traditional complexity metrics concerning the identification of graph-level structural defects[27,53]. Hence, we believe that statistical metrics are suitable to be the high-level defect indicator, which may offer a glance of the distribution of fault-prone nodes in a software network. Moreover, an in-depth analysis under the context of OOP would provide more detailed modification guidance for developers by virtue of the correlations between statistical metrics and OO design metrics, some of which have been reported to be significant predictors for fault proneness of classes in previous literature [5, 7, 10-12, 26].

1) Empirical data shows that if the average shortest path length of a software network (d) is much longer than 6, the overall efficiency of information transport among its nodes will be relatively poor. Hence, the metric provides an insight into the rationality of macroscopic structure design, especially for the design of "hub" nodes that are responsible for message switching.

2) Being similar to the metric CBO of CK (or Fan-in & Fan-out of a module), the degree of a node (k) in a software network actually indicates the degree to which each class (or program module) relies on each of the other classes (or modules). So, the average degree of a software network ($\langle k \rangle$) offers a graph-level metric that measures the degree of software coupling or dependency on the average, which resembles the metric CF of MOOD for the whole system.

3) High cohesion is often contrasted with low coupling. When the parts of a software entity are grouped according to a single well-defined task, the cohesion of the entity is high, suggesting that its parts tend to cluster together in a local community. Thus, average clustering coefficient (C) can assess the degree of cohesion of large-grained software entities such as subsystem or package at a high level, which is useful to complement the metrics for cohesion of a class (e.g., CK's LCOM). In general, a software entity in question with low average clustering coefficient demands for decomposition into several small components with high cohesion.

4) For a legacy system, it is a hard task for traditional OO metrics to quantify the system-wide significance of a random class or relationship. Fortunately, the betweenness centrality of our hybrid set enables a better understanding of the problem. The larger the betweenness of a node or a relationship ($B(v)$ or $B(e)$) is, the greater its impact on message traffic of the whole system will be when suffering intentional attacks or failures. That is to say, if a class with high betweenness but low connectivity fails, the link between subsystems (or packages) may be broken off. Therefore, from the perspective of fault-tolerance, the design of backup or exception handling for these classes or relationships is very important.

5) Since the era of structured programming, Fan-in & Fan-out have been deemed as an efficient indicator for potential design defects of a program module[62]. For complex software networks, the correlations between degrees (such as $corr(k_{\text{in}}, k_{\text{out}})$) can also provide an insight into their structure design independent of OOP context. For example, a simple scatter plot of in-degree vs. out-degree for every node is able to provide a system-wide distribution of defect-prone nodes. As we know, simple classes (or modules) tend to be heavily reused, whereas complex classes (or modules) are inclined to depend on other simple ones. Hence, there

are reasons to expect that such classes (or modules) with both high in-degree and high out-degree could be problematic. A case study to be introduced in Section 4 will confirm such a suspicion.

6) *Scale free* and modularity have been found to be important features of large-scale software networks[33]. High, size-independent clustering coefficient offers strong evidence for modularity, whereas power-law degree distribution strongly supports the scale-free model[30]. The scaling law $C(k) \sim k^{-1}$ quantifies the coexistence of a hierarchy of nodes with different degrees of modularity[57], which implies that the clustering coefficients of nodes generally decrease with the increase of their degrees. Hence, the metric $(corr(k_i, C_i))$ can be used to detect the design defect of nodes with high degree in short order. For instance, due in part to understandability and maintainability, the classes with high out-degree but low in-degree seldom cluster together to offer a more complex function, so their clustering coefficients are generally much smaller than those of nodes with low out-degree.

7) It is known that the metrics CBO, WMC and LCOM of CK, and the metric CF of MOOD have long been efficient predictors to identify fault-prone classes in OO software systems[5,7,10-12]. Even so, is there a more simple method to detect structural defects at a higher level? The further investigation on the correlations between graph-level measures such as out-degree and these traditional OO metrics may provide a more intuitive and statistical means for developers to predict the probability of structural defects based on large samples of empirical data, and to select the most relevant OO metric (or metrics) for the following in-depth analysis. In Section 5, the experiments of empirical validation on 12 open-source systems will be performed.

4 Case Study

4.1 System under Consideration

The system (SCRR) presented here is introduced in [63], whose main function is to register and manage software components based on domain ontologies. It was created with Eclipse 3.1 (available at http://www.eclipse.org) and Jena 2.2 (available at http://jena.sourceforge.net/), which has 4 plug-ins, 89 packages (36 packages are automatically generated by tools), about 600 classes without the classes of JDK or Jena, and more than 100 000 lines of Java code excluding all comments and statements.

4.2 Analysis Approach

The framework of our analysis approach is shown in Fig.2. Source code and class diagram are two kinds of software artifacts that can help describe the function, design and implementation of software. As the basic elements of OO software, classes and the relationships between them are usually extracted from Java/C++ code or UML class diagrams (which are not always available) to form a collection of element and relation, which constitutes an essential description of software.

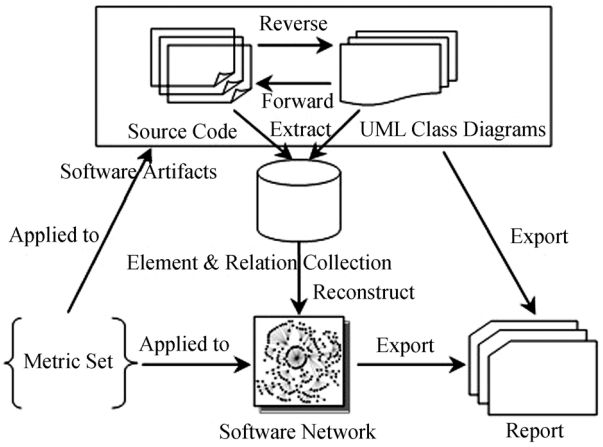


Fig.2. Framework of analysis approach.

Regardless of the formal semantics of elements and relations (with emphasis on the directivity) under the context of OOP, a software network can be reconstructed to characterize the global topological features of its corresponding software system. Then, developers can choose appropriate metrics from the hybrid set to measure necessary properties of software at diverse levels of granularity. According to numerical data of the indicators associated with design defects, a report akin to the result of medical diagnosis would provide helpful instruments or ways for programmers to improve software quality.

4.3 Empirical Validation

Because the system under consideration was designed and developed by ourselves, both Java code and UML class diagrams are available for our experiments. First, all Java class files that describe a class (including attributes and methods) and the collaboration with others (excluding the classes of JDK and Jena) are automatically examined by a simple algorithm. To ensure that the collection derived from Java code is correct, we also analyze UML class diagrams in a similar manner. For a collection of element and relation, the relation that links an element to itself is trivial.

Second, we reconstruct a software network according to the collection and visualize the network with an open-source tool pajek (available at http://vlado.fmf.uni-lj.si/pub/networks/pajek/). In Fig.3, nodes

with different colors belong to various plug-ins of SCRR. As we expected, in-degree and out-degree distributions of the network exhibit rough power-law scaling, with exponents $r_{\text{in}} \approx 2.2$ and $r_{\text{out}} \approx 2.7$. On the other hand, average shortest path length d equals 4.86 (smaller than random graphs with the same node set) and average clustering coefficient C equals 0.202 (much larger than random graphs with the same node set), which implies that SCRR also shares *small world* property.

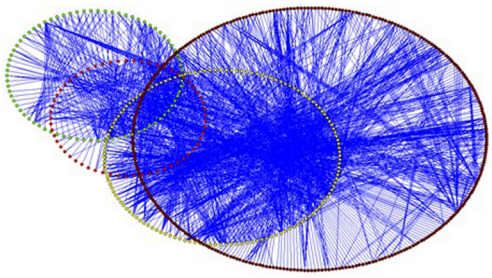


Fig.3. Topological structure of the experimental system.

Degree correlation of the network is shown as Fig.4, where every node is represented by its $(k_{\text{in}}, k_{\text{out}})$ pair. It is visually apparent that nodes with high out-degree generally have low in-degrees, and those with high in-degree usually have low out-degrees. Moreover, we also find that there is a negative correlation between in- and out-degrees for nodes with high degree. What we observed implies that, for the most part, there is a clear distinction between the levels of different functional hierarchies due to the strategies of *divide and conquer* and *high cohesion and low coupling*. So, like the traditional metric Fan-out & Fan-in, the improper ratio of out-degree to in-degree of a class may lead to potential defects, too. For example, the classes with both high in-degree and high out-degree (in the area of ellipse with dashed line of Fig.4) are conjectured to be problematic insofar as software development is concerned.

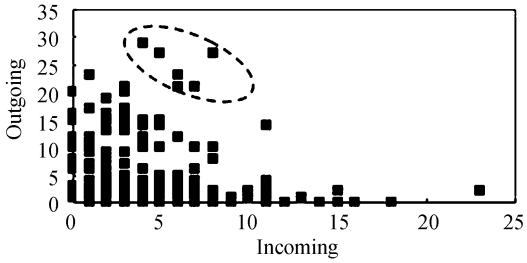


Fig.4. Scatter plot of k_{in} vs. k_{out}.

Finally, we use Eclipse metrics plug-in (available at http://sourceforge.net/projects/metrics) to calculate the metrics of CK and MOOD; at the same time, we also calculate Genero's metrics for UML class diagrams by an algorithm designed based on [15, 56]. All the experimental results are presented in Table 4. Note that SD and FP represent standard deviation and functional point, respectively.

According to the analysis of statistical metrics, SCRR is a middle-sized scale-free system, which has short d and large C, possibly indicating high efficiency of message transport and sound response capability. The metrics of CK, MOOD and Genero's metrics are very helpful measuring technique to evaluate software design quality and help developers to identify software defects. For example, a higher value of LCOM suggests decreased encapsulation and increased complexity, thereby increasing the likelihood of errors. So, system & component metrics are used to ensure a better functionality and quality as a whole under the context of OOP. Traditional metrics such as SLOC, MCC and Fan-in & Fan-out are employed to evaluate source-code complexity and software cost or effort at the implementation level. Hence, we believe that the hybrid set of metrics would contribute to measuring the complexity of practical OO software systems at different levels.

Table 4. Experimental Data of Our Metrics Set for SCRR

Statistics	N	L	r_{in}	r_{out}	d	C	$\langle k \rangle$						
	601	1829	2.22	2.68	4.86	0.202	6.09						
OOP Context	Plug-In	WMC		LCOM		EC		NOC		DIT		RFC	
		Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
	Model	44.99	42.51	0.20	0.33	6.66	5.80	0.72	1.74	5.44	2.32	4.13	2.12
	OntoM	9.58	7.54	0.15	0.29	6.89	7.02	0.24	0.91	2.24	1.22	3.64	1.23
	OntoUI	8.44	12.30	0.12	0.26	5.20	2.79	0.24	0.71	2.64	1.34	3.32	0.96
	Registry	7.17	7.81	0.20	0.31	6.16	3.43	0.04	0.38	2.43	1.15	6.21	3.21
	Class Diagram	NC	NAssocC	HAgg	MAgg	NAssoc	NAgg	NDep	NGen	NGenH	NAggH	MaxDIT	MaxHAgg
	SCRR	601	0.802	4	4	482	104	923	294	6	5	7	6
	System	AHF	AIF	CF	MHF	MIF	PF						
	SCRR	0.622	0.114	0.102	0.183	0.342	0.045						
Code	SLOC	NoC	FP	MCC									
	100 216	601	426	2.017									

5 Correlations Between Different Levels of Metrics

5.1 Data Collection

In order to investigate the significant correlations between statistical metrics and OO metrics, we collect the source code of 12 open-source OO software systems written in Java or C++, including aMule (an eMule-like file-sharing program available at http://sourceforge.net/projects/freshmeat_amule/), BORG Calendar (a calendar and task tracking system available at http://sourceforge.net/projects/borg-calendar/), db4o (an object database for Java and .NET available at http://sourceforge.net/projects/db4o/), FileZilla (a cross-platform graphical FTP client available at http://sourceforge.net/projects/filezilla/), GeOxygene (an OGC/ISO-based open framework for GIS available at http://oxygene-project.sourceforge.net/), HtmlUnit (a GUI-Less browser for Java programs available at http://htmlunit.sourceforge.net/), Tomcat (a servlet container available at http://tomcat.apache.org/), MUTE file sharing (a P2P-based file-sharing program available at http://mute-net.sourceforge.net/), OpenJUMP (an open-source Geographic Information System available at http://jump-pilot.sourceforge.net/), Roller Weblogger (a multi-user weblogging system available at http://sourceforge.net/projects/roller/), Tapestry (a comprehensive component-based web application framework available at http://sourceforge.net/projects/tapestry/) and WinMerge (a differencing and merging tool for window available at http://winmerge.org). Since the original UML class diagrams of these systems are unavailable, we leave Genero's metrics out of account when analyzing the correlations.

5.2 Data Processing and Analysis

For each experimental system, elements and the relations between them are extracted automatically from Java .class files or C++ .h and .cpp files. Redundant and trivial relations in the collection of each system will be deleted in the course of data pre-processing so as to avoid computing bias.

According to the collection derived from Java or C++ source code of each system, we reconstruct and visualize a corresponding software network (or sometimes the maximal connected sub-graph of the network) by means of an open-source tool pajek. The experimental results of basic measures for 12 software networks are presented in Table 5. The symbol r indicates the scaling exponent of degree distribution. To fit the curve of degree distribution well, the scaling region is adjusted with about 5% cutoff. All the 12 software networks are found to exhibit size-independent global features, namely the well-known *scale free* and *small world*.

As mentioned earlier, graph-level measures offer a higher level of measurement for the collaborations among components of software. Within the software engineering community, it has long been recognized that MOOD's CF and CK's CBO and LCOM are important indicators for fault-prone classes in terms of software coupling and cohesion[7,10-12]. In order to explore the correlations between two levels of metrics, in this paper we focus mainly on the metrics of CK and MOOD which are related to coupling, cohesion and complexity.

The values of metrics of interest are calculated by an open-source tool cccc (available at http://cccc.sourceforge.net/). Table 6 exhibits the distribution of CBO for each class in the 12 software systems. More than 80% of classes only have less than 10 dependencies with others, whereas a small number (less than 5%) of classes possess larger couplings more than 20, implying an approximate power-law distribution like the degree distribution for each node in the corresponding software network.

The metric WMC of CK is always used by software programmers to assess the complexity of a class. A high

Table 5. Results of Graph-Level Metrics for 12 Software Networks

Software	N	L	d	d_{rand}	C	C_{rand}	$\langle k \rangle$	r	r_{in}	r_{out}
db4o	2556	9808	3.106	4.032	0.267	0.003	7.674	1.902	1.755	2.429
OpenJUMP	1521	6828	3.055	3.524	0.197	0.005	8.978	2.239	1.976	2.322
WinMerge	987	2262	4.486	4.973	0.068	0.004	4.584	2.597	1.989	2.923
Tomcat	782	3188	2.901	3.204	0.259	0.102	8.153	2.151	1.838	2.267
Tapestry	735	2943	2.756	3.174	0.291	0.011	8.008	2.122	1.717	2.167
HtmlUnit	603	1514	3.048	3.978	0.261	0.008	5.022	2.048	1.878	2.385
aMule	562	1709	3.340	3.534	0.140	0.011	6.082	2.318	2.053	2.399
GeOxygene	435	1127	3.450	3.775	0.171	0.011	5.182	2.141	1.872	2.222
Roller Weblogger	405	1559	2.960	3.085	0.143	0.017	7.699	2.373	1.978	2.382
FileZilla	401	1031	3.435	3.724	0.132	0.012	5.142	2.163	1.856	2.363
BORG	397	1317	3.154	4.032	0.118	0.003	7.674	2.406	1.950	2.838
MUTE	316	703	3.882	4.152	0.113	0.013	4.450	1.953	1.790	2.467

Table 6. Distribution of CBO for Each Class in the 12 Software Systems

Software	Rate of Classes with CBO Between (%)						Average
	0∼5	6∼10	11∼20	21∼30	31∼40	> 40	CBO
db4o	68.47	17.32	9.35	1.99	0.86	1.92	7.716
OpenJUMP	52.74	26.73	14.71	2.29	1.11	2.41	8.925
WinMerge	78.94	13.60	4.72	1.13	0.47	1.13	4.272
Tomcat	90.42	6.35	1.61	0.85	0.08	0.68	2.566
Tapestry	57.47	26.49	10.87	2.58	1.22	1.36	7.997
HtmlUnit	81.57	12.23	3.10	1.63	0.16	1.30	4.939
aMule	69.40	17.56	8.86	2.01	1.00	1.17	5.715
GeOxygene	74.14	15.10	7.09	2.52	0.46	0.69	5.158
Roller Weblogger	54.43	26.60	13.30	2.22	1.72	1.72	7.680
FileZilla	73.91	16.91	6.28	1.21	0.48	1.21	4.981
BORG	60.15	23.56	12.53	2.51	0.75	0.50	6.602
MUTE	80.58	13.62	2.90	1.49	0.87	0.58	4.705

WMC value of a class, which seems disproportionately high compared to the complexity of other classes, suggests that the given class should probably be refactored into more classes. If all methods of a class have the same weight, the value of WMC is proportionate to the number of its methods.

Fig.5 shows the distribution (on a log-log scale) of WMC for each class in the 12 software systems on the assumption that the weight of all the methods within the system is 1. Interestingly, all distribution diagrams of these systems obey roughly a power-law distribution, implying an uneven distribution of software functionality in OO software systems. That is to say, most classes with few methods appear to be simple function that is easy to be reused, whereas few classes with large numbers of methods seem to be very complex and more likely to be prone to errors.

After the detailed introduction to analyzing CBO and WMC, it is unnecessary to go into details about LCOM. Then, we present the experimental results of code-level metrics for the 12 software systems in Table 7. Note that the number in italics indicates that its value is beyond the normal range of related metric. Although lines of source code of a class reflect an effort to construct the class, complex classes (in terms of SLOC and MCC, e.g., SLOC > 100 and MCC > 10) deserve special attention and careful handling, otherwise their readability, understandability and maintainability would be seriously affected.

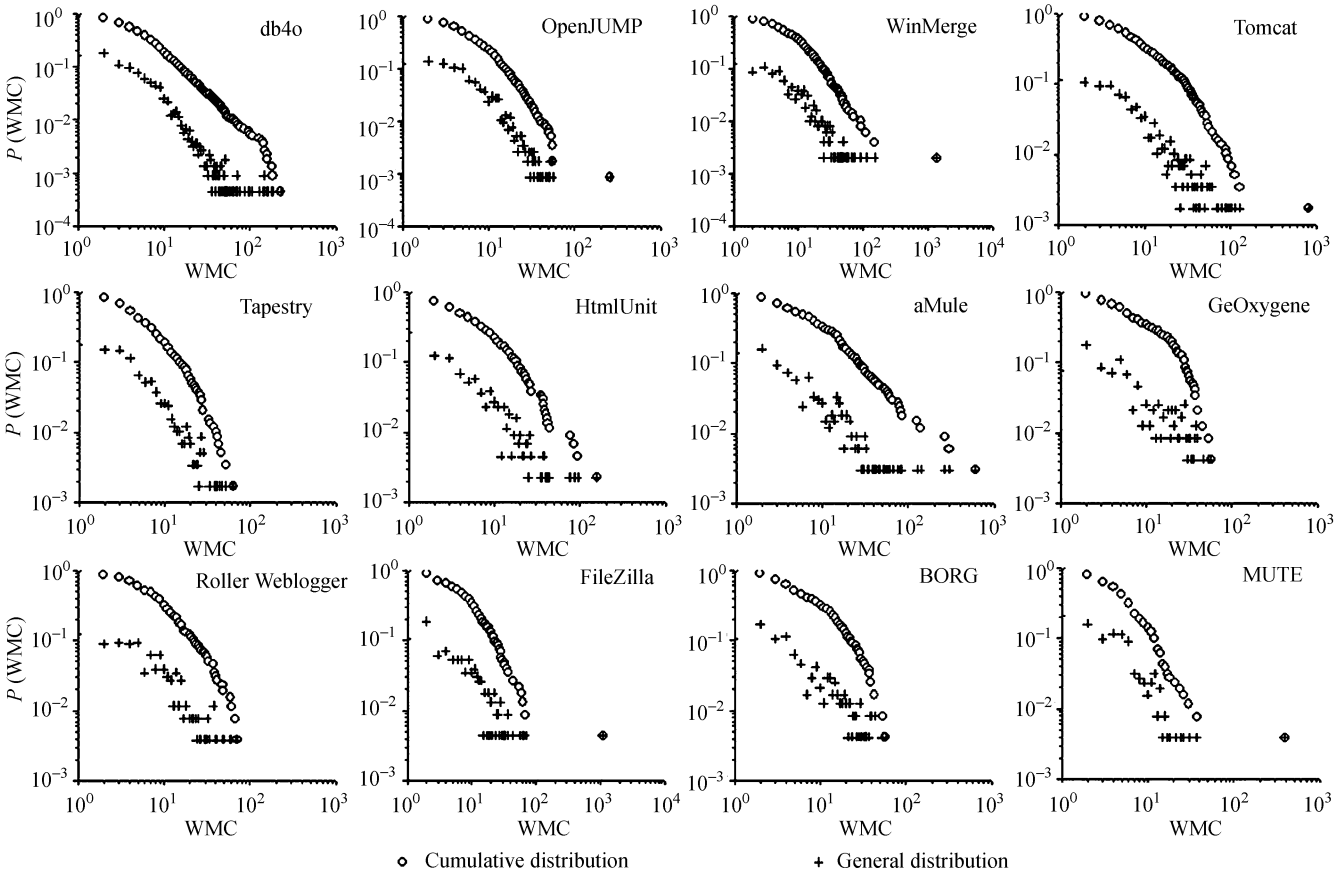


Fig.5. Distribution of WMC for each class in the 12 software systems.

Table 7. Experimental Results of Code-Level Metrics for the 12 Software Systems

Software	SLOC		Comment Lines		SLOC/Comment Lines	MCC	
	Total	Average	Total	Average		Total	Average
db4o	138 679	53.134	66 773	25.584	2.077	13 299	5.095
OpenJUMP	97 525	63.742	42 783	27.963	2.280	8 186	5.350
WinMerge	117 226	*113.153*	46 492	44.876	2.521	20 024	*19.328*
Tomcat	100 126	*118.492*	53 991	*63.895*	1.852	13 506	*15.983*
Tapestry	34 741	47.202	49 380	*67.092*	0.704	2 467	3.352
HtmlUnit	38 212	62.336	25 669	41.874	1.489	1 580	2.577
aMule	86 666	*147.642*	24 952	42.508	3.473	15 252	*25.983*
GeOxygene	30 923	70.762	18 297	41.870	1.690	4 595	*10.515*
Roller Webllogger	32 470	79.975	11 970	29.483	2.713	2 132	5.251
FileZilla	68 319	*152.158*	10 022	22.321	6.817	16 354	*36.423*
BORG	34 817	87.261	7 270	18.221	4.789	3 727	9.341
MUTE	28 269	82.417	15 730	45.860	1.797	4 163	*12.137*

5.3 Correlation Analysis

5.3.1 Degree-Degree and Degree-Clustering

As shown in the case study, there is a clear negative correlation between in-degree and out-degree of nodes with high degree, which may be an indicator for fault-prone classes. Another measure of degree correlations is the mixing by degree of a graph, which reflects the tendency that nodes with similar degree are connected to one another[33]. This could help developers to analyze and understand software structure better. Table 8 shows the values of all kinds of degree-degree correlation coefficients for the 12 software networks in question.

Table 8. Correlation Coefficients of Degrees for the 12 Software Networks

Software	In-In	In-Out	Out-In	Out-Out	Degree-Degree
db4o	−0.042	−0.048	**−0.194**	−0.058	**−0.140**
OpenJUMP	−0.038	−0.004	**−0.134**	−0.047	**−0.093**
WinMerge	−0.002	−0.117	**−0.187**	0.179	**−0.174**
Tomcat	−0.071	−0.051	**−0.198**	−0.069	**−0.147**
Tapestry	0.005	0.097	**−0.168**	0.028	**−0.066**
HtmlUnit	−0.037	−0.101	**−0.210**	0.031	**−0.101**
aMule	0.018	−0.038	**−0.139**	0.020	**−0.103**
GeOxygene	0.030	0.075	**−0.179**	0.054	**−0.054**
Roller Weblogger	−0.038	−0.067	**−0.130**	0.168	**−0.078**
FileZilla	0.087	0.065	**−0.141**	0.057	**−0.098**
BORG	0.060	−0.272	**−0.169**	0.220	**−0.132**
MUTE	0.088	−0.145	**−0.157**	0.358	**−0.121**

It is apparent from Table 8 that a weak negative correlation of out-in pattern occurs in all the 12 software networks. The dissortativity suggests that nodes with high out-degree do not tend to be linked to those nodes with high in-degree, because in part of a functional hierarchy of software design that discourages cross-layered collaboration. Similarly, a weaker negative correlation between the degrees in all the experimental systems probably reflects the tendency that classes with similar degree are uncooperative with each other. Such design may reduce the interactions among cross-layered classes so as to enhance their reusability.

Ravasz *et al.*'s recent work[57] suggested that degree-dependent clustering of the form $C(k) \sim k^{-1}$ (see dashed line in Fig.6) is a signature of hierarchical organization, which serves to resolve the obvious dilemma between power-law degree distribution and modular structure. Most of plots in Fig.6 such as db4o display a flat $C(k)$ for small k which rolls over to a k^{-1} tail at large k. These data of our experimental systems are roughly in line with those presented in [33, 36].

Due in part to *software reuse* (which implies a remarkable power-law in-degree distribution) and *divide and conquer* (which suggests a modular structure), we argue that this correlation reflects a hierarchical organization of software design. Classes with simple functions tend to cluster together to offer a more complex function, whereas complex classes do not in respect that they are highly specialized and only applicable in limited contexts. Therefore, a class with both high degree and large clustering coefficient is likely to be defective, which tallies with the early observation about the negative correlation between high-degree nodes in essence.

5.3.2 Correlations Between Cross-Level Metrics

On one hand, according to the property a metric can measure, there are some intuitive correlations between cross-level metrics. For example, we guess there must be a positive correlation between the degree of a node (k) and the metric CK's CBO of a class. Likewise, a positive correlation certainly exists between the average degree of a network ($\langle k \rangle$) and the metric MOOD's CF (and average CBO) of a system. All the values of this kind of correlation coefficients are introduced in

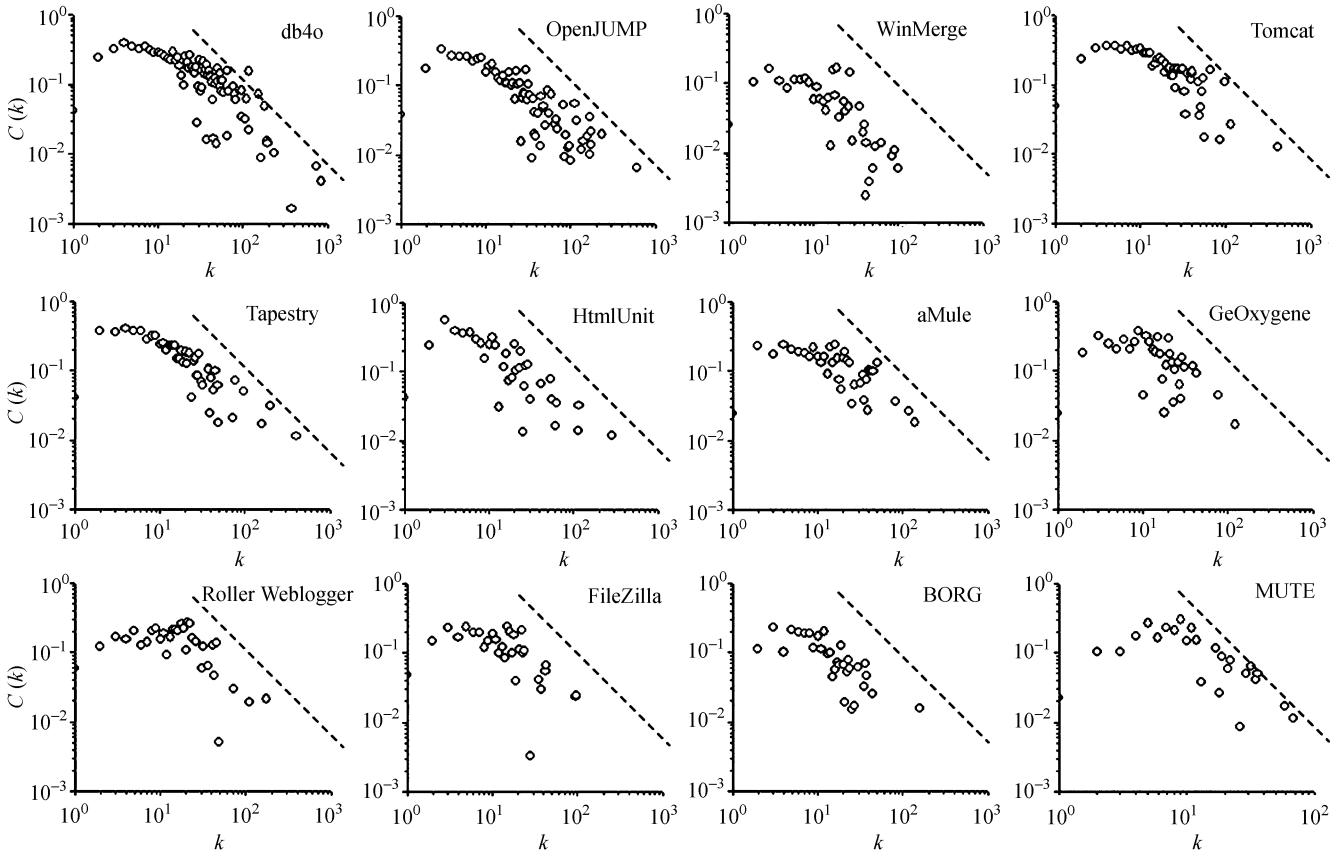


Fig.6. Clustering coefficient $C(k)$ vs. degree k.

Table 9. Values of Intuitive Correlation Coefficients

	k	$\langle k \rangle$	CBO	Average CBO	CF
k	-	-	0.683	-	-
$\langle k \rangle$		-	-	0.586	0.020
CBO			-	-	-
Average CBO				-	0.019
CF					-

Table 9. In sharp contrast with a strong positive correlation between k and CBO, we are somewhat surprised that the correlation of CF with either $\langle k \rangle$ or average CBO is rather weak, due in part to the different definition of coupling for MOOD's CF and CK's CBO.

On the other hand, a few implicit correlations need to be mined based on experimental data. For example, CK's WMC reflects the complexity of a class in terms of method. If a class has more methods (assuming that their weights are the same), we wonder whether its dependencies on other classes are greater. For a software network, an outgoing edge comes from the statements within a class that import other classes. If this is the case, a class with high out-degree deserves particular analysis to identify potential defects that would harm reusability and cohesion.

Then, we examine the correlation coefficient between the metric WMC and degree (k), in-degree (k_{in}), and out-degree (k_{out}), respectively, and find that out-degree has a very strong positive correlation with WMC (see Table 10) as we expected, suggesting that the more functions a class has, the more classes it would depend on. As we know, according to the principle of software construction, a complex class is often composed of many methods with simple function, which tend to build upon other classes by reference to their variables

Table 10. Values of Different Kinds of Correlation Coefficients Between WMC and Degree

Software	WMC $\sim k$	WMC $\sim k_{\text{in}}$	WMC $\sim k_{\text{out}}$
db4o	0.105	0.093	**0.666**
OpenJUMP	0.127	0.191	**0.652**
WinMerge	0.088	−0.043	**0.739**
Tomcat	0.175	0.108	**0.690**
Tapestry	0.126	0.181	**0.783**
HtmlUnit	0.094	0.069	**0.360**
aMule	0.174	0.068	**0.612**
GeOxygene	0.253	0.148	**0.587**
Roller Weblogger	0.161	0.020	**0.551**
FileZilla	0.229	0.079	**0.739**
BORG	0.118	0.037	**0.692**
MUTE	0.142	0.097	**0.318**

or methods. But unfortunately, there is no clear correlation between the reuse of a class (k_{in}) and its functions (WMC) from Table 10. We guess this may lie in part on a programmer's preference for heavily-reused classes and awareness of cautious use of complex classes.

In this case we conduct further investigation on the relationship between (average) WMC and k_{out} defined in [36]. As shown in Fig.7, the distribution of WMC(k_{out}) for all the 12 software systems (except MUTE) fits a straight line well on a log-log scale.

The observation indicates that WMC(k_{out}) follows a scaling law WMC(k_{out}) $\sim k_{\text{out}}^{\alpha}$. Interestingly, α for most of the systems in question (except MUTE) is around 1 (between 0.805 and 1.087, sometimes the unused import statements within classes may result in a slight bias that α is only a little bit smaller than its actual value), implying that average WMC of classes with a specific out-degree k_{out} is approximately proportional to their out-degrees. If the WMC of a specific class c_i with a given out-degree k_{out} exceeds the upper normal range (e.g., $\frac{\text{WMC}c_i - k_{\text{out}}}{\sigma_k} \geqslant 2.5$, where σ_k is the standard deviation of a set of WMC for all the classes whose out-degrees are k_{out}), we suggest that it needs an urgent and careful refactoring. If our observation can be empirically validated in a wider scope, it is believed that the correlation is useful to analyze and quantify the relationship between function distribution of a class and its structural characteristics from a system-scope viewpoint.

A high value of CK's LCOM suggests that methods in a class are not really related to each other and vice versa. Previous work found that the increase of WMC may lead to a larger LCOM[36,43], because without specific optimization newly-created methods often have few interactions with all the existing methods. Hence, based on our observation on WMC we guess that there is a positive correlation between k_{out} and LCOM. The experimental results of a sample of the 12 software systems are presented in Table 11 by carrying out a similar analysis process.

Positive correlations of the size of an OO software system (N) with both (total) MCC and (total) SLOC reflect a growth trend of software complexity. Besides WMC, the out-degree of a class (k_{out}) also has a clear positive correlation with its LCOM, MCC, and SLOC, respectively, which implies that on average a class becomes more complex as well as less cohesive along with the increase of external dependencies. However, there

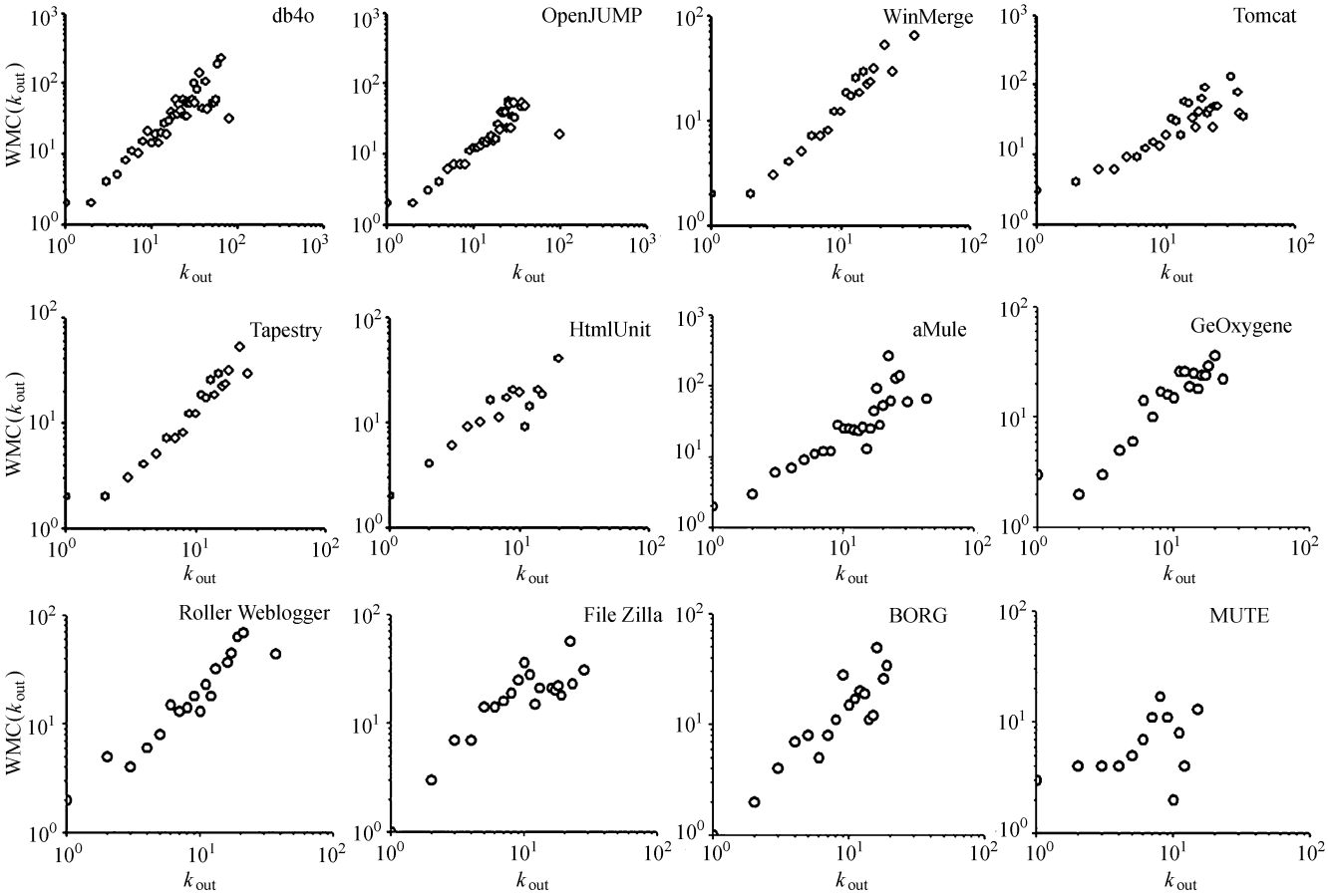


Fig.7. Distribution of WMC(k_{out}) on a log-log scale.

Table 11. Correlation Coefficients of Cross-Level Metrics

	WMC	LCOM	MCC	SLOC
N	-	-	0.328	0.781
k_{out}	0.607	0.384	0.297	0.402
WMC	-	0.269	0.371	0.796
LCOM		-	0.112	0.089
MCC			-	0.814

is no distinct correlation between LCOM and either MCC or SLOC, suggesting that the complexity growth of source code does not necessarily lead to low cohesion of a class. Therefore, we argue that out-degree may be a more effective and intuitive indicator to provide fault-prone information about classes in OO software systems at a high level of graph than traditional OO metrics such as CK's CBO and LCOM.

6 Conclusion

6.1 Limitations of the Set

The graph-level metrics of our set come from class-level software networks which are defined as directed graphs. Within the software engineering community, the significance of various kinds of relationships between classes is different. Hence, a weighted directed graph is more suitable to describe the topological structure of an OO software system. On the other hand, package- and method-level software collaboration (or dependency) graphs have also been proposed in [38-39, 41]. Obviously, their topological features may differ from what we discussed in the paper. A general measurement framework for multi-granularity software entities is our future work.

Our set offers multi-level metrics for a large-scale OO software system from the perspectives of graph, OOP context and source code. However, it is a very hard task to take all properties into consideration when designing a suite of complexity metrics. Hence, our work aims only at the properties that are necessary but not sufficient (e.g., the inheritance of OOP context and the efficiency of information transport of a network). On the other hand, according to the metrics of our set, in the paper we conduct experiments on the correlations between cross-level metrics. That is to say, our empirical studies focus mainly on fault-proneness detection in terms of software coupling and cohesion, which demand for a theoretical validation.

6.2 Summary and Future Work

An adequate set of complexity metrics for large-scale OO software systems based on the Internet is still a challenge for software engineering. In traditional software measurement methodologies, CK, MOOD and the metrics for UML diagrams have widely been recognized in practical software development. However, they do not have sufficient ability to measure some significant graph-level features, which have been found to recur in complex software systems as small-world and scale-free networks[6].

In order to measure a system's complexity at different levels of granularity, namely graph, OOP context and source code, we proposed a hierarchical set of complexity metrics that organizes graph-level measures (such as average shortest path length) and traditional metrics (such as CK, MOOD and MCC) in terms of a few necessary but not sufficient properties of a large-scale OO software system. Then, we proved the validity of our set through a case study that analyzes the empirical data from SCRR platform.

Furthermore, we investigated some intuitive and implicit correlations between cross-level metrics through a detailed analysis of 12 open-source OO software systems, and found that the out-degree of a node in software networks has a clear positive correlation with its corresponding class's WMC, LCOM, MCC and SLOC, respectively. The more classes a class depends on, the more complex (with low cohesion) it will become. Hence, the correlations could be an effective and simple indicator to detect fault-prone classes in a large-scale OO software system.

Our hybrid set of complexity metrics needs a wide range of empirical validation in spite of detailed empirical studies presented in the paper. So, more empirical validation is welcome to really prove that the proposed metrics set is fruitful in practice. On the other hand, recent work[64-65] showed that a few special small-scale sub-graphs (or motifs) describing the relationships among collaborating classes or objects are basic building blocks of complex software systems. Compared with the global measures of software networks, these metrics for the properties of local topological structure would provide a different insight into design quality and be a useful predictor for some specific kinds of structural defects.

Acknowledgements We appreciate Dr. Dehui Du (now at the East China Normal University) and Ms. Yulan Yan's (now at the University of Tokyo) contributions to the early version of this paper very much! We also thank anonymous reviewers for their valuable and constructive comments that help us to improve greatly the quality of our paper.

References

[1] DeMarco T. Controlling Software Projects: Management, Measurement, and Estimates. Englewood Cliffs, N.J.: Prentice-Hall PTR, 1986.

[2] Ramamoorthy C V, Tsai W T, Yamaura T, Bhide A. Metrics

guided methodology. In *Proc. the 9th Int. Comp. Softw. Appl. Conf.*, Chicago, USA, Oct. 9-11, 1985, pp.111-120.

[3] Chidamber S R, Kemerer C F. A metrics suite for object-oriented design. *IEEE Trans. Softw. Eng.*, 1994, 20(6): 476-493.

[4] Fernando Brito e Abreu, Regério Carapuça. Object-oriented software engineering: Measuring and controlling the development Process. In *Proc. the 4th Int. Conf. Softw. Qual.* (*ICSQ 1994*), McLean, USA, Oct. 3-5, 1994, pp.1-8.

[5] Brain H S. Object-Oriented Metrics: Measures of Complexity. Englewood Cliffs, N.J.: Prentice-Hall PTR, 1996.

[6] Ma Y T, He K Q, Du D H, Liu J, Yan Y L. A complexity metrics set for large-scale object-oriented software systems. In *Proc. the 6th Int. Conf. Comp. & Info. Technol.* (*CIT 2006*), Seoul, Korea, Sept. 20-22, 2006, p.189.

[7] Basili V R, Briand L C, Melo W L. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 1996, 22(10): 751-761.

[8] Harrison R, Counsell S J, Nithi R V. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Trans. Softw. Eng.*, 1998, 24(6): 491-496.

[9] El Emam K, Benlarbi S, Goel N, Rai S N. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.*, 2001, 27(7): 630-650.

[10] Subramanyam R, Krishnan M S. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 2003, 29(4): 297-310.

[11] Gyimothy T, Ferenc R, Siket I. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 2005, 31(10): 897-910.

[12] Olague H M, Etzkorn L H, Gholston S, Quattlebaum S. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Trans. Softw. Eng.*, 2007, 33(6): 402-419.

[13] Booch G, Jacobson I, Rumbaugh J. The Unified Modeling Language User Guide. Boston, MA: Addison-Wesley Professional, 1998.

[14] Kleppe A, Warmer J, Bast W. MDA Explained: The Model Driven Architecture: Practice and Promise. Boston, MA: Addison-Wesley Professional, 2003.

[15] Genero M, Piattini M, Calero C. A survey of metrics for UML class diagrams. *J. Obj. Technol.*, 2005, 4(9): 59-92.

[16] Manso M E, Genero M, Piattini M. No-redundant metrics for UML class diagram structural complexity. In *Proc. the 15th Int. Conf. Adv. Info. Syst. Eng.* (*CAiSE 2003*), Klagenfurt, Austria, Jun. 16-18, 2003, pp.127-142.

[17] Zhou Y, Xu B. Measuring structure complexity of UML class diagrams. *Chinese J. Elec.*, 2003, 20(3): 227-231.

[18] Genero M, Piattini M, Manso M E, Cantone G. Building UML class diagram maintainability prediction models based on early Metrics. In *Proc. the 9th Int. Symp. Softw. Metrics* (*ISSM 2003*), Sydney, Australia, Sept. 3-5, 2003, p.263.

[19] Genero M, Piattini M, Manso M E. Finding "early" indicators of UML class diagrams understandability and modifiability. In *Proc. 2004 Int. Symp. Emp. Softw. Eng.* (*ISESE 2004*), Redondo Beach, USA, Aug. 19-20, 2004, pp.207-216.

[20] Yi T, Wu F, Gan C. A comparison of metrics for UML class diagrams. *ACM SIGSOFT Softw. Eng. Notes*, 2004, 29(5): 1-6.

[21] 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 *Proc. the 12th Asia-Pacific Softw. Eng. Conf.*, Taipei, China, Dec. 15-17, 2005, pp.257-263.

[22] Chatzigeorgiou A, Tsantalis N, Stephanides G. Application of graph theory to OO software engineering. In *Proc. the 2006 Int. Workshop on Interdiscipl. Softw. Eng. Research* (*WISER 2006*), Shanghai, China, May 20, 2006, pp.29-36.

[23] Potanin A, Noble J, Frean M, Biddle R. Scale-free geometry in OO programs. *Commun. ACM*, 2005, 48(5): 99-103.

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

[25] Louridas P, Spinellis D, Vlachos V. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 2008, 18(1): Article 2.

[26] Kasunic M. The state of software measurement practice: Results of 2006 survey. Technical Report, No. CMU/SEI-2006-TR-009, Softw. Eng. Inst. at Carnegie Mellon Univ., 2006.

[27] Zimmermann T, Nagappan N. Predicting defects using network analysis on dependency graphs. In *Proc. the 30th Int. Conf. Softw. Eng.* (*ICSE 2008*), Leipzig, Germany, May 10-18, 2008, pp.531-540.

[28] http://en.wikipedia.org/wiki/Complex_network, Jul. 1, 2009.

[29] Watts D J, Strogatz S H. Collective dynamics of small-world networks. *Nature*, 1998, 393(6684): 440-442.

[30] Barabási A L, Albert R. Emergence of scaling in random networks. *Science*, 1999, 286(5439): 509-512.

[31] Guare J. Six Degrees of Separation: A Play. New York: Vintage Books, 1990.

[32] Barabási A L. Linked: The New Science of Networks. New York: Perseus Books Group, 2002.

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

[34] Moura A P, Lai Y C, Motter A E. Signatures of small-world and scale-free properties in large computer programs. *Phys. Rev. E*, 2003, 68(1): 017102.

[35] Valverde S, Solé R V. Hierarchical small worlds in software architecture. Working Paper, No. SFI/03-07-44, Santa Fe Institute, 2003.

[36] Liu J, He K Q, Ma Y T *et al.* Scale free in software metrics. In *Proc. the 30th Int. Comp. Softw. Appl. Conf.* (*COMPSAC 2006*), Chicago, USA, Sept. 17-21, 2006, pp.229-235.

[37] Wheeldon R, Counsell S. Power law distributions in class relationships. In *Proc. the 3rd IEEE Int. Workshop on Source Code Analy. & Manipul.* (*SCAM 2003*), Amsterdam, Holland, Sept. 26-27, 2003, pp.45-54.

[38] LaBelle N, Wallingford E. Inter-package dependency networks in open-source software. *arXiv*: cs.SE/0411096, 2004.

[39] Baxter G, Frean M, Noble J, Rickerby M, Smith H, Visser M, Melton H, Tempero E. Understanding the shape of Java software. *ACM SIGPLAN Notices*, 2006, 41(10): 397-412.

[40] Concas G, Marchesi M, Pinna S, Serra N. On the suitability of Yule process to stochastically model some properties of object-oriented systems. *Physica A*, 2006, 370(2): 817-831.

[41] Hyland-Wood D, Carrington D, Kaplan S. Scale-free nature of Java software package, class and method collaboration graphs. Technical Report, No. TR-MS1286, MIND Laboratory, University of Maryland College Park, 2006.

[42] Bilar D. Callgraph properties of executables. *AI Commun.*, 2007, 20(4): 231-243.

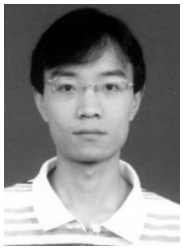
[43] Ichii M, Matsushita M, Inoue K. An exploration of power-law in use-relation of Java software systems. In *Proc. the 19th Australian Conf. Softw. Eng.* (*ASWEC 2008*), Perth, Western Australia, Mar. 26-28, 2008, pp.422-431.

[44] Cai K Y, Yin B B. Software execution processes as an evolving complex network. *Information Sciences*, 2009, 179(12): 1903-1928.

[45] Wen L, Kirk D, Dromey R G. Software systems as complex networks. In *Proc. the 6th IEEE Int. Conf. Cogn. Info.* (*ICCI 2007*), California, USA, Aug. 6-8, 2007, pp.106-115.

[46] Li D Y, Han Y N, Hu J. Complex network thinking in software engineering. In *Proc. 2008 Int. Conf. Comp. Sci. and Softw. Eng.*, Wuhan, China, Dec. 12-14, 2008, pp.264-268.

[47] Dijkstra E W. The Structure of the "T.H.E." multiprogramming system. *Commun. ACM*, 1968, 11(5): 453-457.
[48] Zhang H, Zhao H, Cai W *et al.* A qualitative method for analysis the structure of software systems based on k-core. *Dyn. Conti. Disc. Impul. Syst. B*, 2007, 14(S6): 18-24.
[49] Cai W, Zhao H, Zhang H, Zhao M, Luo G. Static structural complexity metrics for large-scale software. *Dyn. Conti. Disc. Impul. Syst. B*, 2007, 14(S6): 12-17.
[50] Vasa R, Schneider J G, Woodward C, Cain A. Detecting structural changes in object oriented software systems. In *Proc. 2005 Int. Symp. Emp. Softw. Eng.* (*ISESE 2005*), Noosa Heads, Australia, Nov. 17-18, 2005, pp.479-486.
[51] Jenkins S, Kirk S R. Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Information Sciences*, 2007, 177(12): 2587-2601.
[52] Vasa R, Schneider J G, Nierstrasz O. The inevitable stability of software change. In *Proc. the 23rd IEEE Int. Conf. Softw. Maint.* (*ICSM 2007*), Paris, France, Oct. 2-5, 2007, pp.4-13.
[53] Tosun A, Turhan B, Bener A. Validation of network measures as indicators of defective modules in software systems. In *Proc. the 5th Int. Conf. Predictor Models in Softw. Eng.* (*Promise 2009*), Vancouver, Canada, May 18-19, 2009, p.5.
[54] McCabe T J. A complexity measure. *IEEE Trans. Softw. Eng.*, 1976, 2(4): 308-320.
[55] Brooks F P. Three great challenges for half-century-old computer science. *J. ACM*, 2003, 50(1): 25-26.
[56] Genero M. Defining and validating metrics for conceptual models [Ph.D. Dissertation]. University of Castilla-La Mancha, 2002.
[57] Ravasz E, Somera A L, Mongru D A *et al.* Hierarchical organization of modularity in metabolic networks. *Science*, 2002, 297(30): 1551-1555.
[58] Newman M E J. The structure and function of complex networks. *SIAM Rev.*, 2003, 45(2): 167-256.
[59] Maslov S, Sneppen K. Specificity and stability in topology of protein networks. *Science*, 2002, 296(5569): 910-913.
[60] Briand L, Morasca S, Basili V. Property-Based Software Engineering Measurement. *IEEE Trans. Softw. Eng.*, 1996, 22(6): 68-86.
[61] Poels G, Dedene G. Distance-based software measurement: Necessary and sufficient properties for software measures. *Info. & Softw. Technol.*, 2000, 42(1): 35-46.
[62] Henry S M, Kafura D. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 1981, 7(5): 510-518.
[63] Ma Y T, He K Q, Liu W *et al.* A grid-oriented platform for software component repository based on domain ontology. In *Proc. 2007 IEEE Int. Conf. Services Comput.* (*SCC 2007*), Salt Lake City, USA, Jul. 9-13, 2007, pp.628-635.
[64] Valverde S, Solé R V. Network motifs in computational graphs: A case study in software architecture. *Phys. Rev. E*, 2005, 72(2): 026107.
[65] Ma Y T, He K Q, Liu J. Network motifs in object-oriented software systems. *Dyn. Conti. Disc. Impul. Syst. B*, 2007, 14(S6): 166-172.



Yu-Tao Ma was born in 1980. He is now a lecturer of State Key Lab of Software Engineering (SKLSE) at the Wuhan University as well as a post-doctor researcher of the Institute of Electronic System Engineering. He received the Ph.D. degree from the Wuhan University in 2007 and his M.S. and B.S. degrees from the Wuhan University of Science and Technology in 2004 and 2001, all in computer science. His current research interests include software metrics, software evolution and the interdisciplinary research between software engineering and complex networks.

Ke-Qing He was born in 1947. He is now a full professor of SKLSE and the director of Software Engineering Institute (SEI) at the Wuhan University. He received his Ph.D. degree in computer science from the Hokkaido University in 1995 and the B.S. degree in mathematics from the Wuhan University in 1970. His research interests include software infrastructure, software engineering based on complex systems, requirements engineering and software engineering technical standards.

Bing Li was born in 1969. He is now a full professor of SKLSE at the Wuhan University. He worked as a post-doctor researcher in SKLSE from 2003 to 2005. He received his Ph.D., M.S. and B.S. degrees from the Huazhong University of Science and Technology (HUST) in 2003, 1997 and 1990, respectively. His research interests include networked software, service-oriented software engineering, complex systems and complex networks, and cloud computing.

Jing Liu was born in 1979. She is now an associate professor of SKLSE at the Wuhan University. She was a visiting scholar at the Hong Kong Polytechnic University from 2007 to 2008. She received her Ph.D., M.S. and B.S. degrees from the Wuhan University in 2007, 2004 and 2001, respectively, all in computer science. Her research interests include software engineering and complex networks.

Xiao-Yan Zhou was born in 1986. She is now a Master's candidate of SKLSE at the Wuhan University. She received her B.S. degree in software engineering from the Wuhan University in 2009. Her research interests include software engineering and complex networks.