The Journal of Systems and Software 110 (2015) 28–53

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Analyzing maintainability and reliability of object-oriented software using weighted complex network

Chun Yong Chong*, Sai Peck Lee

Department of Software Engineering, Faculty of Computer Science and IT, University of Malaya, 50603 Lembah Pantai, Kuala Lumpur, Malaysia

ARTICLE INFO

Article history:
Received 26 February 2015
Revised 12 August 2015
Accepted 13 August 2015
Available online 24 August 2015

Keywords:
Complex network
Software complexity
Software maintenance

ABSTRACT

Modeling software systems using complex networks can be an effective technique for analyzing the complexity of software systems. To enhance the technique, the structure of a complex network can be extended by assigning a weight to the edges of the complex network to denote the strength of communicational cohesion between a pair of related software components. This paper proposes an approach to represent an object-oriented software system using a weighted complex network in order to capture its structural characteristics, with respect to its maintainability and reliability. Nodes and edges are modeled based on the complexities of classes and their dependencies. Graph theory metrics are applied onto the transformed network with the purpose to evaluate the software system. Comparative analysis is performed using 40 object-oriented software systems, with different levels of maintenance effort. We found that common statistical patterns from the software systems can be identified easily. It is when these software systems are grouped and compared based on their levels of maintenance effort that their statistical patterns are more distinguishable to represent some common behavior and structural complexity of object-oriented software. The evaluations indicate that the proposed approach is capable of identifying software components that violate common software design principles.

1. Introduction

Analyzing and understanding the behavior of software systems is sometimes associated with their modeling through complex networks, where software components are represented as nodes while inter-relationships between components are represented as edges. The concept of complex network has been successfully applied to various domains such as the World Wide Web, social network, power grid, and scholarly citation network, to provide a high-level graph abstraction view (Newman, 2003; Amaral et al., 2000).

An object-oriented software system is generally composed of several interrelated components, subsystems, or packages depending on the level of detail. Passing and exchange of messages, sharing of instance variables, function calls, method invocations, or inheritance relationships, suggest that two or more components have direct or indirect relationships. Various studies have applied the concept of complex network at the source code level, architecture level, or the hybrid of both (Concas et al., 2007; Louridas et al., 2008; Zimmermann and Nagappan, 2008).

Object-oriented software, in particular, is the main focus when associating software systems with complex networks because the concepts of object, method, and inheritance relationship provide a clear distinction between nodes and edges of a complex network. Edges can be weighted by counting the number of function calls or method invocations at the source code level. If UML class diagrams are chosen to be used at the architecture level, edges can be weighted based on the number of links connecting two or more classes (Valverde and Solé, 2003). Various metrics were proposed by researchers to quantify the complexity of a specific component, but the correlations of multiple metrics at different levels of abstraction are seldom addressed.

Furthermore, there is a lack of research that distinguishes different types of relationships connecting two classes. Instead, most studies that use UML class diagram as a basis for complex networks do not differentiate between the types of relationships such as association, generalization, composition, and aggregation, but assume that they are equivalent (Concas et al., 2007; Valverde and Solé, 2003; Myers, 2003). As such, semantic information between classes may be lost when transforming UML class diagrams into complex networks.

This paper aims to present a method to evaluate the complexity of object-oriented software at various levels of granularity based on three levels of metrics, namely code, system, and graph levels. We do not attempt to propose new metrics to evaluate the complexity

* Corresponding author. Tel.: +60 12 7326669; fax: +60 3 79579249.
E-mail addresses: cychong@um.edu.my (C.Y. Chong), saipeck@um.edu.my (S.P. Lee).

http://dx.doi.org/10.1016/j.jss.2015.08.014
0164-1212/© 2015 Elsevier Inc. All rights reserved.

of object-oriented software systems. Instead, we introduce an approach to consolidate and harmonize existing metrics that evaluate object-oriented software systems at multiple levels of abstraction. An object-oriented software system in the form of source code is first converted into UML class diagrams using an off-the-shelf round trip engineering tool. Based on the converted UML class diagrams, classes are converted into nodes while relationships between classes are converted into edges of a complex network. The nodes and edges are weighted using code-level and system-level metrics, which focus on the complexity of classes and their relationships. Next, graph-level metrics are used to analyze the software system from a graph abstraction's point of view. Forty object-oriented software systems are evaluated in this study to test the applicability of the proposed approach. By analyzing the statistical distribution of the graph-level metrics, we are able to capture several common patterns of the selected software systems. The captured patterns are able to represent certain structural characteristics of the software systems with respect to their maintainability and reliability. We then proceed to group the selected software systems based on their levels of maintenance effort and analyze the empirical distribution of their respective graph-level metrics using the boxplot approach.

The paper is organized as follows. Section 2 discusses the related work in modeling software systems using complex networks. Section 3 discusses the motivation and research objectives of this paper. This is followed by Section 4 that presents the proposed approach to evaluate object-oriented software systems using code-level, system-level, and graph-level metrics. Section 5 presents empirical testing to evaluate the maintainability and reliability of the chosen test subjects using the proposed code-level, system-level, and graph level metrics. Section 6 presents the comparative analysis of the proposed approach on the software systems that are grouped based on their levels of maintenance effort. Section 7 gives an overall discussion based on the results obtained in meeting the research objectives. Section 8 discusses the threats to validity in this study. Finally, concluding remarks and potential future work are presented in Section 9.

2. Related work

Object-oriented programming is arguably one of the most widely used programming paradigms to design and implement software systems (Meyer, 2000). The object-oriented paradigm advocates constructing software components with high modularity to improve the overall maintainability. However, software requires continuous change and maintenance to satisfy new business rules and technologies. Thus, it is common for software to become significantly more complex, insubstantial, and hard to maintain (Loo et al., 2012; Zamani et al., 2014).

To deal with software systems that undergo frequent changes, software engineers need to evaluate their quality and complexity after changes have been made (Hneif and Lee, 2011). This is to minimize the probability of faults introduced during maintenance work and also to prevent any faults from propagating to other parts of the software (Parizi et al., 2014). Evaluation of object-oriented software can be carried out at different levels of granularity in terms of classes, packages, or the entire system. However, evaluation of software is fuzzy in nature and a systematic approach is required to quantify the results (Turnu et al., 2013).

Software metrics, for instance, provide one of the means to conduct a quantitative assessment on software complexity and maintainability. Examples of well-established software metrics are the Chidamber and Kemerer's Metrics Suite (CK) (Chidamber and Kemerer, 1994) and the Metrics for Object Oriented Design (MOOD) (Abreu and Carapuça, 1994). CK and MOOD metrics are well known for measuring the complexity of object-oriented software as well as identifying software defects. CK metrics evaluate software at the class level by looking into factors such as class cohesion, coupling, complexity, and inheritance. MOOD metrics focus on object-oriented characteristics such as encapsulation, polymorphism, and inheritance to provide a system-wide assessment. In spite of their wide usage, CK and MOOD metrics share the same disadvantages where they focus mainly on single classes and rarely take the interactions between classes into consideration (Zimmermann and Nagappan, 2008). In addition, several works have found that the empirical effects of these metrics are less effective on large-scale object-oriented software systems (El Emam et al., 2001; Subramanyam and Krishnan, 2003; Gyimothy et al., 2005; Olague et al., 2007).

In recent years, research in software engineering in the aspect of modeling software systems using complex networks has started to emerge with the aim to gain a high-level abstraction view of the analyzed software systems (Ma et al., 2010; Concas et al., 2011). Modeling software systems using complex networks allows one to gain more insights on the studied software through the application of well-established graph theory metrics (Turnu et al., 2013).

Graph theory is a field of study that looks into the formal description and analysis of graphs (Bullmore and Sporns, 2009). A graph is generally defined as a set of nodes that are connected by edges, which may or may not be weighted. If the relationships or interactions between the nodes are asymmetric, then the graph is usually presented as directed graph, as opposed to the undirected ones. When describing a real-world network, be it social network, scholarly citation network, or software system, a graph provides an abstract representation of the network's elements and their interactions. Real-world networks display fundamental topological features and patterns that are not found in random networks, such as the scale free and small world properties (Concas et al., 2007; Myers, 2003). Graphs that are formed based on these real-world networks are commonly referred as complex networks, as discussed by Simon (1991), where he described a complex network as an integrated set of nodes that are organized in a hierarchical structure and interact in a non-simple way.

The work by Rozenberg and Ehrig (1999) discusses the notion of graph transformation and how it can be applied in software specification and development, VLSI layout schemes, database design, and several fields of computer science and software engineering. Graph transformation is a field of study that focuses on using formal and algorithmic methods to create a new graph out of an original graph. Based on certain transformation rules and constraints, the transformed graph is capable of revealing some extra deterministic information of the original graph. Object-oriented software systems in particular are good candidates for graph transformation because objects and classes are normally related through different kinds of binary relationships, such as inheritance, composition and dependence. Thus, the notion of associating graph theory to model large object-oriented software systems and to analyze their properties, be it structural complexity or maintainability, is feasible.

In the domain of associating software systems with graph theory, directed graph is more suitable because it is capable of capturing the semantic relationships between software components. The work conducted in Anquetil and Lethbridge (1999) and Davey and Burd (2000) shows that software features are asymmetric in nature. Examples of such behavior are superclass and subclass relationship, master and slave relationship in MPI (Message Passing Interface) programming, and an encapsulated object or data which has its internal behavior or function hidden from outside of its own definition (Chong et al., 2013).

Besides that, there are several features in graph theory that can be used to analyze the structure and behavior of software systems. Recent studies of representing objected-oriented software systems as complex networks revealed that many of them share some global and fundamental topological properties such as scale free and small world (Concas et al., 2007; Louridas et al., 2008; Potanin et al., 2005; Pang and Maslov, 2013; Baxter et al., 2006).

The scale free characteristic of a network is defined as follows. In graph theory, the degree k_i of a node i is the total number of its edges. In general, a node with a higher degree indicates that it possesses a higher impact with respect to the whole network. The average of k_i over all nodes is the average degree of a network, which is represented as k. The spread of nodes' degrees over a complex network is described as a distribution function $P(k)$, which is the probability that a randomly selected node has exactly k edges (Barabási et al., 2000). A simple network normally has a simple distribution function because all the nodes contain a similar number of edges (Barabási and Albert, 1999). Therefore, the probability of finding a node with edges is very high. However, studies have discovered that many complex networks derived from software systems obey the degree distribution of a power law in the form $P(k) \sim k^{-\gamma}$ where the function falls off more rapidly than an exponential function (Concas et al., 2007). This results in situations where a few nodes with very high degree (highly connected nodes) exist in these complex networks. Because the aforementioned power law is free from any characteristic scale, these complex networks are also called as scale free networks. The scale free characteristic in software systems can be interpreted as the level of reuse of important classes, or the number of dependencies between classes.

The small world property is related to the average shortest path length and clustering coefficient in graph theory. A shortest path is defined as the shortest distance between a pair of nodes in a graph. The average shortest path length, on the other hand, calculates the average number of steps along the shortest path between every pair of nodes in a network. The average shortest path length is often used to measure the efficiency of information passing and response time in object-oriented software. Existing works that use complex networks to analyze software systems indicated that the average shortest path length of software is around 6 (Valverde and Solé, 2003). A clustering coefficient, on the other hand, is the average tendency of pairs of neighbors of a node that are also neighbors of each other. It can be used to measure the degree to which nodes tend to cluster together. In short, networks that exhibit small world property signify that the distances between nodes are relatively shorter as compared to random networks.

Combining both small average path length and high clustering coefficient, one can identify the cohesion strength among software components from a graph theory's point of view. Thus, it is clear that complex networks and software metrics have their own advantages in analyzing the quality of software systems. CK and MOOD excel in evaluating class-level complexities, particularly in the object-oriented paradigm. Complex network, on the other hand, is capable of evaluating the impact of a particular class with respect to the whole system.

However, before applying graph theory metrics onto a software system to be analyzed, one must construct its complex network in advance. An object-oriented software is typically composed of multiple classes. At the source code level, classes in object-oriented software may contain data structures, objects, methods, and variables. Two classes can be considered related if there are actions such as passing of messages. Due to multiple ways of representing nodes and edges, there is a need to perform an in-depth review on existing works that model software systems using complex networks.

2.1. *Modeling software systems using un-weighted edges*

The work by Valverde and Solé (2003) discusses the usage of two graphs, namely Class Graph and Class-Method Graph, to analyze the global structure of software systems. Class Graph is derived based on UML class diagrams, where classes are represented as nodes, while relationships among classes, such as dependency and association, are depicted as edges between nodes. Class-Method Graph is modeled based on source code using the similar concept. For both types of graphs, the complexity of nodes and edges is ignored mainly because the authors assumed that internal complexities do not change the global structure of a software. Thus, the nodes and edges of the constructed graph are modeled to be un-weighted.

Myers (2003) proposed a method to model software systems using complex networks by analyzing the interdependencies of source code. A software collaboration graph based on the calling of methods by one another is used to analyze the structure and complexity of software systems. The work by Myers is later extended in the work by LaBelle and Wallingford (2004) and Hyland-Wood et al. (2006) to include the usage of classes and packages. However, similar to the work by Valverde and Solé, the software collaboration graphs by LaBelle et al. and Hyland et al. are constructed using un-weighted edges and nodes.

Jenkins and Kirk (2007) used source code as the basis to construct a software architecture graph. Classes in the source code are identified and represented by nodes. When a class accesses or refers to data or functionality in another class, it is represented as an un-weighted edge that connects both classes. Similar way of modeling a software architecture graph is also presented in the work by Louridas et al. (2008) using class interactions. The work by Louridas et al. aims to investigate the power law's behavior in software systems written in Java, C, Perl, and Ruby.

The work by Taube-Schock et al. (2011) investigates the problem of high coupling in software systems. Generally, the notion of high cohesion and low coupling is a well acknowledged software design principle. However, various studies that modeled software systems using complex networks have empirically discovered that all software exhibit the scale-free behavior, which means that in certain cases, some high coupling is unavoidable (Myers, 2003). In order to verify the aforementioned behavior, Taube-Schock et al. performed empirical studies using the Qualitas Corpus, which is a collection of 100 independent open-source software systems written in the Java (Tempero et al., 2010). The source code is converted into a directed graph, where nodes represent source code entities ranging from packages to variables, and edges represent connections between entities. The authors found that all the studied software systems exhibit a similar scale-free dependency structure and some high coupling might be a necessary characteristic for good software design.

On the other hand, the work by Oyetoyan et al. (2015) proposes an approach to investigate the relationship between circular dependencies and software maintainability. Cyclic dependency graphs are used in this work, where classes are represented as nodes and relationships between classes are represented as edges. The authors examined the change frequency of software components in multiple releases, and identified if the classes involved in circular dependencies are more prone to changes. Based on their finding, the authors discovered that circular dependencies are positively correlated to change frequency, and it will adversely affect the maintainability of software systems.

Finally, the work by Hamilton and Danicic (2012) proposes a Backward Slice Graphs (BSGs) made up of nodes that represent program statements and un-weighted edges that represent dependencies between these statements. The main purpose of BSG is to find groups of strongly related software components by inspecting the number of dependencies between program statements.

We found several commonalities based on the discussed literature that use un-weighted edges in modeling software systems. First, these works mainly use source code as a basis for modeling complex networks except for the work by Valverde and Solé. Most of them took a black box approach when transforming source code into nodes and edges by assuming that the types of relationships between nodes do not change the global structure of the constructed network. The types of relationships, such as inheritance and method invocation, are represented by a common type of 'dependency' to signify interactions between nodes. Besides that, the complexity of nodes is ignored. Instead, most work weights the importance and complexity of a particular node by counting the number of in-coming and out-going dependencies. The authors are more concerned on the global structure and behavior of the analyzed software, either from a static or evolutionary's point of view. For instance, the work by Valverde and Solé, LaBelle et al., Hyland et al. and Louridas et al. focuses on representing software using a static network abstraction and applied statistical analysis to discover the behavior of software systems. The work by Myers and Jenkins et al., on the other hand, took an evolutionary approach by inspecting software in different releases and compared the results using a statistical approach.

2.2. *Modeling software systems using weighted edges*

Next, we discuss some of the related work that deals with weighted complex network representation of software systems.

The work by Aier et al. (Aier, 2006, Aier and Schönherr, 2007) discusses the use of complex network in clustering enterprise architectures. Complex networks are modeled based on the respective enterprise business models and the business processes required to fulfill their business goals. Each business process is modeled as a corresponding complex network. IT systems are modeled as nodes, while the use of IT systems along with a business process, and the interrelationships of IT systems are modeled using weighted edges. The edges are weighted based on the number of connections that exists between two IT systems along with a business process.

Concas et al. (2007) demonstrated the use of weighted complex network to study the structure and properties of an object-oriented software called the VisualWorks Smalltalk. Nodes are made up of classes, while edges represent method calling between classes. By inspecting the source code, an edge is created toward each of the classes that implements a particular method and assigned a weightage value of 1. If the same method is called by multiple methods of a class, the weight of that corresponding edge is then multiplied by the total number of method invocations. Through this method, the constructed graph will be able to distinguish nodes with higher interdependency. However, the effectiveness of this method is highly dependent on the programming language and styles practiced by different individuals.

On the other hand, Bird et al. (2008) proposed a social network graph to investigate the organization of commercial software teams by studying the email exchanges of several open-source projects, including Apache HTTPD, Python, PostgresSQL, Perl, and Apache ANT. Open-source contributors are represented as nodes, while exchanges of email between the contributors are represented as edges. The edges are weighted based on the number of emails exchanged between multiple contributors. The authors argued that a high number of email exchanges between a pair of contributors signifies that they should belong to the same group of social structure.

The work by Sun et al. (Shiwen et al., 2009) investigates the structural properties of Linux kernel by constructing a complex network using C++ header files. The header files are represented as nodes, and two nodes are connected with a weighted edge if both header files are included in the same source file, i.e. using the 'include' operation. The weights of the edges are calculated based on the number of include operations in the header files.

In the domain of software testing, Lan et al. (Wenhui et al., 2010) proposed to use complex network to study software execution process, and subsequently identify the process that is more fault-prone. Functions are modeled as nodes and function calls as weighted directed edges. The edges are weighted based on the number of function calls between a pair of nodes, while the directionality of the edges is based on the sequence of the execution process. Empirical validations are conducted on Linux based programs, namely tar, gedit, and emacs. Nodes are not weighted. Instead, the authors used in-degree as the metric to identify highly critical components. In-degree in this context refers to the number of times a function is being called by other processes. Thus, the weights of edges are used to measure the complexity and significance of nodes.

A hybrid approach which extracts information from both the source code and software architecture is presented by Ma et al. (2010). The authors proposed a set of metrics to measure object-oriented software from multiple levels of granularity, at the class level, code level, graph level, and system level. Representation of nodes and edges depends on how much information is supplied during the analysis. For instance, if only the UML class diagram is provided, classes are represented as nodes, while interactions between classes such as inheritance and association are modeled as edges. The edges can be weighted using several well-known software metrics depending on the choice of the user, such as Genero's metrics (Genero et al., 2003) (number of associations, number of aggregations, and depth of inheritance). The same concept is applied to weight the nodes by using well-known software metrics such as CK and MOOD metrics. The focus of the work of Ma et al. is to measure the complexity of software, and subsequently, detect fault-prone software components.

The work by Wang et al. (Beiyang and Jinhu, 2012) presents an approach to model complex software systems using weighted complex networks. The authors used two open-source software written in Java, namely Junit and JEDIT to demonstrate the applicability of the approach. Classes in the source code are used to model the nodes. If a method of a particular class is dependent on other classes, a weighted edge is used to model this behavior. The weight of an edge is measured by inspecting the number of dependencies between a pair of nodes connected by the edge.

The works discussed in this section use very similar technique to measure the weights of edges. Almost all of them use the frequency of interactions among nodes, be it the number of class dependencies, number of method invocations, number of information exchanges, or number of method dependencies, to measure the weights of edges. A summary of the discussed literature is presented in Table 1. There is however an exception; the work by Ma et al. uses well-known software metrics to formulate the weighted edges. Most work discussed in Section 2.1 and this section counts the number of in-coming and out-going interactions between nodes to identify the critical nodes.

There are potential drawbacks of relying only on counting the frequency of interactions between nodes to quantify the weightage of edges. First of all, utility classes, such as classes with static methods that are heavily called by other classes, might distort the result of statistical analysis. Generally, a high number of interactions is often observed for utility classes that are used extensively in a software system. Thus, a high out-degree or edge-weighted value is not necessarily an indication of bad design and fault proneness (Ragab and Ammar, 2010). Besides that, an edge-weighted method based on the frequency of interactions might not be suitable for software in the domain of parallel processing and software that deal with a high number of information exchanges, i.e. mail servers and database systems. Flow of information and data across all layers of the system is very common for master and slave interactions in MPI programming, and database update or query operations in a typical database management system.

The advantages of using weighted complex network over un-weighted complex network are also discussed in the work by Wang et al. and Ma et al. Wang et al. found that distribution of nodes with a high in-degree and out-degree in edge-weighted networks is much more concentrated than those in un-weighted networks. Thus, one can easily identify the group of nodes that are critical to the systems, as well as those nodes that are associated with it. Based on the identified nodes with a high node-strength, software developers can choose to either decompose or reuse the associated software components to improve software stability and maintainability. Ma et al., on the other hand, observed that nodes with a high out-degree

Table 1
Summary of related work.

Related work	Type of network/graphs	Representation of nodes	Representation of edges	Weighted edges	Weighted nodes
Valverde and Solé (2003)	Class Graph/Class-Method Graph	Source code/UML class	Method dependencies/UML class relationships	No	No
Myers (2003)	Software collaboration graph	Source code	Calling of methods	No	No
LaBelle and Wallingford (2004)	Software collaboration graph	Classes and packages	Access or refer to other classes	No	No
Hyland-Wood et al. (2006)	Software collaboration graph	Classes and packages	Access or refer to other classes	No	No
Jenkins and Kirk (2007)	Software architecture graph	Classes	Access or refer to other classes	No	No
Taube-Schock et al. (2011)	Software collaboration graph	Packages, classes, methods, blocks, statements, and variables	Hierarchical containment, method invocation, superclass, implementation, variable usage, and method overriding	No	No
Oyetoyan et al. (2015)	Cyclic dependency graphs	Classes	Calling of methods, dependencies between methods	No	No
Hamilton and Danicic (2012)	Backward Slice Graph	Program statements	Dependencies between statements	No	No
Aier (2006), Aier and Schönherr (2007)	Enterprise architecture graph	Architecture elements	Usage or interrelationships between elements	Yes, based on frequency	No
Concas et al. (2007)	Class Graph	Classes	Calling of methods between classes	Yes, based on frequency	No
Bird et al. (2008)	Social network graph	Participants/Contributors	Exchange of emails	Yes, based on frequency	No
Sun et al. (Shiwen et al., 2009)	Software collaboration graph	C++ header files	Use of include operations	Yes, based on frequency	No
Lan et al. (Wenhui et al., 2010)	Software execution process	Functions	Functions call	Yes, based on frequency	No
Ma et al. (2010)	Hybrid Graph	Source code/UML classes	Multiple metrics	Yes, software metrics	Yes, software metrics
Wang et al. (Beiyang and Jinhu, 2012)	Class Graph	Classes	Calling of methods, dependencies between methods	Yes, based on frequency	No

generally have a low in-degree, and vice versa. Nodes with an improper ratio, i.e. high in node-strength and out node-strength, are nodes that do not adhere to high cohesion and low coupling design principle. Thus, these sets of nodes may lead to potential defects and bugs. Furthermore, Wang et al. found that the effect of bug propagation in a weighted network is lesser compared to an un-weighted network based on the analysis of average shortest path length. Similar observations are also found in the work by Ma et al. where the average shortest path of the analyzed software is around 4.86. All in all, weighted networks are found to be able to capture the behavior and characteristics of software systems in a more well-defined and detailed manner, especially when modeling the relationships among nodes.

However, most of the studies discussed are working on the source code level except for the works by Valverde and Solé and Ma et al. which also involve the software design level. At the software architecture level, UML classes are typically chosen to denote nodes providing a standardized conceptual model that represents the system's components, operations, attributes, and relationships. UML class diagram is a better choice when compared to raw source code because it is platform and language independent. UML class diagrams are also less susceptible to human factors, which in this context, refers to different programming styles practiced by different individuals. Because the structure, notations, and modeling of UML class diagrams are standardized, it is easier to construct complex networks based on class diagrams. We found that the work by Mat et al. is more comprehensive because it deals with source code and software design level, as well as the use of node and edge-weighted network.

3. Motivation

Based on the current research scenario, the approaches for representing edges in complex networks are still not well defined for software based on UML class diagrams. The edges signify direct relationships between two nodes where the edges can be associated with some weightage values to denote the communicational cohesion of nodes connected by the edges. Communicational cohesion in this context refers to classes that share similar characteristics and behavior, or classes that perform a certain operation on the same input or output data (Stevens et al., 1979). The notion of communicational cohesion in complex network can be associated with UML class diagram, where two classes with high communicational cohesion indicate that they share similar functionalities and there is a high tendency for these two classes to be placed into the same software package.

For instance, in the work by Wang et al., edges are weighted based on the number of method callings between classes. A higher value of weight associated with an edge signifies a higher communicational cohesion between the associated classes because it is an indication that these two classes might belong to the same package. On the other hand, one can also use distance as the basis to derive the weights instead of using communicational cohesion to indicate the dissimilarity between the associated classes. A typical way to convert communicational cohesion to distance measure is by calculating the inverse of the strength of communicational cohesion (Cilibrasi and Vitanyi, 2007). For example, if the weight of an edge (in terms of communicational cohesion) between two nodes is in the range of [0, 1], one can convert it to a distance value by computing 1− strength of communicational cohesion. There are diverse ways to transform the information observed from UML class diagrams.

The greater the weight of an edge (in terms of communicational cohesion), the more dependency exists between the two classes. For instance, given two classes *A* and *B*, where there exists one method in class *A* that passes messages associated with three methods in class *B*. Therefore, the weight of the edge that is linking classes *A* and *B* is assigned as 3. Such approach of transforming software systems into complex networks can be observed in the work by Yang et al. (2013), Yang et al. (2010), and Guoai et al. (2008). However, certain semantic behavior and relationships of class diagrams cannot be captured using this naive transformation. For example, classes related with inheritance relationships and classes related with common association would be assumed to have equal strength of communicational cohesion, which is illogical from a software engineer's point of view.

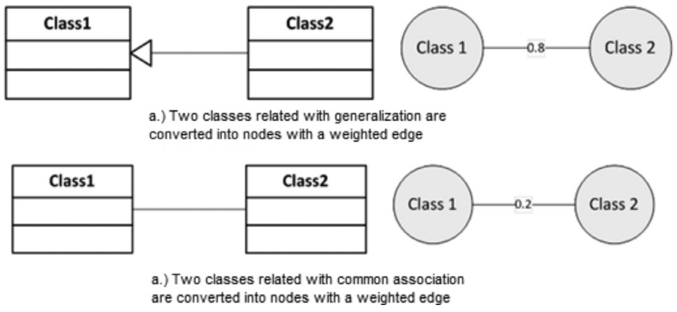

Class1
Class2
Class 1
0.8
Class 2
a.) Two classes related with generalization are converted into nodes with a weighted edge
Class1
Class2
Class 1
0.2
Class 2
a.) Two classes related with common association are converted into nodes with a weighted edge

Fig. 1. Example of UML classes related with different relationships.

As mentioned earlier, the work by Ma et al. (2010) explores the possibility of representing a UML class diagram as a directed complex network to analyze the relationships between classes at different levels of abstractions. However, the construction of edges does not consider the different kinds of relationships connecting multiple classes or the weighted values of edges. Instead, the authors assumed those edges to be equivalent.

The same assumption can also be observed in other work that relates software with complex network (Concas et al., 2007; Valverde and Solé, 2003; Myers, 2003). In the work by Concas et al. (2007), Valverde and Solé (2003), and Myers (2003), all kinds of interclass relationships such as inheritance, composition, and generalization are simplified and represented as common dependencies. This can be a poor assumption because different types of relationships in software such as dependency, association, composition, and aggregation denote different degrees of communicational cohesion and structural complexity. Fig. 1a shows a scenario where two classes are related with generalization, and Fig. 1b shows the same classes related with common association. In general, classes that are related with generalization signify a strong parent and child class relationship. This is because any changes in the parent class will directly affect the child class. Removal of the parent class would render the child class unusable. Classes related with generalization should have a higher degree of communicational cohesion when compared to classes related with common association. Thus, the strength of communicational cohesion between classes in Fig. 1a should behave differently when compared to that in Fig. 1b. In this case, this is shown in the transformed nodes and edges with random weighted values of 0.8 and 0.2.

In addition, the complexity of a class can also affect the complexity of a relationship. For instance, two simple classes related with common association should have a different strength of communicational cohesion when compared to two complex classes related with the same type of relationship. An example is shown in Fig. 2 to depict the aforementioned scenario. Given two complex classes shown in Fig. 2b that consist of hundreds of methods and variables. If the interactions between these classes are simply passing a few parameters, then the strength of communicational cohesion will be insignificant. On the other hand, if the classes are very well designed, simple and only contain two methods and variables, as depicted in Fig. 2a, then the strength of communicational cohesion between the two classes will be much stronger.

Thus, in general, the type of relationship and complexity of classes can influence the degree of communicational cohesion between classes. This is important because the degree of communicational

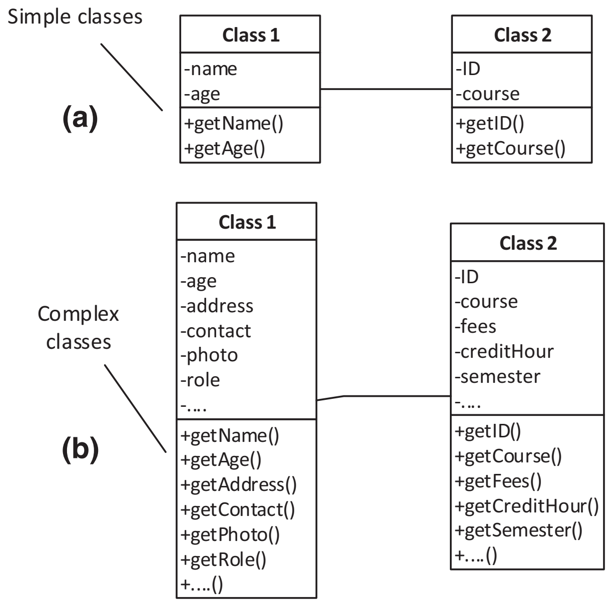


Fig. 2. Example of UML classes with different class complexity.

cohesion can subsequently affect the weights of edges in complex networks. Based on the discussed studies, we found that there is a lack of attention in formulating a proper way to calculate the weights of edges when transforming UML class diagrams into complex networks.

In this paper, we focus on the issue of preserving semantic relationships and information of software components when modeling software systems using complex networks. The research objectives (RO) are:

1. RO1—To identify appropriate measurement constructs that are capable of quantifying maintainability and reliability of object-oriented software systems modeled with complex networks.
2. RO2—To investigate the correlations between graph-level metrics and the selected software quality attributes related to maintainability and reliability.
3. RO3—To identify the effectiveness of the proposed approach when applied to object-oriented software systems with different levels of maintenance effort.

Suitable measurement constructs focusing on software maintainability are chosen to calculate the weights of edges in a complex network. Then, graph theory metrics are applied onto the transformed complex network to evaluate the quality of the chosen software systems. Finally, a comparative analysis is performed in order to compare and contrast the effectiveness of the proposed approach when applied to object-oriented software systems with different levels of maintenance effort.

4. Proposed approach

To tackle the problems mentioned in Section 3, we first provide justifications of the selected measurement constructs before introducing our proposed approach.

4.1. Measuring the structural complexity of UML relationships and classes

In the work by Dazhou et al. (2004), the authors defined “structural complexity” as the global metric to evaluate the complexity of UML class diagrams. According to the authors, structural complexity can integrate multiple class diagram metrics, such as metrics to evaluate individual classes, metrics to evaluate interaction between classes, and metrics to evaluate the whole class diagram.

Table 2
Ordering of relationships in UML class diagram proposed by Dazhou et al. (2004).

No.	Relation	Weight
1	Dependency	H1
2	Common association	H2
3	Qualified association	H3
4	Association class	H4
5	Aggregation association	H5
6	Composition association	H6
7	Generalization (concrete parent)	H7
8	Binding	H8
9	Generalization (abstract parent)	H9
10	Realize	H10

In this study, the proposed weighting mechanism is based on two parameters, the complexity of classes and the complexity of relationships. Relationships (dependency, realization, association, etc.) are taken into consideration because each end of the relationship must be linked to a certain class. This implies that the complexity of relationship has direct implication toward measuring the complexity of classes. In order to measure the complexity of relationships, the authors introduced a conceptual idea to assign different weightage depending on the type of relationship as shown in Table 2.

The table is arranged in an ascending order of weight values. Since the complexities of different relationships are relative with each other, arbitrary values of 1–10 are respectively given to H1–H10. Based on this ordinal scale, one can compare the complexities between different kinds of relationships in UML class diagram. Empirical testing using real open-source software has been demonstrated in Chong et al. (2013) based on the ranking in Table 2.

The work by Fang et al. (Hu et al., 2012) also proposes to rank UML classes and relationships using an ordinal scale based on PageRank algorithm. The purpose of the ranking is to differentiate the importance of associated UML classes based on their inherent characteristics and relationships with other classes. However, Fang et al. only tackled three types of UML relationships in the following order:

Composition $>$ Aggregation $>$ Association.

Similarly the research conducted by Briand et al. (2001, 2003) also involves the ranking of relationships in UML class diagram. Briand et al. mentioned that one of the most important problems during integrating and testing object-oriented software is to decide the order of class integration. The authors proposed a strategy to minimize the number of test stubs to be produced during software integration and testing phase. Relationships are ranked based on their complexities, where the most complex relationships (i.e. inheritance relationships) are integrated first. Common associations are perceived as the weakest links in class diagram and placed at the lowest hierarchy during software integration and testing phase. The discussed works (Hu et al., 2012; Briand et al., 2001, 2003) only compare three major types of relationships, namely inheritance (generalization and realization), composition (aggregation), and common association. The concept of ordering of relationships in UML class diagram based on their complexities is similar to the aforementioned work. Thus, we argue that the notion of ordering UML class relationship in an ordinal scale, and subsequently identifying the importance or complexity of classes, is feasible.

If multiple classes are related with the same kind of relationships, the weighting mechanism must be able to distinguish this difference. For example, two relatively simple classes linked with generalization (H7) should exhibit a different weightage when compared to two complicated classes linked with the same type of relationship. Thus, the complexity of classes plays an important role to make a distinction between these two cases. However, the proposed structural

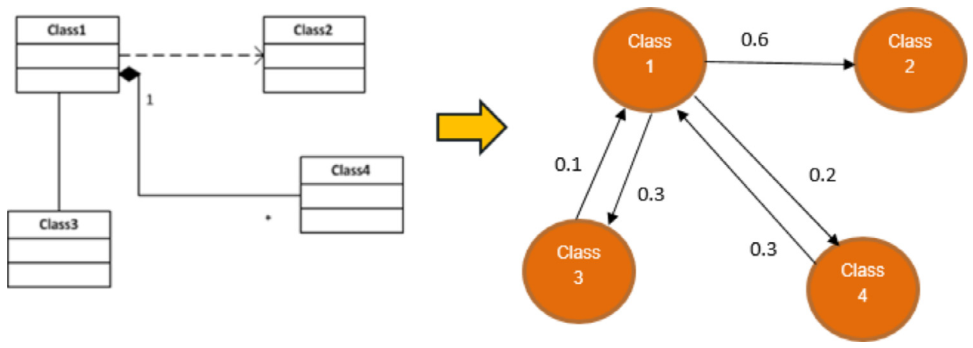


Fig. 3. Illustration of converting a UML class diagram into a complex network.

complexity metric in Dazhou et al. (2004) is a conceptual idea without a proper evaluation.

This study attempts to integrate the concept of structural complexity in order to convert UML class diagrams into complex networks. The object-oriented abstraction from a UML class diagram will be represented as nodes and edges in a complex network. From the constructed complex network, we can then calculate the weight of each edge. The weight of an edge is calculated based on the strength of communicational cohesion between the connected nodes, which will be discussed in the next section. The directionality of edges also plays an important role to indicate a one-way relationship from the origin node to the terminus node, not vice versa. Thus, it is important that the directionality of edges is captured and analyzed properly in order to provide a better understanding of the analyzed software.

As shown in Fig. 3, a class diagram $D = [D_1, D_2, \cdots D_n]$ consists of a set of n classes, $D_1, D_2, \cdots D_n$. The aim of this study is to model object-oriented software systems using complex networks in order to provide a graph abstraction view of the software. This will facilitate the application of well-established graph theory metrics onto the transformed complex network from the class diagram. Classes are represented as nodes while relationships among classes are represented as edges connecting a pair of nodes. Relationships in class diagrams can impose one-way or bidirectional relationships, which need to be interpreted in advance. As such, we adopted the transformation rules introduced by Dazhou et al. (2004) which convert association, composition, and aggregation into bidirectional relationships in a complex network. Since relationships such as generalization, realization, dependency and binding usually impose a one-way relationship in the model-driven architecture (MDA) perspective, they will remain as a single directed edge that links two nodes.

A relationship R connecting two classes, $R = (D_i, D_j)$, where D_i, $D_j \in D$; $i \neq j$, R links D_i to D_j where D_i is the origin of the relationship and D_j is the terminus. R carries a weight which denotes the strength of this relationship. The weight of relationship R, which denotes the strength of the communicational cohesion between classes D_i, D_j depends on:

1. The complexity of relationship R.
2. The complexity of classes D_i, D_j linked by R.

4.1.1. *The complexity of relation* R

The example below explains the details in calculating the weight of a given relationship R.

Given a class D_i that depends on class D_j through a one-way relationship R, such that $D_i \neq D_j$. The complexities of class D_i and class D_j are $Comp_{(i)}$ and $Comp_{(j)}$ respectively. Since this is a one-way relationship and D_i is dependent on D_j, the complexity of class D_j will affect this relationship. For a bidirectional relationship, the weight will be calculated based on the average of both directions. By referring to Table 2, we can identify the relative complexity of relationship R and measure the weight of the relationship R between class D_i and D_j using the proposed equation formulated in Eq. (1).

$$Weight_{(R_{i \to j})} = (H_{R_{i \to j}} \times \alpha) + \left[\left(Comp_{(D_j)} \right) \times \beta \right] \tag{1}$$

In Eq. (1), the first operand denotes the complexity of relationship R while the second operand denotes the complexity of terminus class linked by R. H_R indicates the relative complexity of relationship R (by referring to Table 2). The complexity of a relationship H_R is relative to the other types of relationships in Table 2. It is more significant to identify the ranking of this relationship in terms of influence and complexity. This can be done by assigning a relative weight in the range of [0, 1] to each relationship H_R based on its ranking. For example, given a relationship $R =$ Dependency (H1), a relative weight of 0.1 is assigned to this relationship. α and β in this context carry the meaning of preferences and risk tolerance in obtaining the relative complexity of relationship and complexity of the terminus class D_j. The preferences and risk tolerance parameters are used to relax the constraints on obtaining the complexity of the relationships and class. Since the ranking in Table 2 is presented in an ordinal scale, one can assign the weight of H1–H10 based on their own preferences. If users are not confident about the weight to be given on the relationship, more emphasis can be given on the complexity of the terminus class instead. Values of α and β range between 0 and 1, in such a way that a lesser value indicates a greater uncertainty in obtaining the complexity of the relationship R and the terminus class linked by it. For example, if the value of H1–H10 cannot be retrieved easily, or users are not confident regarding the weight of relationship R, value of 0.2 can be assigned to α, while 0.8 on β to indicate that the complexity of the terminus class linked by R carries more significance. Value of 0.5 for α and β will be used in this study to represent a balanced environment where both values can be obtained easily.

4.1.2. *The complexity of classes* D_i, D_j *linked by* R

In order to measure the complexity of a class, we adopt the three-level metrics introduced by Ma et al. In the work by Ma et al. (2010), the authors categorized their metrics into three levels, namely code-level, system-level, and graph-level, in order to analyze the object-oriented aspect of a particular system using different levels of granularity. The authors suggested that solely relying on a particular level of metrics is not sufficient to measure the properties of object-oriented software derived from source code or UML class diagrams. Hence, our proposed approach combines the information from both the raw source code and UML class diagrams of a software system to model its respective complex network.

The code-level metrics, which are at the finest level of granularity, measure the code complexity. Examples of metrics used are SLOC, fan in and fan out, and cyclomatic complexity (McCabe, 1976; Martin, 1994). The system-level metrics focus on object-oriented aspect of the system by measuring characteristics such as inheritance, coupling, cohesion, and modularity. Examples of metrics used are CK metrics. The CK metrics consist of six metrics as follows:

1. Coupling Between Object Classes (CBO).
2. Weighted Methods per Class (WMC).
3. Depth of Inheritance Tree (DIT).
4. Number of Children (NOC).
5. Lack of Cohesion of Methods (LCOM).
6. Response for a Class (RFC).

CBO measures the coupling of a given class by counting the number of dependencies of that particular class on other classes. WMC is the weighted sum of all the methods in a given class. DIT is based on the inheritance hierarchy by identifying the longest inheritance path for a given class. NOC of a given class is defined as the number of immediate child classes. LCOM, on the other hand, measures cohesion of a given class by inspecting the relationships between the methods declared in the class. Finally, RFC measures the number of methods that can be used by other classes through the associated messages.

The graph-level metrics are based on complex network to measure the global features and provide an overview of large-scale software systems. Examples of graph-level metrics are degree distribution and correlation coefficient.

While the work by Ma et al. (2010) aims to discover a variety of metrics and examine each of them individually, we choose to focus on only one metric at each level, except for the graph-level metrics. The measurement of each level is then normalized locally. The rationale behind this decision is because we wanted to propose a standardized metric with a common scale of unit that measures complexity as a whole. In our proposed approach, code-level and system-level metrics are used to measure the complexity of a particular class. On the other hand, several graph-level metrics will be used to examine the overall structure of the analyzed software.

In order to select an appropriate metric, a survey on the existing software metrics were conducted. We selected a few related studies that performed empirical studies of CK metrics on real-world software systems. The work by Li and Henry (1993) examined the correlations of CK metrics with software maintenance effort. The software metrics were applied on two commercial software systems in order to predict maintainability. The authors found that except for CBO, CK metrics are able to effectively predict the software maintainability of real-world software systems. Another work by Binkley and Schach (1998) applied CK metrics on four software systems. The authors found that there is no correlation between NOC and the frequency of source code changes due to field failure. The inventors of CK metrics themselves applied the proposed metrics on three commercial software systems and found that high LCOM was associated with lower productivity, high maintenance cycle, and higher maintenance effort (Chidamber et al., 1998). In terms of fault proneness, the work by Briand et al. (1999) applied CBO, RFC, and LCOM on an industrial software and found that the three metrics are associated with defects found in the case study. Similar observations were found in the work by Olague et al. (2007), where the authors applied the complete suite of CK metrics, MOOD metrics and QMOOD (Bansiya and Davis, 2002) metrics on Mozilla's Rhino open-source software. CK and QMOOD metrics are found to be good indicators for detecting error prone classes, while MOOD metrics are less effective. The work by Mei-Huei et al. (1999) applied CK metrics on three industrial-grade real-time systems to identify the correlations between the software metrics and faults found during software maintenance. DIT and NOC were found to have almost zero correlations, while WMC and RFC are strongly correlated to error prone classes. Based on the discussed literature, it is clear that not all of the CK metrics are suitable to be used in this paper. DIT and NOC in particular were found to be less effective when measuring the software maintenance effort and fault proneness of software systems. WMC and LCOM, on the other hand, were found to be suitable when identifying error prone classes and estimating software maintenance effort, as shown in Chidamber et al. (1998), Briand et al. (1999), Mei-Huei et al. (1999). Thus, we have decided to use WMC and LCOM in this study, which aligns to RO1 stated in Section 3.

At the code level, WMC is chosen to measure the complexity of source code. WMC measures the average cyclomatic complexity of methods inside a class. Cyclomatic complexity calculates the number of independent paths through program source code using the concept of directed network. A class that possesses high WMC value suggests that it is very complex and does not focus on its functionality.

At the system level, we opt for the Lack of Cohesion of Methods version 4 (LCOM4) (Hitz and Montazeri, 1995), which is an extended metric based on the LCOM included in CK metrics set. LCOM4 is used to measure the cohesion of a particular class by inspecting the relationships between the methods and variables. LCOM (Chidamber and Kemerer, 1994), LCOM2 (McCabe, 1976), and LCOM3 (Henderson-Sellers et al., 1996) are less suitable for modern object-oriented software systems because they do not evaluate the importance of mutator and accessor methods, which are widely used to encapsulate information in the object-oriented paradigm. In LCOM4, both shareable and non-shareable variables and methods are taken into account to cater for encapsulated variables or data. A high value of LCOM4 suggests that correlations between methods inside a class are weak, and it is undesirable in common software engineering practices.

The choice of software metrics used in this study is not random, as it is based on previous research (Subramanyam and Krishnan, 2003; Olague et al., 2007; Basili et al., 1996) that WMC and LCOM4 are complementary with each other when used to predict faults in object-oriented software. Furthermore, the work by Ichii et al. (2008) has found that there is a positive correlation between WMC and LCOM4 such that an increase in WMC will lead to an increase in LCOM4. This is mainly because a class will tend to become less cohesive when more functionalities or modules are added into the class. Furthermore, the focus of our paper is to capture two particular software quality attributes, namely maintainability and reliability as discussed in RO1 and RO2. Therefore, we decided to combine WMC and LOCM4 and use an aggregated measure to determine the complexity of classes when modeling object-oriented software using complex network.

An example is given below to calculate the complexity of a particular class. Given a class D_j, LCOM4 and WMC of class D_j are represented as $L(D_j)$ and $W(D_j)$ respectively. The following equation is used to quantify the complexity of D_j:

$$Comp_{(D_j)} = \left(\widetilde{L\left(D_j\right)} \times \alpha\right) + \left(\widetilde{W\left(D_j\right)} \times \beta\right) \tag{2}$$

where $0 \le \alpha \le 1,\ 0 \le \beta \le 1$.

$\widetilde{L(D_j)}$ and $\widetilde{W(D_j)}$ represent the normalized LCOM4 and WMC values respectively over all classes in the system using a ratio scale (value range between 0 and 1). Normalization is needed in this case because both metrics are measured using a different scale of unit. The values α and β behave similarly to Eq. (1) where it denotes the preferences and risk tolerance in obtaining the two metric values. Depending on the difficulty and confidence of obtaining the values $\widetilde{L(D_j)}$ and $\widetilde{W(D_j)}$, α and β can be manipulated accordingly. Thus, higher values signify higher complexity. However, before finalizing the formula, we need to determine if there is a direct correlation between complexity of classes and their respective strength of communicational cohesion.

The work by Satuluri and Parthasarathy (2011) proposes a degree-discounting symmetrization method to analyze the contribution of each node to the strength of communicational cohesion based on its degree (in-coming and out-going edges). Suppose that Case 1 with two nodes i, j, both point to a node z, which has a high in-degree (Fig. 4a), and Case 2 with two nodes i, j, both point to a node z, which has a low in-degree (Fig. 4b). The authors proposed that the strength of communicational cohesion between node i and node z together with that of node j and node z contributes more in Case 2 as compared

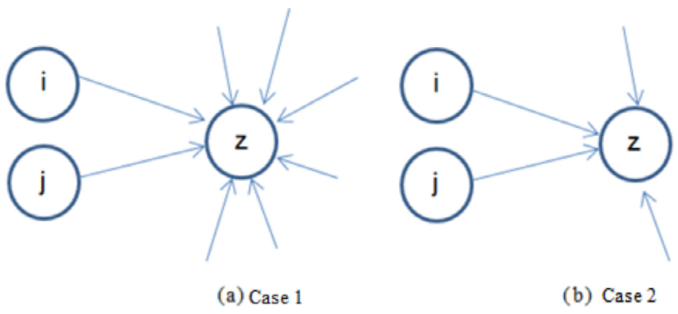


Fig. 4. Degree-discounting symmetrization based on Saturuli and Parthasarathy (2011).

to in Case 1 because node z has a low in-degree. The communicational cohesion between nodes is inversely proportional to the in-degree.

The concept of degree-discounting symmetrization can be applied to our case where the degree is represented as UML class complexity, $Comp_{(D_j)}$. Given $(D_i, D_j) \in R$, D_i is dependent on D_j, and the complexity of class D_j will affect the weight of R. If class D_j consists of 1000 methods and variables, and the interactions between D_i, D_j are simply passing a few parameters, the strength of communicational cohesion between D_i, D_j, or the tendency of D_i, D_j belonging to the same package will be insignificant. On the other hand, if class D_j is very well designed and focuses on its own functionality, then the strength of communicational cohesion between classes D_i, D_j will be much stronger. To provide a more concrete illustration, given for example, a complex and unorganized class "*Alpha*" which contains a lot of static methods and is constantly invoked by other classes from different software packages. As a result, cohesion of methods inside class "*Alpha*" will be very weak. Placing class "*Alpha*" into a suitable package will also be a challenging task because there are lesser distinct features and common behavior shared by class "*Alpha*" with other classes. A common way to solve this problem is by decomposing it into multiple modular classes that focus on their own functionality. The modular classes can then be grouped into packages that share the same functionalities.

Thus, the complexity of class D_j is inversely proportional to the weight (strength of communicational cohesion) of R. Since the value $Comp_{(D_j)}$ is normalized into the value range between 0 and 1, we can inverse the complexity by using the formula $1 - Comp_{(D_j)}$. Thus, Eq. (1) is updated to Eq. (3) to measure the relationships between two classes.

$$Weight_{(R_{i\to j})} = \left(H_{R_{i\to j}} \times \alpha\right) + \left[\left(1 - Comp_{(D_j)}\right) \times \beta\right] \tag{3}$$

4.2. Measuring software maintainability and reliability through a weighted complex network

By combining both methods to calculate the complexity of relationships and classes, we can produce the weights of edges (UML relationships) that connect between two nodes (UML classes) in complex networks. Finally, at the graph level, well-established graph theory metrics can be used to measure the cohesion strength among classes. The results from the graph-level metrics may offer additional insights toward understanding the complexity and maintainability of the software.

In our work, six graph-level metrics are chosen, namely in-degree, out-degree, average weighted degree, average shortest path of nodes, average clustering coefficient, and betweenness centrality. These metrics were selected because they are related to measuring software qualities (Concas et al., 2007; Valverde and Solé, 2003; Jenkins and Kirk, 2007). Besides that, average weighted degree, average shortest path, and average clustering coefficient in particular can be used to identify if a network follows the scale free and small world properties. The details of the metrics are explained in the following paragraphs.

As mentioned earlier, in a generic network, the degree k_i of a node i is measured by counting the number of edges that point toward or outward from the node. The in-degree is concerned with measuring the number of edges pointing toward the selected node. In the domain of object-oriented software systems, in-degree of a class represents the usage of that class by other classes (Concas et al., 2007). Classes with high in-degree suggest that they contain a high degree of reusability. However, if majority of the classes exhibit very high in-degree, software bugs can propagate easily to all related classes (Turnu et al., 2013).

On the other hand, out-degree is measured by counting the number of edges pointing out from the selected node. As such, out-degree represents the number of classes used by the given class. In the object-oriented paradigm, out-degree should be kept minimal to improve the modularity of software systems.

The average degree of a network is represented as k, where it represents the average degree of all nodes in a network. In this paper, the edges are weighted. Thus, average weighted degree is used instead. Average weighted degree of a node is calculated by summing up all the weights of edges linked to the selected node and dividing it by the total number of edges. If the distribution of average weighted degree, $P(k)$, exhibits power law behavior, it suggests that the constructed network obey the scale free characteristic. Power law characteristic also implies that there are a few important classes that are being heavily reused.

The average shortest path length used in this work calculates the average shortest path length between a source and all other reachable nodes for the weighted complex network. This will allow us to analyze the efficiency of information passing and response time of each node in the network.

A clustering coefficient measures the probability of a node's neighbors to be neighbors among themselves. A node with a high clustering coefficient indicates that there is a high tendency that the selected node will cluster together with its neighbors. The average clustering coefficient is used to represent the clustering coefficient of the whole network. In the object-oriented point of view, a network with a high average clustering coefficient indicates high cohesion strength among groups of related functionalities. It could be also used to determine the modularity of the analyzed software. Combining both the average shortest path length and average clustering coefficient allows one to examine if the network exhibits the small world characteristic.

The betweenness centrality of a node measures the number of shortest paths that pass through the selected node. It measures the importance and load of a particular node over the interactions of other nodes in the network (Yoon et al., 2006). Nodes with a high betweenness centrality often act as the communication bridge along the shortest path between a pair of nodes. Analyzing the betweenness centrality allows one to comprehend the robustness and structural complexity of a given software. One can recognize in advance the potential loss of communication if nodes with high betweenness centrality are removed from the network.

5. Empirical testing

This section starts by discussing the implementation plan to evaluate the effectiveness of the proposed approach. The evaluation is conducted using real-world datasets gathered from open-source projects.

5.1. Data collection

A total of 40 open-source Java software systems were chosen in this study. The sizes of the software systems vary from 128 to 2408 classes and 7436 to 216,093 lines of code. The software systems are chosen to reflect some representative distribution on the population of open-source Java software based on the following class count categories:

- less than 250 classes—seven projects,
- between 250 and 500 classes—11 projects,
- between 500 and 1000 classes—14 projects,
- more than 1000 classes—eight projects.

Because the nature of this work is based on an exploratory study, the selected software systems must be of high quality and reputable among the open-source communities. As it is, all the 40 software systems are being actively developed and maintained by a large number of open-source contributors. Besides that, three out of these software systems (JFreeChart, Apache Log4J, and Apache Maven) are adopted from the Qualitas Corpus (Tempero et al., 2010).

The selection of software greatly affects the results of empirical testing. The chosen software systems have to demonstrate a certain level of quality in terms of maintainability and reliability to allow for baseline evaluations and comparisons. Thus, the number of defects and maintenance cost of the chosen software systems have to be identified to allow for baseline evaluations and comparisons. However, as the selected software systems are open-source project, it is hard to accurately measure the maintenance cost of the selected software systems in terms of man-day. One alternative to measure the quality of open-source software is by the means of technical debt (Izurieta et al., 2013).

Technical debt, as discussed by Sterling (2010), is related to the issues in software that will hinder future development if left unresolved. Software systems with a high technical debt are at the risk of high maintenance cost. Curtis et al. (2012) argued that it is hard to measure technical debt using a generic measurement because identification of technical debts is based on the structural flaws that software developers intend to fix. Certain developers might just ignore the flaws or fail to recognize the flaws. Curtis et al. claimed that it is hard to quantify technical debt using a generic algorithm or technique.

The work by Heitlager et al. (2007) proposes a model to measure the maintainability of software based on ISO/IEC 9126 standard. A Maintainability Index (MI) is used to quantify the maintainability of software systems by analyzing the source code. The algorithm to calculate MI is based on several software metrics, including cyclomatic complexity and average number of lines of code per module. However, this technique only focuses on one particular non-functional requirement stated in ISO/IEC 9126 standard.

The work by Letouzey and Ilkiewicz (2012) introduces a method to estimate the technical debt of software systems by examining the source code. Letouzey and Ilkiewicz proposed the SQALE (software quality assessment based on life-cycle expectations) method that provides a systematic model to estimate technical debt, and subsequently ranks the severity of debts using five scales, ranging from A to E. Although there are no existing studies that attempt to demonstrate how SQALE ratings can correspond to actual development cost or effort, the work by Lim et al. (2012) did demonstrate a considerable finding on how software practitioners view technical debts, and how technical debts are relevant toward the maintainability of software systems. The authors showed that technical debts do play an important role in commercial projects and are widely recognized by software practitioners.

The SQALE method uses eight non-functional requirements, namely Testability, Reliability, Changeability, Efficiency, Security, Maintainability, Portability, and Reusability adapted from ISO/IEC 9126, as a reference to estimate technical debt of software. Software components that do not comply with the non-functional requirements are treated as debts. For each non-functional requirement, there is an estimation of time needed to fix the debt generated from the requirements. The sum of all the identified debts, along with the time estimated to solve them, are quantified as the total technical debt of a software system. As mentioned earlier, the focus of our work is to measure and analyze the maintainability and reliability of the software systems when modeled using complex network, which are also software quality attributes covered by the SQALE method. Thus, we argue that inclusion of SQALE method as a basis of measuring the maintenance cost of the selected software systems will allow for better comparative analysis.

To give a more concrete example on how to measure the maintenance effort of software, given a software with 5000 lines of code. The average cost of developing 1000 lines of code is 100 days, resulting in 500 days for the overall development cost. While analyzing the software using the SQALE method, it was discovered that there are several parts of the source code that do not conform to the reliability requirement. The rule “Switch cases should end with an unconditional break statement” is part of the reliability requirement. Given that there are five occurrences in source code that violate this rule, and the cost to fix this violation is approximately 1 day each. Thus, the total technical debt for Reliability is 5 days because of the violations. In order to measure the Reliability rating, the technical debt is divided by the total development cost, which is 5 days/500 days = 1%.

There are several rules for each of the non-functional requirements, such that each of the rules contributes toward estimating the technical debt associated with each non-functional requirement. Hence, eight indices are produced, namely SQALE Testability Index, SQALE Reliability Index, SQALE Changeability Index, SQALE Efficiency Index, SQALE Security Index, SQALE Maintainability Index, SQALE Portability Index, and SQALE Reusability Index which estimate the amount of technical debt associated with each of the ISO/IEC 9126 non-functional requirements. In order to provide a high-level indicator based on the ratio between the estimated technical debts and the development cost, all the aforementioned indices are aggregated into a single index called the SQALE rating. A SQALE rating, ranging from “A” to “E”, where A signifies high conformance of requirements, is rated. Thus, the overall SQALE rating for all eight non-functional requirements provides a systematic evaluation of the analyzed software.

In order to estimate the maintenance efforts of the selected software systems, we evaluate them using the SQALE method. The evaluation is performed using the SonarQube (2014) tool, with SQALE plugin installed. In the evaluation, software with overall SQALE rating of 0 to <2% are rated as A, while 2% to <4% are rated as B, 4% to <8% as C, 8% to 16% as D, and E for any rating higher than 16%. Below are the results of evaluations extracted from Table 3.

Software that achieved SQALE rating of A—Apache Maven Wagon, Apache Tika, openFAST, Apache Synapse, IWebMvc, JEuclid, Jajuk, Apache Mahout, Fitnesse, Apache Shindig, Apache XBean, Apache Commons VFS, and Apache Tobago.

Software that achieved SQALE rating of B—Apache Karaf, Apache EmpireDB, Apache Log4j, Apache Gora, Eclipse SWTBot, Apache Deltaspike, JFreeChart, Titan, Jackcess, Apache Pluto, Apache Roller, jOOQ, Apache Sirona, Apache Hudson, Apache JSPWiki, Apache Wink, Apache Commons Collections, and Apache Commons BCEL.

Software that achieved SQALE rating of C—Apache Rampart, Kryo, Apache Abdera, ApacheDS, Apache Archiva, Apache Helix, Struts, Apache Falcon, and Apache Mina.

We believe that the selected software systems can reveal some of the properties and characteristics of good object-oriented software. The preferences and risk tolerance variables, α and β, in calculating the complexity of edges and classes are set at 0.5. Table 3 shows an overview of the selected software systems.

5.2. Experiment design

Since the proposed technique transforms software systems from UML class diagrams to complex networks, pre-processing of the source code is needed.

Table 3
Summary of selected software systems.

No.	Name	Number of classes	Lines of code	Technical debts (days)	SQALE rating
1	Apache Maven Wagon	128	14,582	89	A
2	Apache Gora	131	8668	112	B
3	IWebMvc	178	7436	23	A
4	Apache Rampart	191	20,585	235	C
5	JEuclid	230	12,664	20	A
6	Apache Falcon	235	20,362	276	C
7	openFAST	236	11,656	63	A
8	Apache Commons VFS	280	23,059	34	A
9	Jackcess	302	21,452	180	B
10	Apache Sirona	345	57,736	428	B
11	Kryo	346	23,908	339	C
12	Apache Pluto	375	25,888	193	B
13	Apache Commons BCEL	396	28,966	325	B
14	Apache XBean	401	26,845	77	A
15	Apache JSPWiki	411	40,738	398	B
16	Apache Commons Collections	441	26,371	321	B
17	Apache Tika	457	34,558	200	A
18	Apache EmpireDB	470	41,775	307	B
19	Apache Archiva	506	75,638	535	C
20	Apache Roller	528	55,395	532	B
21	Titan	532	35,415	350	B
22	Jajuk	543	57,029	58	A
23	Apache Mina	583	36,978	723	C
24	Apache Abdera	682	50,568	783	C
25	Apache Log4j	704	32,987	209	B
26	Apache Helix	710	51,149	1561	C
27	Eclipse SWTBot	731	52,841	302	B
28	Apache Wink	740	54,416	930	B
29	Apache Karaf	773	46,544	662	B
30	Fitnesse	852	47,818	112	A
31	Apache Tobago	873	53,024	239	A
32	Apache Shindig	950	54,975	98	A
33	Apache Deltaspike	1002	31,504	502	B
34	JFreeChart	1013	95,396	670	B
35	jOOQ	1106	96,520	656	B
36	Apache Mahout	1130	82,002	143	A
37	Apache Synapse	1276	84,266	165	A
38	Apache Hudson	1492	119,005	1173	B
39	Struts	1646	120,025	2259	C
40	ApacheDS	2408	216,093	3664	C

5.2.1. *Convert source code into UML class diagrams*

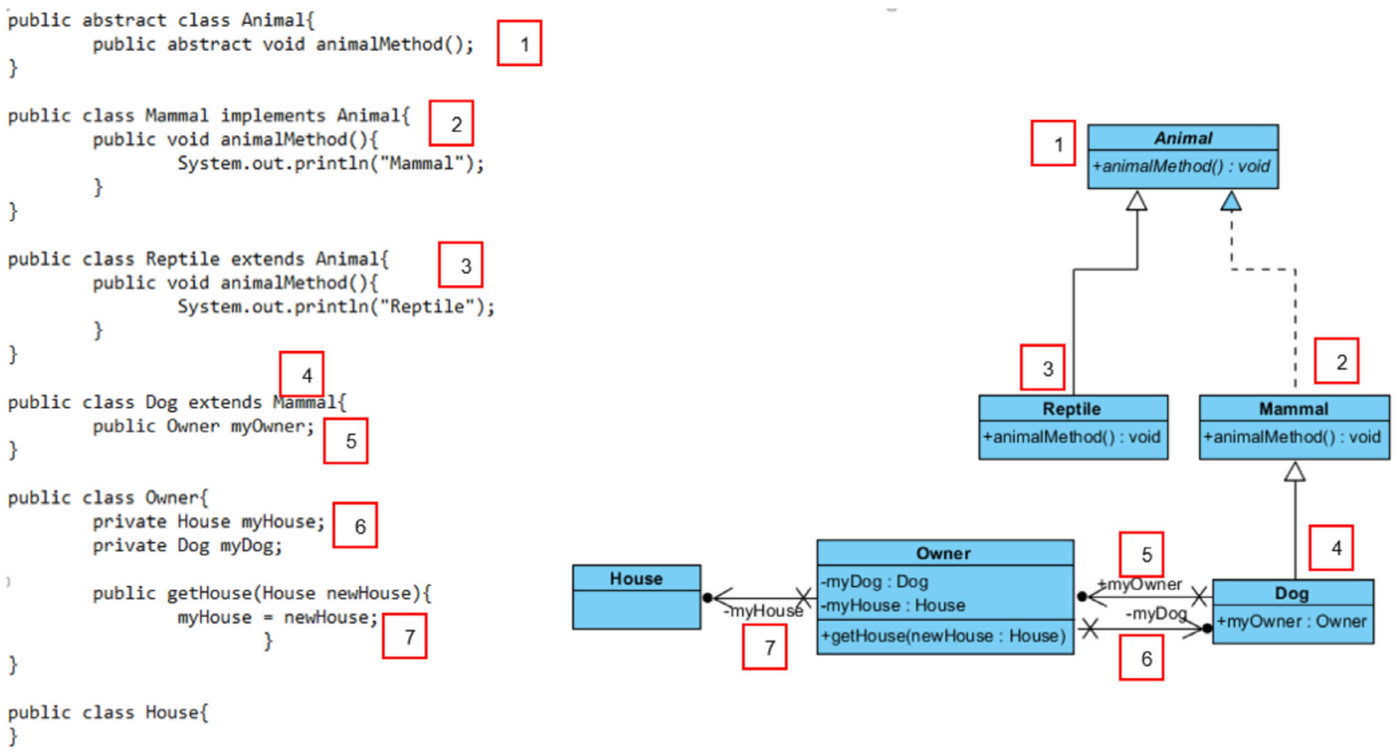
A round-trip engineering tool provided by Visual Paradigm is used to transform raw source code into UML class diagrams. Although one might argue that converting source code into UML class diagrams might result in loss of information, we attempt to mitigate this risk by incorporating code-level metrics, namely WMC and LCOM4 discussed in Section 4.1.2 toward modeling the respective complex networks. Fig. 5 shows an example of how Visual Paradigm converts Java source code into a UML class diagram.

1. Abstract class ‘Animal’ with one abstract method.
2. Class ‘Mammal’ implements abstract class ‘Animal’ to create a concrete class. Realization notation is used to represent the relationship.
3. Class ‘Reptile’ extends ‘Animal’. Generalization notation is used to represent the relationship.
4. Class ‘Dog’ extends ‘Mammal’. Generalization notation is used to represent the relationship.
5. An object myOwner of class ‘Owner’ is created. Association notation is used to represent the relationship.
6. An object myDog of class ‘Dog’ is created. Association notation is used to represent the relationship.
7. An input parameter newHouse of class ‘House’ is parsed into the method getHouse(). Association is used to represent this relationship.

Note that Visual Paradigm is unable to reverse engineer composition and aggregation relationships from the source code. Instead, association is used to represent the relationship. Thus, we carefully studied the relevant relationships in the source code and manually extracted this information to form composition and aggregation. One way to automatically differentiate between composition and aggregation relationships is by checking if deep cloning or shallow cloning is used in the clone() and equal() methods declared in the software.

Shallow copy refers to methods that create a new object that has an exact copy of the values in the original object. If the original object contains references to other object, shallow copy will only copy the associated memory addresses. On the other hand, deep copy copies the complete data structure of the original object recursively and allocates new memory addresses in a different location. From the modeling perspective, deep cloning implies composition relationship while shallow cloning suggests aggregation between two classes (Porres and Alanen, 2003; Karsai et al., 2004).

According to the Java API documentation (Oracle, 2015), Java provides a ⟨Cloneable⟩ interface which allows cloning of objects. Implementing this interface allows programmers to duplicate objects by calling the clone() method in java.lang.Object class. By default, the clone() method creates a new object instance of the class and initializes all the fields of the new object with exactly the contents of the corresponding fields of the original object. As such, the contents of the newly created object are not cloned directly, which is commonly referred as shallow copy. If the programmer wants to perform a deep copy, he/she has to override the clone() method and give his/her own definition of the cloning operation for all the variables, methods, and constructors, declared in the original object.



```
public abstract class Animal{
        public abstract void animalMethod();
}

public class Mammal implements Animal{
        public void animalMethod(){
                System.out.println("Mammal");
        }
}

public class Reptile extends Animal{
        public void animalMethod(){
                System.out.println("Reptile");
        }
}

public class Dog extends Mammal{
        public Owner myOwner;
}

public class Owner{
        private House myHouse;
        private Dog myDog;

        public getHouse(House newHouse){
                myHouse = newHouse;
                        }
}

public class House{
}
```

Fig. 5. Example of Java to UML class diagram transformation.

However, we found that most of the equal() and clone() methods used in the chosen test subjects do not override the default equal() and clone() methods declared in java.lang.Object class. We are unsure if it is intended or the programmers do not actually differentiate between deep and shallow copy. Therefore, we are unable to automatically extract all the composition relationships using a simple program parser.

According to the UML specification documents, composition and aggregation are specific forms of association relationship between two objects or classes. Composition is referred as the type of association when one object owns another object, as depicted in Fig. 5 when 'Owner' class owns an object of 'House'. Aggregation, on the other hand, is described as a whole–part relationship between source and target classes (Grand, 2003). The work by Kollmann et al. (2002) discussed that the best way to recover composition and aggregation relationships from source code is to acquire sufficient knowledge of the software architecture. However, since we only have limited information and are not involved in the development of the selected software systems, we relied on the clone() and equal() methods to differentiate the types of relationships. Unless otherwise indicated, the programmers did not differentiate between shallow clone and deep clone methods and we are unable to distinguish between aggregation and composition relationships, so aggregation is used to represent both scenarios.

There is a growing interest among the research community to formally identify composition and aggregation relationships for UML class diagrams from raw source code (Milanova, 2005, 2007; Yann-Gaël, 2004). The work by Milanova had proven that their proposed approach can achieve perfect accuracy when capturing composition relationships from raw source code. However, Milanova suggested that there are no definitive conclusion or solution that can be drawn from the limited and small-scale experiment setup.

Since the focus of our work is on modeling software systems using complex network, we chose to opt for a more straightforward and simple method. We deployed the selected software systems in Netbeans IDE and used the hierarchical view functionality to identify the aforementioned relationships. This allowed us to manually assign the aggregation or composition relationship onto related classes.

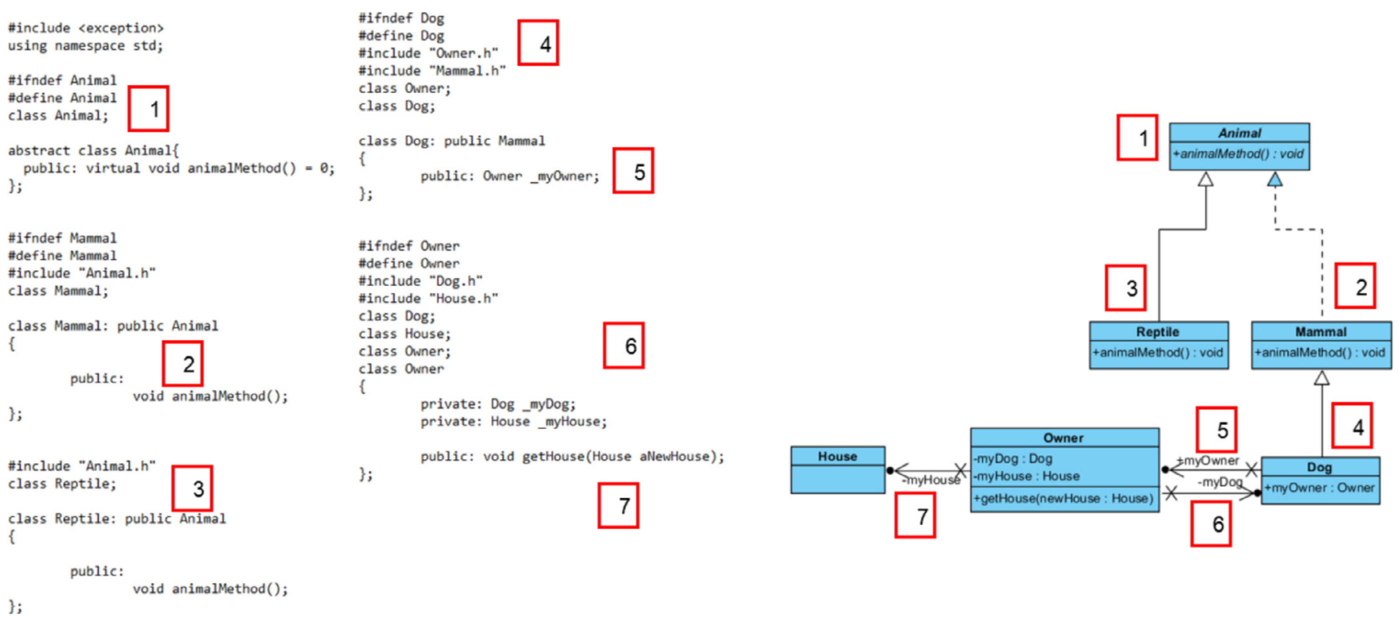
Fig. 6 shows an example of how Visual Paradigm converts C++ header files (.h) into a UML class diagram.

Fig. 7 illustrates the result of transformation from UML class diagrams to a complex network using the proposed technique, facilitated by Cytoscape, which is an open-source software tool to visualize complex networks (Shannon et al., 2003). Fig. 7 is modeled based on Apache Synapse system which consists of 1276 classes, where each class in the software system is represented by one node. The experimental subjects involve all the maximal connected subgraphs of the software system. However, we need to emphasize that the proposed approach can also be applied for forward engineering in software development to provide a better understanding of the software during the early stage of development. The Cytoscape and Visual Paradigm data files of all 40 software systems, along with the retrieved graph-level metrics can be accessed by the following URL: http://sourceforge.net/projects/umltocomplexnetwork/files/

5.2.2. *Datasets distribution fitting*

Similar to the work performed by Ferreira et al. (2012), we attempt to find the best fit probability distribution of all the selected graph-level metrics. The best fit distribution will be able to show the common patterns and structural characteristics of the chosen software systems, and subsequently identify the correlation between graph-level metrics and the associated software quality attributes, i.e. maintainability and reliability.

In terms of the statistical distribution of complex network based on software system, the work by Concas et al. (2007) found that in-degree follows a Pareto distribution while out-degree follows a log-normal distribution. Both distributions exhibit a power law distribution, which agrees with several works (Louridas et al., 2008; Valverde and Solé, 2003; Zimmermann and Nagappan, 2008). The reason why in-degree and out-degree are distributed in a power-law manner was further analyzed in the work by Chatzigeorgiou and Melas (2012). Chatzigeorgiou et al. discovered that software follows a



```
#include <exception>
using namespace std;

#ifndef Animal
#define Animal
class Animal;

abstract class Animal{
  public: virtual void animalMethod() = 0;
};


#ifndef Mammal
#define Mammal
#include "Animal.h"
class Mammal;

class Mammal: public Animal
{

        public:
                void animalMethod();
};


#include "Animal.h"
class Reptile;

class Reptile: public Animal
{

        public:
                void animalMethod();
};
```

```
#ifndef Dog
#define Dog
#include "Owner.h"
#include "Mammal.h"
class Owner;
class Dog;

class Dog: public Mammal
{
        public: Owner _myOwner;
};


#ifndef Owner
#define Owner
#include "Dog.h"
#include "House.h"
class Dog;
class House;
class Owner;
class Owner
{
        private: Dog _myDog;
        private: House _myHouse;

        public: void getHouse(House aNewHouse);
};
```

Fig. 6. Example of C++ to UML class diagram transformation.

'preferential attachment' where classes tend to interact with the classes that belong to a similar community or functional groups. The authors claimed that important nodes in a software system (high in-degree and out-degree) tend to act as attractors for new members that join an existing network. Thus, we strived to test if our proposed approach of transforming UML class diagrams into a complex network agrees with the finding of existing works.

A distribution fitting tool, EasyFit (MathWave, 2014) is used to fit the datasets into various probability distributions. Once the best fit probability is found, the probability density function (pdf), $f(x)$, is calculated to identify continuous random variables. The Generalized Pareto distribution and the Normal distribution have shown to be best-fitted for our datasets.

The pdf of Generalized Pareto distribution, $f_g(x)$, is defined in Eq. (4) (Hosking and Wallis, 1987). The parameters k, σ and μ denote the shape, scale, and location respectively.

$$f_g(x) = \begin{cases} \frac{1}{\sigma}\left(1 + k\frac{(x-\mu)}{\sigma}\right)^{-1-(1/k)} & k \neq 0 \\ \frac{1}{\sigma}\exp\left(-\frac{(x-\mu)}{\sigma}\right) & k = 0 \end{cases} \tag{4}$$

The scale parameter, σ, defines the height and spread of the distribution. The larger the σ value, the more spread out the distribution is.

The pdf of Normal distribution, $f_n(x)$, is defined in Eq. (5) (Stein, 1981). The parameters σ and μ denote the scale and location respectively, where σ represents the standard deviation while μ represents the mean.

$$f_n(x) = \frac{\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)}{\sigma\sqrt{2\pi}} \tag{5}$$

We started off the analysis by looking into all datasets as a whole. Two diagrams are generated for each of the graph-level metrics. First, a scatter plot in log–log scale is generated to show the frequency of the metric values and to identify if the distribution shows power law behavior. In the scatter plot of a power law distribution on log–log scale, the distribution tends to show right-skewed properties and the points lie approximately along a line (Concas et al., 2007). Finally, for each graph-level metric, we identified the best fit probability distribution using EasyFit tool and represented the results visually.

5.3. *Result of distribution fitting for all datasets*

In this section, we show the results of the computed graph-level metrics and the best fit probability distributions. Note that the metrics are computed based on all datasets as a whole. The results of each graph-level metric are analyzed and discussed with respect to its software maintainability and fault proneness. Eventually, based on the experimental results and analysis, we can achieve the research objective stated in RO2—*to investigate the correlations between graph-level metrics and the selected software quality attributes related to maintainability and reliability.*

5.3.1. *In-degree*

Fig. 8(a) shows that majority of the classes in the analyzed open-source software have in-degree of less than 5, with the mean value at 1.998. The figure also shows an almost linear behavior, which is the characteristic of power law distribution. The in-degree is best fitted with Generalized Pareto distribution shown in Fig. 8(b). The goodness of fit based on Kolmogorov–Smirnov (KS) test (Smirnov, 1948) is also shown in Fig. 8(b). However, it is to be noted that the KS test results show a relatively low significance level due to the large sample size, $N > 10{,}000$. This is a well-known issue discussed in the existing literature such that for a large sample size, goodness of fit test becomes sensitive to very small and insignificant deviation from a distribution (Tanaka, 1987; Bollen, 1990; Inman, 1994). The parameters k, σ and μ are 0.409, 1.258, and −0.125 respectively. The analysis shows that majority of the classes do not provide services to other classes. Most of the tasks are handled locally to promote loose coupling between classes. Although there are some classes that contain significantly higher in-degree values, those classes are typically utility classes that are designed to be reused frequently in the system. All in all, the observed behavior in terms of high modularity and loose coupling contributes toward improving the maintainability of the software.

5.3.2. *Out-degree*

The frequency plot of out-degree in log–log scale is shown in Fig. 9(a). The average out-degree of the analyzed open-source software is at 1.991. The maximum out-degree is roughly 44 times higher than the average value. Fig. 9(a) also exhibits the characteristic of a power law distribution. The results of data fitting, depicted in Fig. 9(b), have shown that out-degree is best fitted in Generalized

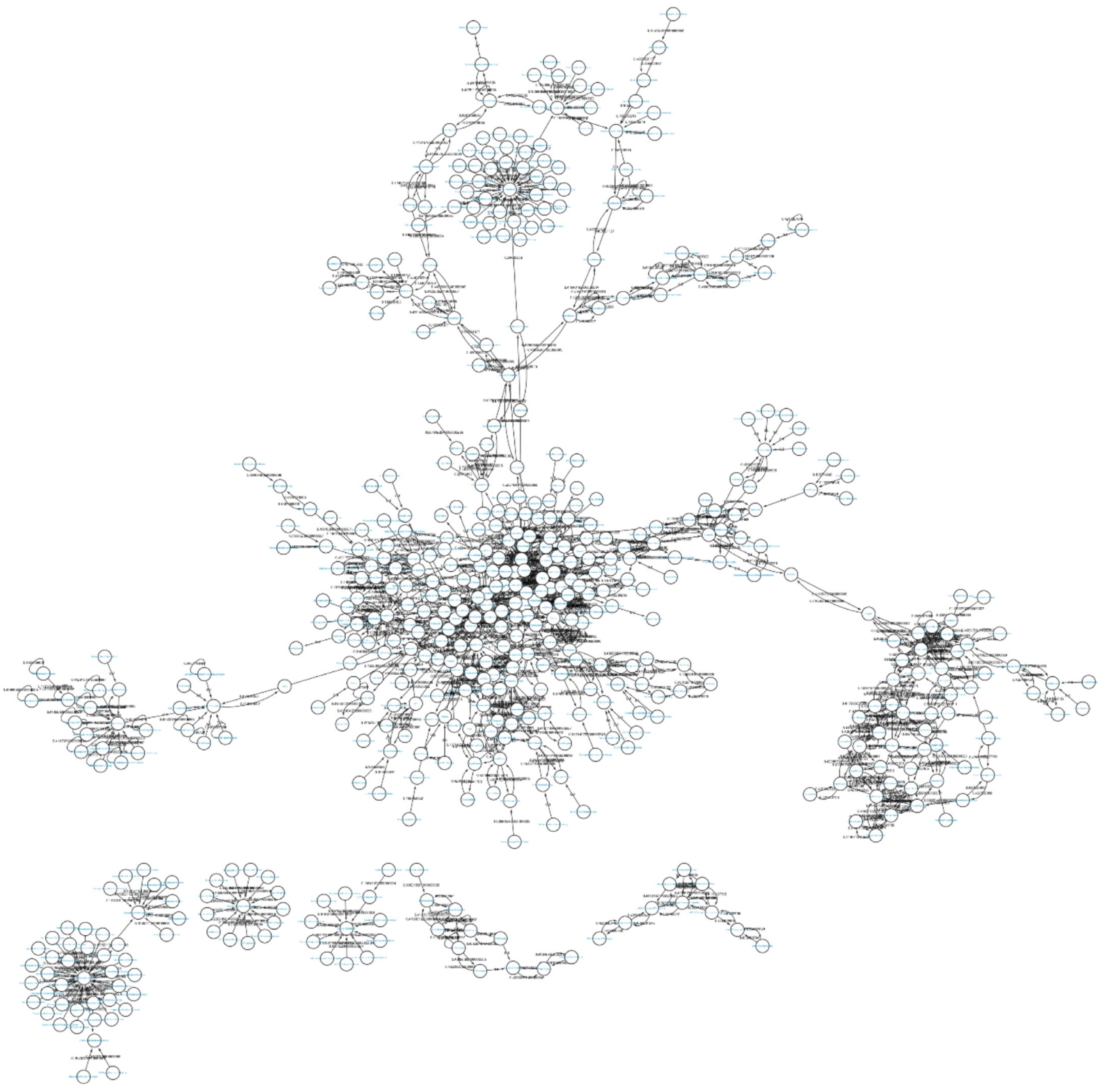


Fig. 7. Apache Synapse system modeled in a complex network using the proposed technique.

Pareto distribution, with parameters k, σ and μ at 0.2663, 1.161 and 0.420 respectively.

In the in-degree analysis, the maximum in-degree is roughly 90 times higher than the average in-degree. The maximum out-degree is roughly 44 times higher than the average out-degree, which is relatively lower when compared to in-degree. This observation shows several important behaviors of the analyzed software systems. Calling classes outside of their package should be minimized to avoid coupling. The work by Valverde et al. (2002) suggests that making use of large hubs, or in this context, nodes with high in and out-degree are bad software design practices, or also known as anti-patterns in existing software engineering literature. If the node becomes too complex, it might become a burden when maintaining the software. It is better to break large hubs into smaller and more modular components that focus on specific functionalities, as discussed in the work by Myers (2003). Thus, theoretically the maximum in-degree and out-degree should not deviate too much from its mean value. However, it is unavoidable in certain scenarios where important classes are being repeatedly reused.

5.3.3. *Average weighted degree*

Fig. 10(a) shows the frequency plot in log–log scale, where the mean average weighted degree is 1.071. The maximum average weighted degree derived from all datasets is around 80, which is about 75 times larger than the mean value. Similar to in-degree and out-degree, average weighted degree shows the characteristic of

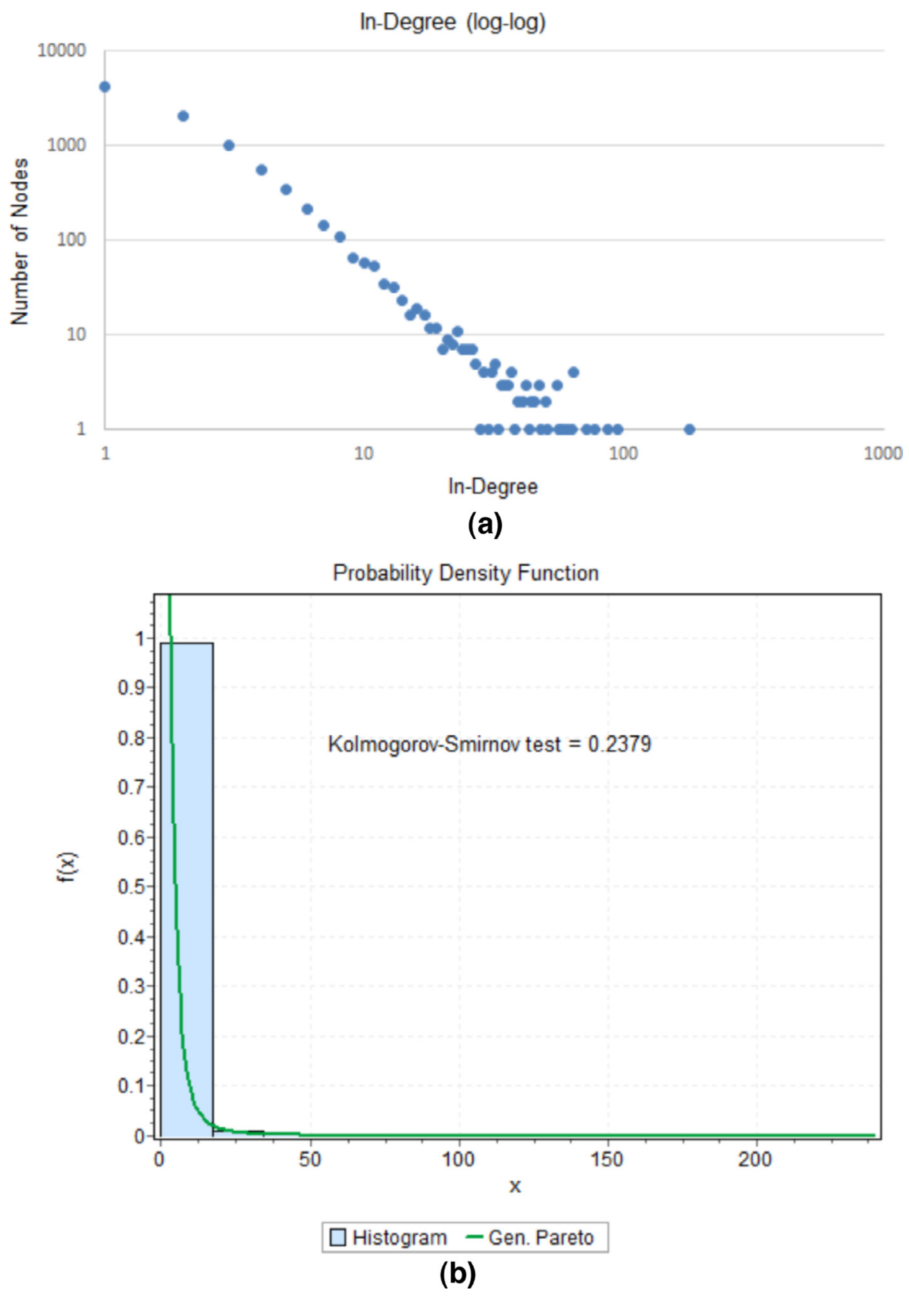


Fig. 8. In-degree (a) frequency in log–log scale, (b) fit into Generalized Pareto distribution.

power law distribution. Fig. 10(b) shows that the average weighted degree of all data can be modeled by the Generalized Pareto distribution. The parameters k, σ and μ are 0.402, 0.658, and -0.002 respectively. The observation shows that majority of the analyzed software systems have low average weighted degree, where the probability of classes having a value of less than 5 is very high. Although there are a few nodes with a very high average weighted degree, these classes are usually utility or main system classes that supply services to other classes. Examples of utility classes that exhibit a high average weighted degree are WikiEngine.java of Apache JSPWiki (77.589), Hudson.java of Hudson (51.75), Field.java of jOOQ (70.535) and Logger.java of Struts (77.23).

5.3.4. *Average shortest path length*

The frequency plot in histogram for average shortest path length is shown in Fig. 11(a). The mean average shortest path for all datasets is around 3.8 steps, which agrees with Valverde and Solé (2003) who found that the average shortest path in software is less than six steps. The log–log frequency plot for average shortest path length is not included because we found that average shortest path length is best fitted in Normal distribution with parameters σ $\sigma = 2.816$ and $\mu = 3.877$ (Fig. 11(b)). Fig. 11(a) shows that the diagram is positively skewed, where the distribution is concentrated on the left of the figure. Typically, power law behavior is not shown in log–log plot for Normal distributed data. This result demonstrates that most classes in object-oriented software can communicate with each other easily. Low average shortest path length also contributes toward high response capability of the analyzed software, especially Apache Deltaspike and Apache Synapse, because they are usually deployed on a web-based environment.

5.3.5. *Average clustering coefficient*

Clustering coefficient is a graph-level metric that measures if a given node's neighbors are neighbors among themselves. Average

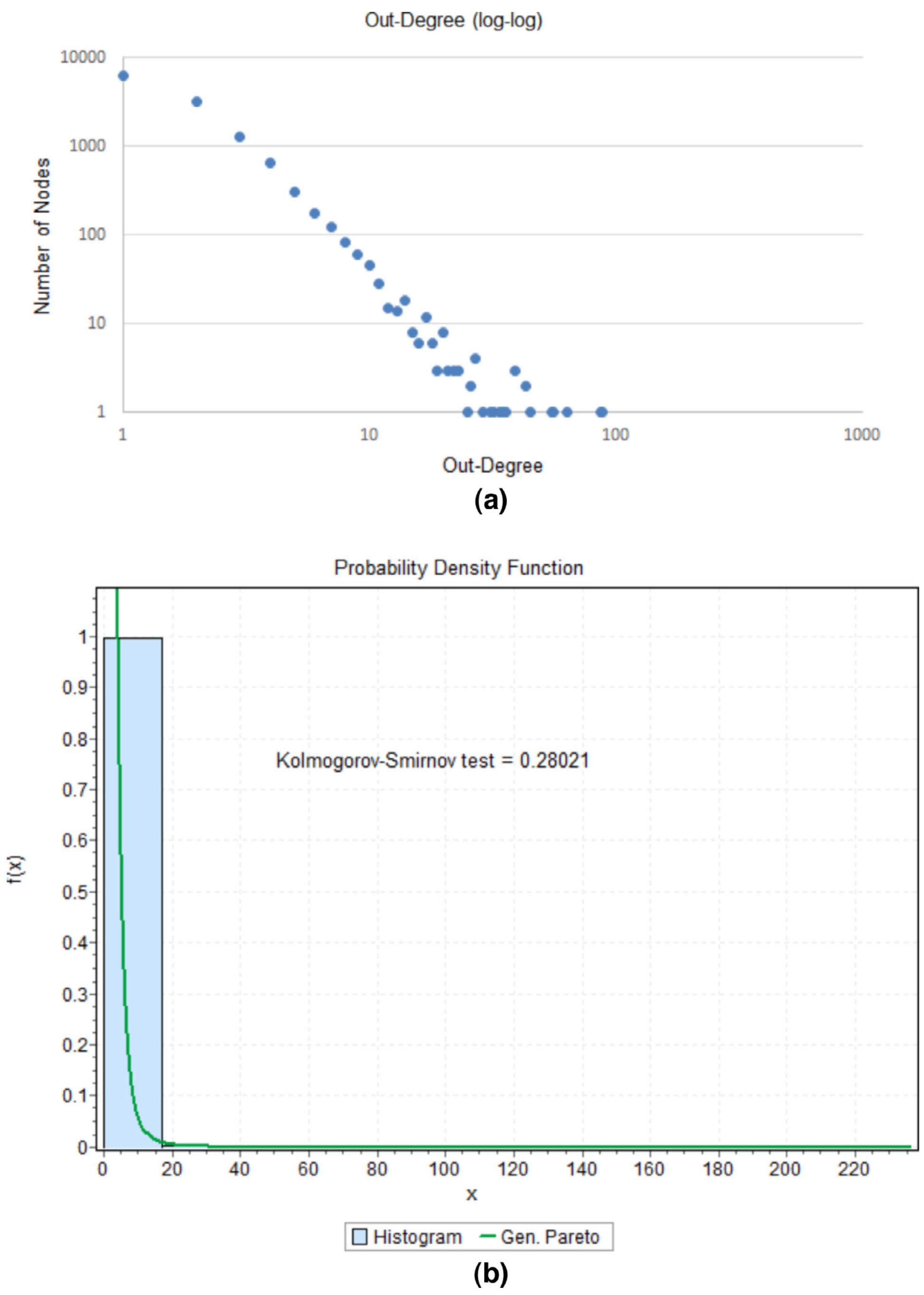


Fig. 9. Out-degree (a) frequency in log–log scale, (b) fit into Generalized Pareto distribution.

clustering coefficient provides the average score of clustering coefficients of all nodes for the whole network. If the average clustering coefficient of the network is equal to 1, the network is called a clique where each pair of nodes is connected by an edge. Analyzing the frequency plot of clustering coefficient is difficult because it does not translate directly into any object-oriented behavior. Thus, we adopt a different approach to analyze the average clustering coefficient.

As mentioned earlier in Section 2, small world properties of a network can be determined by looking into the average shortest path length and clustering coefficient. A network with a high clustering coefficient has a strong characteristic of small world property. Many researchers have found that the average clustering coefficients of object-oriented software are much higher than those of random networks constructed based on the same node property (Concas et al., 2007; Louridas et al., 2008; Potanin et al., 2005; Pang and Maslov, 2013). The findings suggest that software systems possess a higher degree of cohesion with respect to random networks.

The work by Newman (2006) has further proven that clustering coefficients of real-world networks should be higher than what would be expected if edges were randomly placed, using a 'graph modularity' measurement. Newman shows that in real-world networks, the number and density of interactions among nodes belonging to a community are higher than expected in a random network of the same size.

In order to test this particular behavior, we used a Cytoscape plugin developed by Mcsweeney (2008) which is capable of randomizing an existing network. The algorithm used to generate a random network from an existing one works as follows:

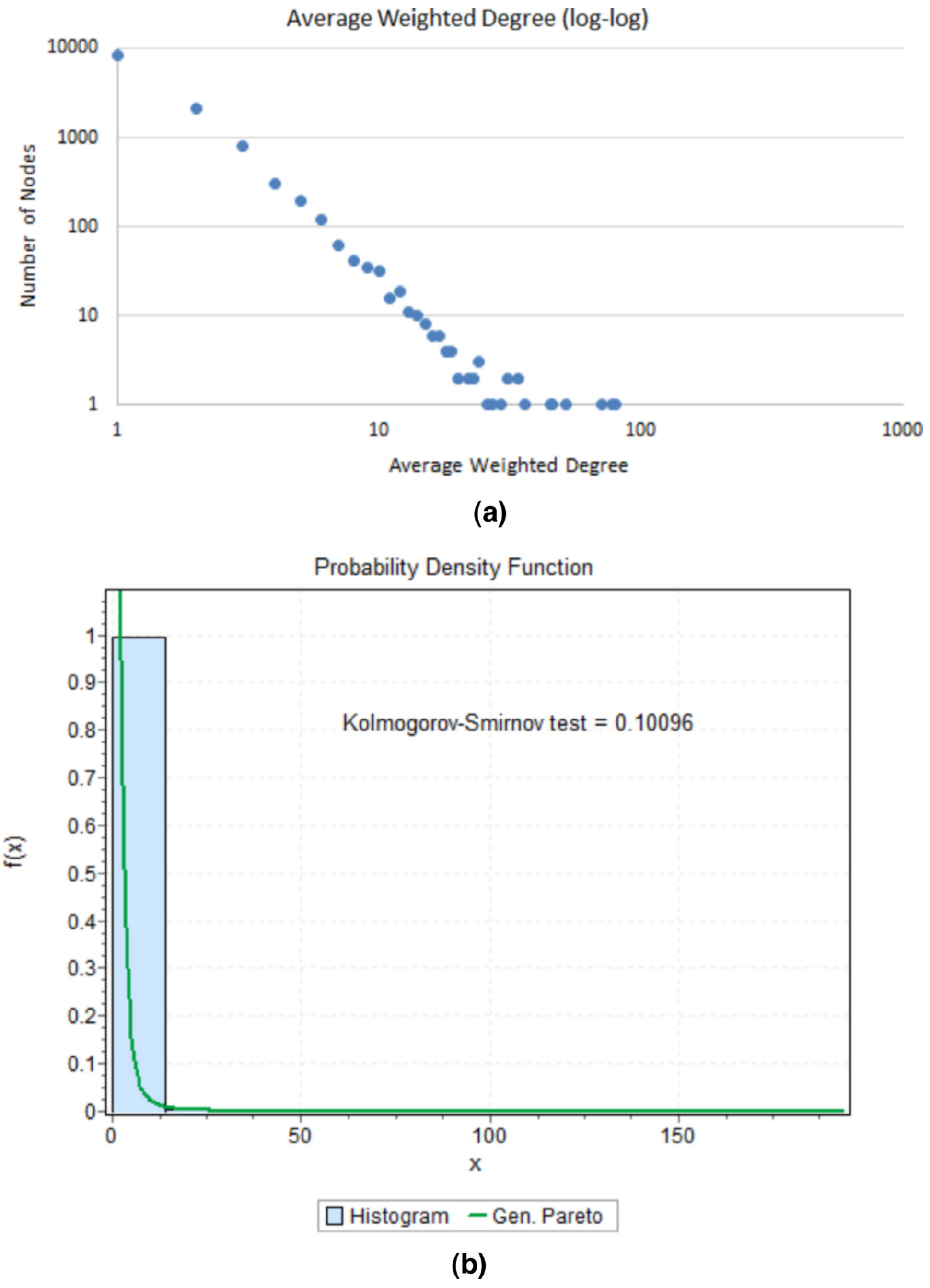


Fig. 10. Average weighted degree (a) frequency in log–log scale, (b) fit into Generalized Pareto distribution.

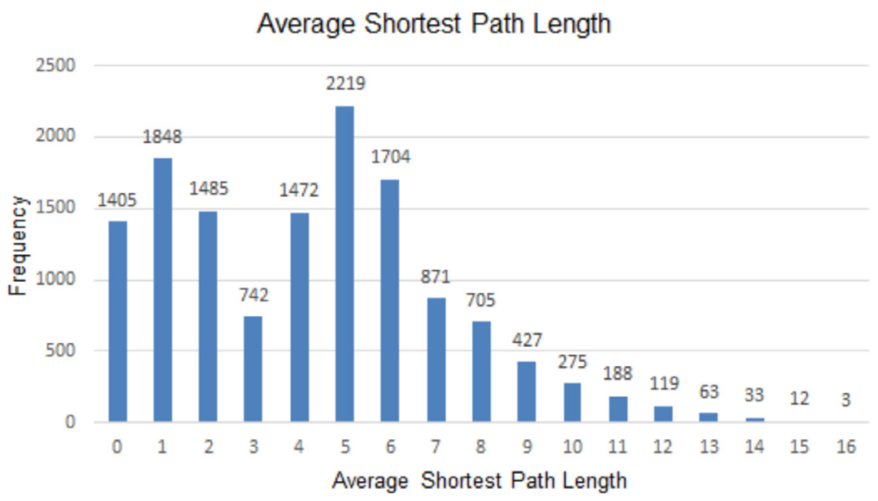
1. A random edge (u, v) is selected from the network.
2. A second random edge (s, t) is selected with the constraints that:
   - $u \neq v \neq s \neq t$
   - (u, t) and (s, v) do not already exist in the network.
3. Edges (u, v) and (s, t) are removed and edges (u, t) and (s, v) are inserted into the network.
4. Repeat steps 1–3 n times.

First, we calculate the average clustering coefficients of all datasets as a whole. Next, based on the nodes' properties of our datasets (such as number of nodes and direction of edges), we generate 100 random networks and calculate their average clustering coefficients based on the plugin developed by Mcsweeney. Finally, the average clustering coefficients of all the real datasets are compared against the average clustering coefficients of all the random networks.

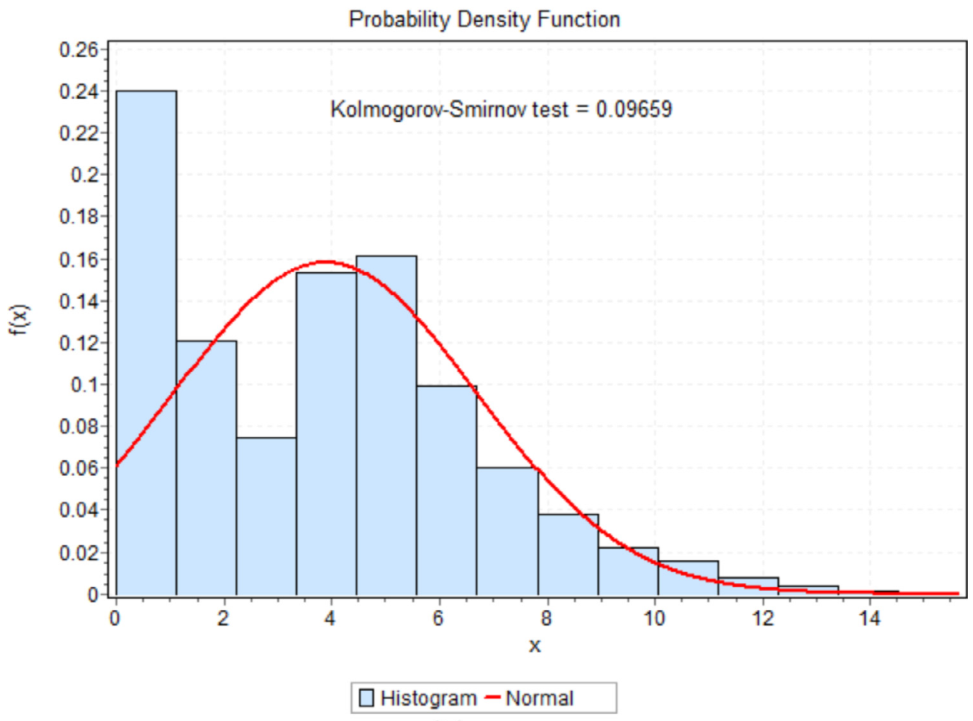
We measured and found that the average clustering coefficient of the object-oriented software network is at 0.048, while the average metric value for 100 random networks is 0.0012. Combined with the observation from average shortest path length, we can convincingly claim that the constructed network using the proposed technique does adhere to small world properties commonly found in the existing literature.

5.3.6. *Betweenness centrality*

Fig. 12(a) shows the distribution of betweenness centrality for all analyzed software systems in log–log scale. Note that the betweenness centrality for a given node n is normalized by dividing by the number of node pairs in the network, excluding n. Thus, the values range from 0 to 1. The scatter plot shows that majority of the nodes have value of less than 0.1, which indicates that they do not control



(a)



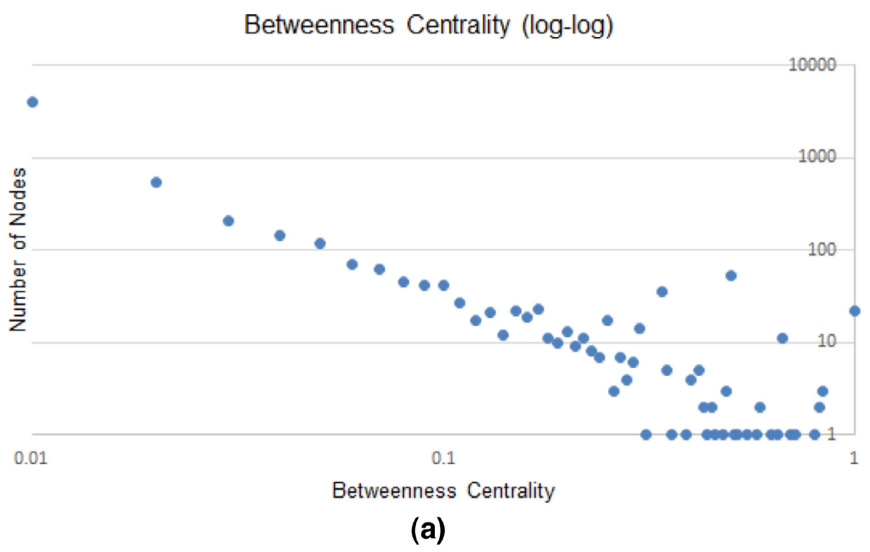
(b)

Fig. 11. Average shortest path length (a) frequency in histogram, (b) fit into Normal distribution.

the flow of information in the network. Fig. 12(a) suggests that power law characteristic is present. The data are best fitted in Generalized Pareto distribution as shown in Fig. 12(b), where the parameters k, σ and μ are 0.887, 0.0016, and $-6.63E-4$. This observation reveals that most of the classes in object-oriented software do not have dominant power that dictates the flow of information and data. Removing certain components from the software will have minimal impact on the structure stability. Although there are a few nodes with very high centrality value, those nodes are normally interface classes that act as the 'authority classes', as discussed by the work by Ovatman et al. (2011). Ovatman et al. found that classes and groups of classes in UML class diagrams show distinctive recurring patterns in terms of dependencies between each other and to other classes. 'Authority classes' is one of the patterns where a large number of classes are coupled to.

6. Comparative analysis

Besides the distribution fitting approach, we have also analyzed the empirical distribution of the datasets by looking into the respective quartiles. This research is carried out in an exploratory manner, where we intend to discover the common statistical behavior of object-oriented software systems when modeled in complex networks. Although we have introduced the concept of distribution fitting in Section 5.2.2, certain patterns and behavior might not be visible using the aforementioned approach. Thus, in this section, we attempt to analyze the empirical distribution of the datasets by examining the quartiles using boxplot. The boxplot analysis is one of the statistical methods used to visually identify patterns that may otherwise be hidden in a dataset (Williamson et al., 1989; Hyndman and Fan, 1996). This will allow us to perform a comparative



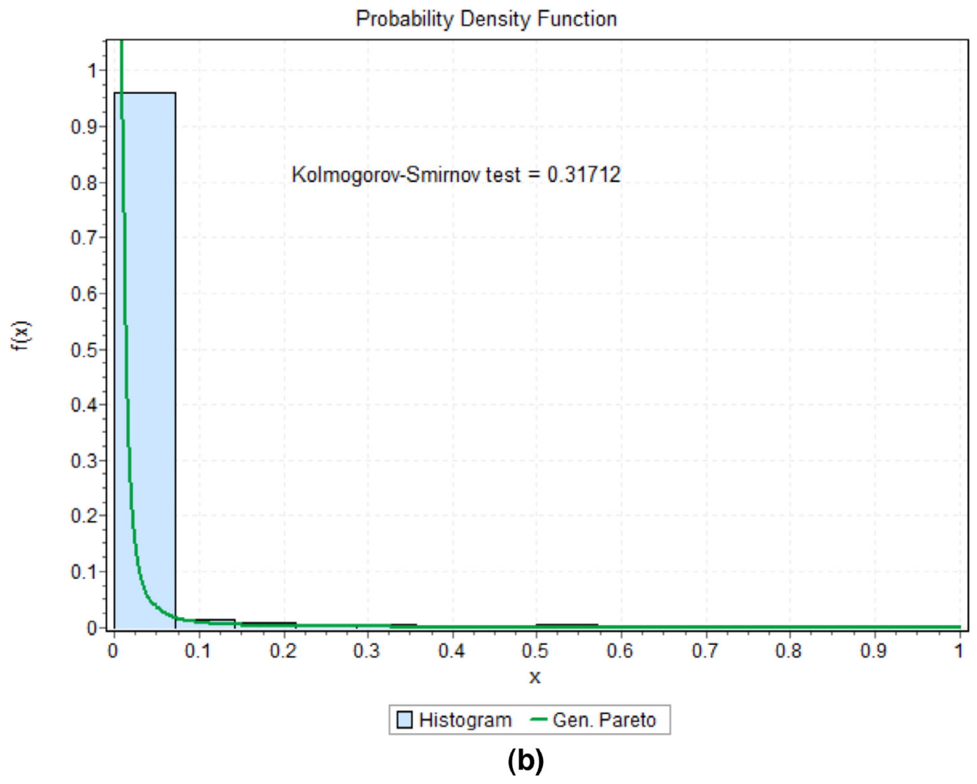


Fig. 12. Betweenness centrality (a) frequency in log–log scale, (b) fit into Generalized Pareto distribution.

analysis of the selected software systems and eventually tackle the research objective discussed in RO3—*to identify the effectiveness of the proposed approach when applied to object-oriented software systems with different levels of maintenance effort*.

In Section 5, we discussed SQALE rating method to estimate the maintenance efforts of the chosen open-source software systems. By combining all the data of A-rated projects, we can produce a boxplot and subsequently identify the median, the approximate quartiles, spread, and symmetry of the distribution (Williamson et al., 1989). We can then compare the results with B-rated and C-rated projects in order to compare the differences in statistical behavior for these groups of datasets.

Fig. 13 depicts the boxplots of all the graph-level metrics for A-rated, B-rated, and C-rated software systems except for betweenness centrality. The boxplot of betweenness centrality is separated as shown in Fig. 14 due to the difference in the scale of values.

Table 4 presents a summary of analysis including first quartile, median, third quartile, interquartile range, and whiskers of the boxplots from Fig. 13. We discovered a surprising observation where there are no difference between the in-degree and out-degree boxplots of A-rated, B-rated, and C-rated software systems. The boxplots of in-degree are positively skewed due to the power-law behavior observed earlier in the frequency distribution plot. The out-degree shows lesser variability, with an interquartile range of 1. Low interquartile range of out-degree is consistent with our observation in Section 5.3.2, where we observed that most of the nodes have a similar out-degree value except for a few nodes. This shows that developers of A-rated, B-rated and C-rated software systems adhere to

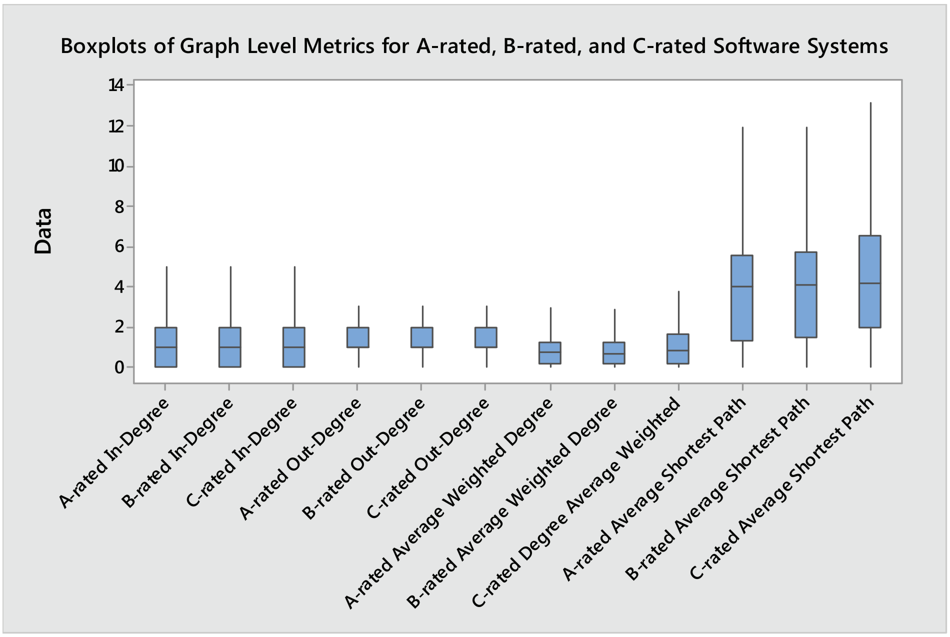


Fig. 13. Boxplots of in-degree, out-degree, average weighted degree, and average shortest path for A-rated, B-rated, and C-rated software systems.

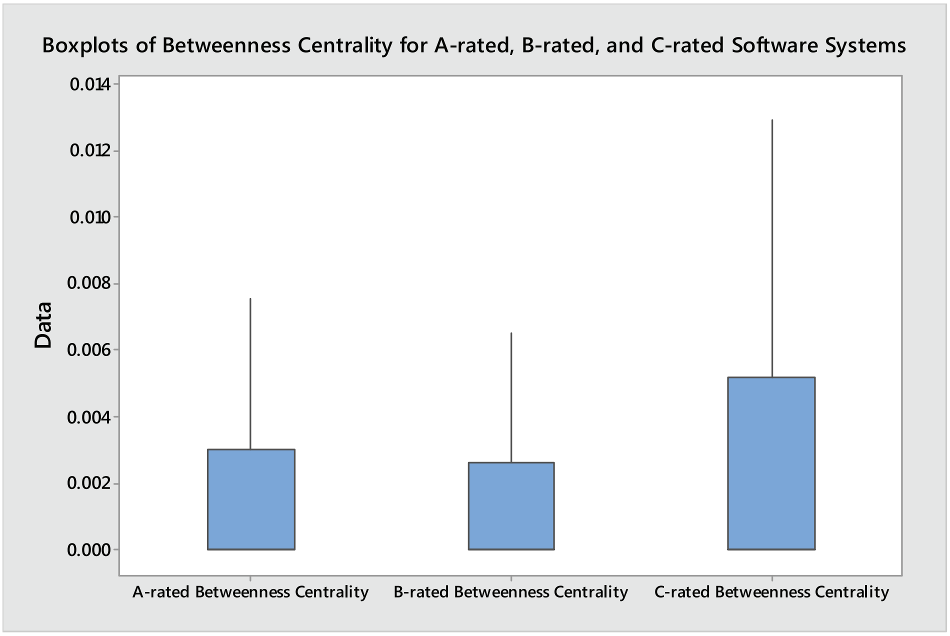


Fig. 14. Boxplots of betweenness centrality for A-rated, B-rated, and C-rated software systems.

Table 4
Analysis of boxplots from Fig. 13.

Metrics	First quartile	Median	Third quartile	Interquartile range	Whiskers
A-rated in-degree	0	1	2	2	0, 5
B-rated in-degree	0	1	2	2	0, 5
C-rated in-degree	0	1	2	2	0, 5
A-rated out-degree	1	1	2	1	0, 3
B-rated out-degree	1	1	2	1	0, 3
C-rated out-degree	1	1	2	1	0, 3
A-rated average weighted degree	0.161	0.75	1.282	1.121	0, 296
B-rated average weighted degree	0.158	0.701	1.232	1.075	0, 2.844
C-rated average weighted degree	0.157	0.810	1.618	1.461	0, 3.799
A-rated average shortest path	1.333	4.039	5.562	4.229	0, 11.898
B-rated average shortest path	1.5	4.120	5.690	4.190	0, 11.933
C-rated average shortest path	2	4.187	6.5	4.5	0, 13.098

Table 5
Analysis of boxplots from Fig. 14.

Metrics	First quartile	Median	Third quartile	Interquartile range	Whiskers
A-rated betweenness centrality	0	0	0.00301	0.00301	0, 0.00754
B-rated betweenness centrality	0	0	0.00261	0.00261	0, 0.00651
C-rated betweenness centrality	0	0	0.00517	0.00517	0, 0.0129

the high modularity concept when developing and updating the software systems, as discussed in the work by Myers (2003).

The boxplots of average weighted degree are also positively skewed due to the power-law behavior. When the edges and nodes are weighted, we found that there is a slight variation from the observed quartile, where the whisker of C-rated software systems is slightly higher at 3.799. This indicates that the coupling strength of C-rated software systems is relatively higher, which might contribute toward their high maintenance efforts. In terms of average shortest path, the whiskers of B-rated and C-rated software systems are slightly higher at 11.933 and 13.098, which could indicate that several classes have high communication costs. Upon further investigation, we found that several classes in Apache Log4J, namely CompositeTriggeringPolicy, AbstractRolloverStrategy, DatePatternConverter, RolloverStrategy, ExitTag and EntryTag, have an average shortest path length of 13–14 steps. These classes contain methods that depend on the Logger class, which is located separately in another package.

The boxplots of betweenness centrality for A-rated, B-rated, and C-rated software systems are shown in Fig. 14, along with the analysis in Table 5.

Betweenness centrality measures the number of shortest paths that pass through a selected class. Classes with high betweenness centrality signify that they are important because they usually act as the communication bridge. On the flip side, these classes are highly error prone and can easily propagate bugs due to their behavior (Concas et al., 2011). As can be observed from Table 5, C-rated software systems have much higher value of betweenness centrality when compared to A-rated and B-rated software systems. Upon further investigation, we found that the ApacheDS project contains a few utility classes that possess high betweenness centrality. These classes are AvlNode, Marshaller, KeyIntegrityChecker, NtpService, PasswordPolicyConfiguration, PasswordValidator, NtpMessage, NtpMessageModifier, and LdapServer. These classes should be given more attention as they are highly error prone.

- Comparison of node weighted and edge weighted approach.

In Section 4, we proposed Eq. (3) as a means to measure communicational cohesion-based weights by looking into the complexity of classes and relationships.

$$Weight_{(R_{i\to j})} = \left(H_{R_{i\to j}} \times \alpha\right) + \left[\left(1 - Comp_{(D_j)}\right) \times \beta\right] \quad (3)$$

The approach is based upon a hybrid of node (class) weighted and edge (relationship) weighted approach similar to the work presented by Ma et al. (2010). There are different strategies to represent the weights of nodes or edges and we believe that there are certainly advantages and disadvantages for each approach. Thus, in this subsection, we expand the comparative analysis discussed before this by contrasting the performance of different representation strategies, i.e. node weighted and edge weighted approaches.

The first operand of Eq. (3) represents the complexity of a relationship, while the second operand represents the complexity of the terminus class linked by the associated relationship. In order to compare the performance of different representation strategies, we perform the following steps:

1. For each software system, recalculate the weights of edges by using only the first operand of Eq. (3), $(H_{R_{i\to j}} \times \alpha)$ and only the second operand of Eq. (3), $[(1 - Comp_{(D_j)}) \times \beta]$.
2. Reconstruct the weighted complex network. Since there are two representations of weights, there will be two weighted complex networks for each software—node weighted only network and edge weighted only network.
3. Recalculate the value of average weighted degree for each network.
4. Perform a comparative analysis of different representation strategies by grouping the software based on their SQALE rating.

Fig. 15 depicts the result of our analysis and the details of the analysis are presented in Table 6.

The first three boxplots represent the average weighted degree calculated using the exact Eq. (3). The ‘Node Only’ and ‘Edge Only’ boxplots, on the other hand, represent the calculation based on class complexity $[(1 - Comp_{(D_j)}) \times \beta]$ and relationship complexity $(H_{R_{i\to j}} \times \alpha)$ respectively.

From the ‘Node Only’ boxplots, we can observe that the median values for B-rated and C-rated software systems are much higher compared to A-rated software systems. Several reasons contributed toward this observation. First, when calculating the weights for classes related with unidirectional relationships (generalization and realization), only the complexity of parent or supplier class is considered. Thus, for interface or utility classes that are heavily reused, the occurrence of duplicate weightage increases significantly. Second, we found that these groups of highly reused classes tend to be more complex and more LOC in B-rated and C-rated software systems. For example, AbstractBTreePartition.Java class in ApacheDS project which contains 1980 LOC and WMC of 399 is a very complex interface class. Another example is the Registries.Java class in ApacheDS which contains 1711 LOC and WMC of 392. The Registries.Java acts as a utility class and it is implemented by many other classes.

As for the ‘Edge Only’ boxplots, the occurrence of duplicate weightage is even more distinguishable because we only consider the type of relationship between two classes. Recall that the weights of relationships are based on the ordinal scale shown in Table 2. Thus, the variability of weightage is very limited and very little information can be extracted from the boxplot. We are unable to extract any useful information related to software maintainability and fault proneness based on the network represented with ‘Edge Only’ weightage.

All in all, we observe that the ‘Node Only’ strategy to represent communicational cohesion-based weight can be useful to identify highly reused and fault-prone classes. The comparative analysis in Fig. 15 has clearly shown that software systems with a higher level of maintenance effort tend to have a higher average weighted degree. Software with a higher average weighted degree indicates that most of the software components are entangled and depend on each other (Concas et al., 2007; Louridas et al., 2008). Thus, more efforts are needed to maintain this group of software systems. One disadvantage of ‘Node Only’ strategy is that we are unable to clearly distinguish which class and relationship have higher communicational cohesion over other classes because it is assumed that all relationships are even. Duplicate of weightage occurs very often. As for the ‘Edge Only’ strategy, we found that it does not provide much useful information toward understanding the maintainability and reliability of software.

Table 6
Analysis of boxplots from Fig. 15.

Metrics	First quartile	Median	Third quartile	Interquartile range	Whiskers
A-rated	0.173	0.813	1.495	1.322	0, 3.476
B-rated	0.158	0.701	1.232	1.075	0, 2.844
C-rated	0.157	0.810	1.618	1.461	0, 3.799
A-rated (Node Only)	0.234	0.5	1.485	1.251	0, 3.360
B-rated (Node Only)	0.285	0.767	1.498	1.213	0, 3.317
C-rated (Node Only)	0.499	1	2.258	1.758	0.125, 4.892
A-rated (Edge Only)	0.35	0.55	1.05	0.7	0, 2.1
B-rated (Edge Only)	0.3	0.6	0.95	0.65	0, 1.9
C-rated (Edge Only)	0.35	0.55	0.9	0.55	0.1, 1.7

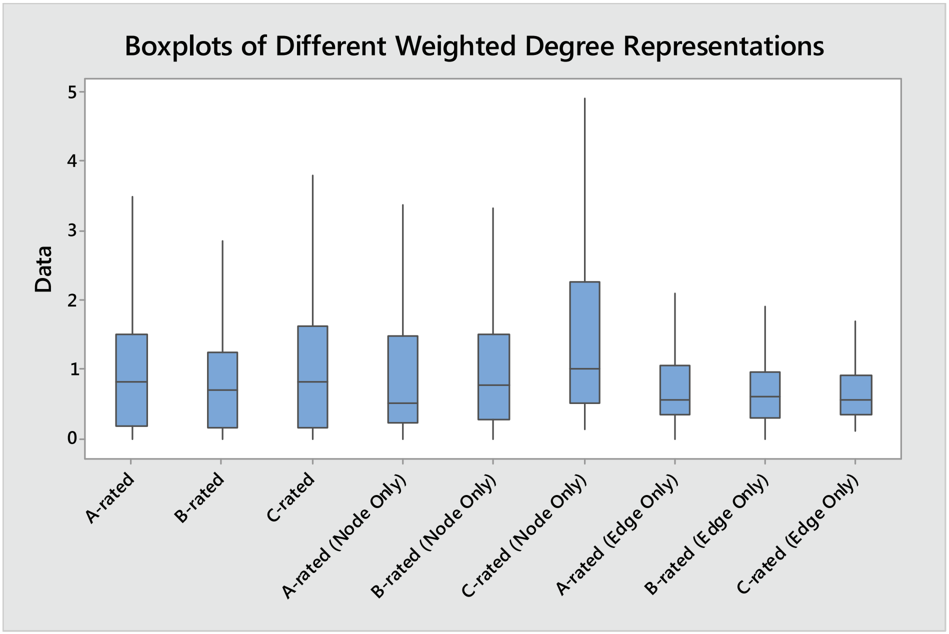


Fig. 15. Boxplots of different weighted degree representations for A-rated, B-rated, and C-rated software systems.

7. Discussion

In Section 3, three research objectives were proposed. The first research objective, RO1, outlines the goal to identify appropriate measurement constructs that are capable of quantifying maintainability and reliability of object-oriented software systems modeled with complex networks. Based on the discussed literature, we found that weighted and directed network is more suitable to be used in the context of software engineering because not all software features are symmetrical in nature. Besides that, most of the existing studies tend to use the frequency of method calling as a basis of modeling weighted networks, although it is very much language-dependent. Subsequently, we propose a unique weighting function using several well established software metrics based on the work of Ma et al. Software systems are first converted into UML class diagrams in order to standardize the transformation rules. The proposed weighting function focuses on the complexity of UML classes and their associated relationships.

The second research objective, RO2, discusses about investigating the correlations between the selected graph-level metrics and the maintainability as well as reliability of software systems. In Section 5, we use a distribution fitting tool called the Easyfit to identify the best fit distribution of all datasets, followed by analysis of the observation. It was discovered that most of the selected graph-level metrics show the power-law behavior, which is a typical characteristic of complex networks. A large majority of the test subjects are shown to have low in-degree and out-degree. From a software engineering perspective, we learnt that most of the classes from the pool of test subjects are designed to be focused on their own functionalities and easy to maintain. Classes that are frequently reused or called (i.e. utility classes and interface classes) can be easily identified through the inspection of distribution fitting diagrams in Figs. 8–10. The details of the observation are discussed throughout Section 5.

In order to validate the proposed weighting function, we compare the statistical pattern of the selected graph-level metrics with the aid of SQALE rating, where RO3 comes into the picture. We group the datasets into three different categories, namely A-rated, B-rated, C-rated software systems. If the proposed weighting function and the selected graph-level metrics are able to represent the maintainability and reliability of software systems, the resulting comparison should exhibit a certain degree of correlation.

Based on the results shown in Section 6, we found that when software systems are grouped according to their maintenance effort, their statistical patterns are consistent with the existing literature. For example, the average weighted degree and shortest path length of B-rated and C-rated software systems are relatively higher than those of A-rated software systems. A higher weighted degree is associated with high coupling, while a high average shortest path length signifies poor communication between classes. Both observations contribute toward low maintainability and reliability of software systems, which is consistent with our experimental setup. This has demonstrated that the proposed weighting function is able to distinguish the degree of maintenance efforts of the analyzed software systems, and eventually provide a concrete answer for RO1, RO2, and RO3.

From the software engineering perspective, several implications can be drawn from the experimental results. First, we have shown that UML diagram can be an effective tool to aid in the modeling of a software-based complex network. The proposed weighting function has shown to be able to successfully quantify the maintainability and reliability of software systems, and provide an alternative to the conventional techniques that count the frequency of method interactions.

Furthermore, classes that violate common software design principles, or those that are more prone to bugs and errors can be easily identified with the aid of graph-level metrics and complex networks. For instance, by using the betweenness centrality metric, we found that HasCurrentMarkup.Java and HasIDBindingAndRendered of Apache Tobago project are very vulnerable toward bug propagation. These two classes possess a very high betweenness centrality value, which means that plenty of communications between classes (including passing of variables or parameters) need to go through them. From the software engineering perspective, unless they are purposely designed as an interface or mediator class, it is risky to have multiple classes that dictate the flow of communications because the failure or removal of these classes will cause a system-wide service interruption.

Besides that, the results shown in Fig. 15 and Table 6 have shown that a hybrid of node and edge-weighted strategy is more appropriate because it is able to represent the dynamic interactions between software features. Both the classes and their interactions play equally important roles to quantify the communicational cohesion of classes. Furthermore, the proposed hybrid weighting strategy can also be used in forward engineering phase to evaluate the effectiveness of the software design.

8. Threats to validity

This section discusses threats to the internal and external validity. Countermeasures against the threats to the validity were taken and are described below.

We examined the internal validity with respect to three aspects, which are the regression toward the mean, the selection of subjects, and the confounding variables. With respect to the first aspect, we performed two analysis where the first one involves distribution fitting of each graph-level metric, followed by analyzing the empirical distribution of data using boxplot method. We had also grouped the projects based on their maintenance effort and performed a thorough comparative analysis.

To address the threat from subject selection, we chose 40 different software systems, varying according to class counts. We categorized the projects into four groups—projects with less than 250 classes, between 250 and 500 classes, between 500 and 1000 classes, and more than 1000 classes. We believe that the selected software systems are able to reflect some representative class count distribution on the population of open-source Java software systems available in the market. The chosen software are well-known projects that are actively developed and maintained by the open-source community. Although we are unable to guarantee that these software systems are the best examples, good software should exhibit similar behavior when analyzed from a graph-level abstraction.

The choice of code-level, system-level, and graph-level metrics used in this study might impose the threat of confounding variables. The chosen code-level and system-level metrics are WMC and LOCM4 respectively. Both metrics are originated from the CK object-oriented metrics suite and proven to be complimentary (Chidamber and Kemerer, 1994). An in-depth analysis of CK metrics is presented in Section 4.1.2, which discussed the development of CK metrics in the past decade, along with its effectiveness in predicting software maintenance cost and software bug prediction. Besides that, we introduced the preferences and risk tolerance parameters to provide more flexibility in obtaining the values of WMC and LCOM4. The chosen graph-level metrics are selected based on their interpretation toward the behavior of object-oriented software systems. The details of explanations have been discussed in Section 4.1.

In order to mitigate the threat to construct validity, we selected measurement constructs that focus on measuring the maintainability and reliability of software systems. Besides that, we also introduced the SQALE rating to estimate the maintenance cost of the selected software systems. Furthermore, we performed a comparative analysis in Section 6 by grouping the software systems into three categories, A-rated B-rated, and C-rated, depending on their maintenance efforts.

9. Conclusion and future work

Existing work that models software systems using complex networks typically focuses on analyzing the complexity of software at a specific level of detail. Furthermore, less attention is given to measure the weights of edges, where weighted complex networks are used to model software systems. This study introduces three levels of metrics, namely code-level, system-level, and graph-level metrics that aim to fill in the research gap.

The 40 open-source software were analyzed using the proposed approach by converting the software into complex networks. We focus on measuring and analyzing the maintainability and reliability of the selected software systems. Three research objectives are discussed in Section 3. To achieve RO1, we performed a thorough literature review on existing software metrics that are capable of representing the maintainability and reliability of software system. WMC and LCOM4 are selected to calculate the weightage of edges. As for RO2, we performed distribution fitting of all graph-level metrics in Section 5 in order to identify the common patterns and correlations between those metrics and the chosen software quality attributes. Finally, a comparative analysis based on SQALE rating is performed in Section 6 in order to achieve RO3. All in all, we found that a hybrid of node and edge weighted strategies is more suitable for modeling software systems using complex networks. We have to emphasize that our work is only focusing on two software quality attributes, namely the maintainability and reliability.

Future work can be considered by including more software quality attributes when converting the UML class diagrams into complex networks. Furthermore, when converting source code into UML class diagram, we took a simple approach to identify aggregation and composition relationships. Additional work can be considered by looking into a formal way of converting those UML class diagram notation. Besides that, further work to correlate the graph-level metrics with a more direct measurement of maintenance effort, for instance, by measuring changes and issues of software in multiple releases can be considered. Measuring the frequency of changes between different releases of software systems can be a reliable way to measure the maintainability and reliability of software systems, such that the more changes that are required to address a bug, the greater the maintenance effort.

Acknowledgments

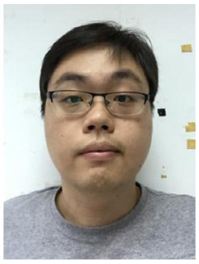
This work is carried out within the framework of a research project supported by eScienceFund, funded by Ministry of Science, Technology and Innovation (MOSTI), Malaysia under grant no. 01-01-03-SF0851.

References

Abreu, F.B., Carapuça, R., 1994. Object-oriented software engineering: measuring and controlling the development process. In: Proceedings of the Fourth International Conference on Software Quality.

Aier, S., 2006. How clustering enterprise architectures helps to design service oriented architectures. In: IEEE International Conference on Services Computing, 2006. SCC '06, pp. 269–272.
Aier, S., Schönherr, M., 2007. Model driven service domain analysis. In: Georgakopoulos, D., Ritter, N., Benatallah, B., Zirpins, C., Feuerlicht, G., Schoenherr, M., Motahari-Nezhad, H. (Eds.), Service-oriented Computing ICSOC 2006. Springer, Berlin / Heidelberg, pp. 190–200.
Amaral, L.A.N., Scala, A., Barthélémy, M., Stanley, H.E., 2000. Classes of small-world networks. Proc. Natl. Acad. Sci. 97, 11149–11152.
Anquetil, N., Lethbridge, T.C., 1999. Experiments with clustering as a software remodularization method. In: Proceedings of the Sixth Working Conference on Reverse Engineering 1999, pp. 235–255.
Bansiya, J., Davis, C.G., 2002. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. 28, 4–17.
Barabási, A.-L., Albert, R., 1999. Emergence of scaling in random networks. Science 286, 509–512.
Barabási, A.-L., Albert, R., Jeong, H., 2000. Scale-free characteristics of random networks: the topology of the world-wide web. Phys. A: Stat. Mech. Appl. 281, 69–77.
Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. 22, 751–761.
Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E., 2006. Understanding the shape of Java software. SIGPLAN Not. 41, 397–412.
Beiyang, W., Jinhu, L., 2012. Modelling complex software systems via weighted networks. In: 10th World Congress on Intelligent Control and Automation (WCICA), 2012, pp. 3533–3537.
Binkley, A.B., Schach, S.R., 1998. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In: Proceedings of the 20th International Conference on Software Engineering. IEEE Computer Society, Kyoto, Japan, pp. 452–455.
Bird, C., Pattison, D., D'Souza, R., Filkov, V., Devanbu, P., 2008. Latent social structure in open source projects. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, Atlanta, Georgia, pp. 24–35.
Bollen, K.A., 1990. Overall fit in covariance structure models: two types of sample size effects. Psychol. Bull. 107, 256.
Briand, L.C., Wüst, J., Ikonomovski, S.V., Lounis, H., 1999. Investigating quality factors in object-oriented designs: an industrial case study. In: Proceedings of the 21st International Conference on Software Engineering. ACM, Los Angeles, California, USA, pp. 345–354.
Briand, L.C., Labiche, Y., Yihong, W., 2001. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In: Proceedings of the 12th International Symposium on Software Reliability Engineering, 2001. ISSRE 2001, pp. 287–296.
Briand, L.C., Labiche, Y., Yihong, W., 2003. An investigation of graph-based class integration test order strategies. IEEE Trans. Softw. Eng. 29, 594–607.
Bullmore, E., Sporns, O., 2009. Complex brain networks: graph theoretical analysis of structural and functional systems. Nat. Rev. Neurosci. 10, 186–198.
Chatzigeorgiou, A., Melas, G., 2012. Trends in object-oriented software evolution: investigating network properties. In: 34th International Conference on Software Engineering (ICSE), 2012, pp. 1309–1312.
Chidamber, S.R., Darcy, D.P., Kemerer, C.F., 1998. Managerial use of metrics for object-oriented software: an exploratory analysis. IEEE Trans. Softw. Eng. 24, 629–639.
Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20, 476–493.
Chong, C.Y., Lee, S.P., Ling, T.C., 2013. Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach. Inf. Softw. Technol. 55, 1994–2012.
Cilibrasi, R.L., Vitanyi, P.M.B., 2007. The Google similarity distance. IEEE Trans. Knowl. Data Eng. 19, 370–383.
Concas, G., Marchesi, M., Murgia, A., Tonelli, R., Turnu, I., 2011. On the distribution of bugs in the eclipse system. IEEE Trans. Softw. Eng. 37, 872–877.
Concas, G., Marchesi, M., Pinna, S., Serra, N., 2007. Power-laws in a large object-oriented software system. IEEE Trans. Softw. Eng. 33, 687–708.
Curtis, B., Sappidi, J., Szynkarski, A., 2012. Estimating the principal of an application's technical debt. IEEE Softw. 29, 34–42.
Davey, J., Burd, E., 2000. Evaluating the suitability of data clustering for software remodularisation. In: Proceedings of the Seventh Working Conference on Reverse Engineering, 2000, pp. 268–276.
Dazhou, K., Baowen, X., Jianjiang, L., Chu, W.C., 2004. A complexity measure for ontology based on UML. In: Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004, FTDCS 2004, pp. 222–228.
El Emam, K., Benlarbi, S., Goel, N., Rai, S.N., 2001. The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans. Softw. Eng. 27, 630–650.
Ferreira, K.A.M., Bigonha, M.A.S., Bigonha, R.S., Mendes, L.F.O., Almeida, H.C., 2012. Identifying thresholds for object-oriented software metrics. J. Syst. Softw. 85, 244–257.
Genero, M., Piattini, M., Manso, E., Cantone, G., 2003. Building UML class diagram maintainability prediction models based on early metrics. In: Proceedings of the Ninth International Software Metrics Symposium, 2003, pp. 263–275.
Grand, M., 2003. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML. John Wiley & Sons.
Guoai, X., Yang, G., Fanfan, L., Aiguo, C., Miao, Z., 2008. Statistical analysis of software coupling measurement based on complex networks. In: International Seminar on Future Information Technology and Management Engineering, 2008. FITME '08., pp. 577–581.
Gyimothy, T., Ferenc, R., Siket, I., 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. 31, 897–910.
Hamilton, J., Danicic, S., 2012. Dependence communities in source code. In: 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 579–582.
Heitlager, I., Kuipers, T., Visser, J., 2007. A practical model for measuring maintainability. In: Sixth International Conference on the Quality of Information and Communications Technology, 2007. QUATIC 2007, pp. 30–39.
Henderson-Sellers, B., Constantine, L.L., Graham, I.M., 1996. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). Object Oriented Syst. 3, 143–158.
Hitz, M., Montazeri, B., 1995. Measuring coupling and cohesion in object-oriented systems. In: Proceedings of the International Symposium on Applied Corporate Computing, pp. 75–76.
Hneif, M., Lee, S.P., 2011. Using guidelines to improve quality in software nonfunctional attributes. IEEE Softw. 28, 72–77.
Hosking, J.R.M., Wallis, J.R., 1987. Parameter and quantile estimation for the generalized Pareto distribution. Technometrics 29, 339–349.
Hu, H., Fang, J., Lu, Z., Zhao, F., Qin, Z., 2012. Rank-directed layout of UML class diagrams. In: Proceedings of the First International Workshop on Software Mining. ACM, Beijing, China, pp. 25–31.
Hyland-Wood, D., Carrington, D., Kaplan, S., 2006. Scale-free nature of java software package, class and method collaboration graphs. In: Proceedings of the Fifth International Symposium on Empirical Software Engineering. Rio de Janeiro, Brasil. Rio de Janeiro, Brasil.
Hyndman, R.J., Fan, Y., 1996. Sample quantiles in statistical packages. Am. Stat. 50, 361–365.
Ichii, M., Matsushita, M., Inoue, K., 2008. An exploration of power-law in use-relation of Java software systems. In: 19th Australian Conference on Software Engineering, 2008. ASWEC 2008, pp. 422–431.
Inman, H.F., 1994. Karl Pearson and R.A. Fisher on statistical tests: a 1935 exchange from nature. Am. Stat. 48, 2–11.
Izurieta, C., Griffith, I., Reimanis, D., Luhr, R., 2013. On the uncertainty of technical debt measurements. In: 2013 International Conference on Information Science and Applications (ICISA), pp. 1–4.
Jenkins, S., Kirk, S.R., 2007. Software architecture graphs as complex networks: a novel partitioning scheme to measure stability and evolution. Inf. Sci. 177, 2587–2601.
Karsai, G., Maroti, M., Ledeczi, A., Gray, J., Sztipanovits, J., 2004. Composition and cloning in modeling and meta-modeling. IEEE Trans. Control Syst. Technol. 12, 263–278.
Kollmann, R., Selonen, P., Stroulia, E., Systa, T., Zundorf, A., 2002. A study on the current state of the art in tool-supported UML-based static reverse engineering. In: Proceedings of the Ninth Working Conference on Reverse Engineering, 2002, pp. 22–32.
LaBelle, N., Wallingford, E., 2004. Inter-package dependency networks in open-source software. arXiv preprint cs/0411096.
Letouzey, J., Ilkiewicz, M., 2012. Managing technical debt with the SQALE method. IEEE Softw. 29, 44–51.
Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. J. Syst. Softw. 23, 111–122.
Lim, E., Taksande, N., Seaman, C., 2012. A balancing act: what software practitioners have to say about technical debt. IEEE Softw. 29, 22–27.
Loo, K.N., Lee, S.P., Chiew, T.K., 2012. UML extension for defining the interaction variants of design patterns. IEEE Softw. 29, 64–72.
Louridas, P., Spinellis, D., Vlachos, V., 2008. Power laws in software. ACM Trans. Softw. Eng. Methodol. 18, 1–26.
Ma, Y.-T., He, K.-Q., Li, B., Liu, J., Zhou, X.-Y., 2010. A hybrid set of complexity metrics for large-scale object-oriented software systems. J. Comput. Sci. Technol. 25, 1184–1201.
Martin, R., 1994. OO Design Quality Metrics, an Analysis of Dependencies.
MathWave, EasyFit. 2014. http://www.mathwave.com/easyfit-distribution-fitting.html.
McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. SE-2, 308–320.
Mcsweeney, P.J., 2008. Random Network Plugin. https://sites.google.com/site/randomnetworkplugin/Home.
Mei-Huei, T., Ming-Hung, K., Mei-Hwa, C., 1999. An empirical study on object-oriented metrics. In: Proceedings of the Sixth International Software Metrics Symposium, 1999, pp. 242–249.
Meyer, B., 2000. Object-oriented Software Construction. Prentice Hall PTR.
Milanova, A., 2005. Precise identification of composition relationships for UML class diagrams. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM, Long Beach, CA, USA, pp. 76–85.
Milanova, A., 2007. Composition inference for UML class diagrams. Autom. Softw. Eng. 14, 179–213.
Myers, C.R. , 2003. Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. Phys. Rev. E 68, 046116.
Newman, M., 2003. The structure and function of complex networks. SIAM Rev. 45, 167–256.
Newman, M.E.J., 2006. Modularity and community structure in networks. Proc. Natl. Acad. Sci. 103, 8577–8582.
Olague, H.M., Etzkorn, L.H., Gholston, S., Quattlebaum, S., 2007. 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. 33, 402–419.
Oracle, 2015. Java Platform SE 7 Documentation. Available online: http://docs.oracle.com/javase/7/docs/api/.
Ovatman, T., Weigert, T., Buzluca, F., 2011. Exploring implicit parallelism in class diagrams. J. Syst. Softw. 84, 821–834.

Oyetoyan, T.D., Falleri, J.R., Dietrich, J., Jezek, K., 2015. Circular dependencies and change-proneness: an empirical study. In: IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER), 2015, pp. 241–250.
Pang, T.Y., Maslov, S., 2013. Universal distribution of component frequencies in biological and technological systems. Proc. Natl. Acad. Sci. 110 (15), 6235–6239.
Parizi, R.M., Lee, S.P., Dabbagh, M., 2014. Achievements and challenges in state-of-the-art software traceability between test and code artifacts. IEEE Trans. Reliab. 63 (4), 913–926.
Porres, I., Alanen, M., 2003. A generic deep copy algorithm for MOF-based models. Model Driven Architecture: Foundations and Applications, p. 49.
Potanin, A., Noble, J., Frean, M., Biddle, R., 2005. Scale-free geometry in OO programs. ACM Commun. 48, 99–103.
Ragab, S.R., Ammar, H.H., 2010. Object oriented design metrics and tools a survey. In: The Seventh International Conference on Informatics and Systems (INFOS), 2010, pp. 1–7.
Rozenberg, G., Ehrig, H., 1999. Handbook of Graph Grammars and Computing by Graph Transformation. World scientific, Singapore.
Satuluri, V., Parthasarathy, S., 2011. Symmetrizations for clustering directed graphs. In: Proceedings of the 14th International Conference on Extending Database Technology. ACM, pp. 343–354.
Shannon, P., Markiel, A., Ozier, O., Baliga, N.S., Wang, J.T., Ramage, D., Amin, N., Schwikowski, B., Ideker, T., 2003. Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome Res. 13, 2498–2504.
Shiwen, S., Chengyi, X., Zhenhai, C., Junqing, S., Li, W., 2009. On structural properties of large-scale software systems: from the perspective of complex networks. In: Sixth International Conference on Fuzzy Systems and Knowledge Discovery, 2009FSKD '09, pp. 309–313.
Simon, H., 1991. The architecture of complexity. Facets of Systems Science. Springer, US A, pp. 457–476.
Smirnov, N., 1948. Table for Estimating the Goodness of Fit of Empirical Distributions, pp. 279–281.
SonarQube, SonarQube 2014. http://www.sonarqube.org/.
Stein, C.M., 1981. Estimation of the mean of a multivariate normal distribution. Ann. Stat. 9 (6), 1135–1151.
Sterling, C., 2010. Managing Software Debt: Building for Inevitable Change. Addison-Wesley Professional.
Stevens, W., Myers, G., Constantine, L., 1979. Structured design. In: Edward Nash, Y. (Ed.), Classics in Software Engineering. Yourdon Press, pp. 205–232.
Subramanyam, R., Krishnan, M.S., 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Trans. Softw. Eng. 29, 297–310.
Tanaka, J.S., 1987. "How big is big enough?": sample size and goodness of fit in structural equation models with latent variables. Child Dev. 58, 134–146.
Taube-Schock, C., Walker, R., Witten, I., 2011. Can we avoid high coupling? In: Mezini, M. (Ed.), ECOOP 2011—Object-oriented Programming. Springer, Berlin / Heidelberg, pp. 204–228.
Tempero, E., Anslow, C., Dietrich, J., Han, T., Jing, L., Lumpe, M., Melton, H., Noble, J., 2010. The Qualitas Corpus: a curated collection of Java code for empirical studies. In: 17th Asia Pacific Software Engineering Conference (APSEC), 2010, pp. 336–345.
Turnu, I., Concas, G., Marchesi, M., Tonelli, R., 2013. The fractal dimension of software networks as a global quality metric. Inf. Sci. 245, 290–303.
Valverde, S., Cancho, R.F., Sole, R.V., 2002. Scale-free networks from optimal design. EPL (Europhys. Lett.) 60, 512.
Valverde, S., Solé, R.V., 2003. Hierarchical small worlds in software architecture. arXiv preprint cond-mat/0307278.
Wenhui, L., Kuanjiu, Z., Jinjin, F., Zongzheng, C., 2010. Research on software cascading failures. In: International Conference on Multimedia Information Networking and Security (MINES), 2010, pp. 310–314.
Williamson, D.F., Parker, R.A., Kendrick, J.S., 1989. The box plot: a simple visual method to interpret data. Ann. Intern. Med. 110, 916–921.
Yang, G., Guoai, X., Yixian, Y., Xinxin, N., Shize, G., 2010. Empirical analysis of software coupling networks in object-oriented software systems. In: IEEE International Conference on Software Engineering and Service Sciences (ICSESS), 2010, pp. 178–181.
Yang, G., Jia, L., Shuai, S., Guoai, X., Gong, C., 2013. Weighted networks of object-oriented software systems: the distribution of vertex strength and correlation. In: Yang, G. (Ed.), Proceedings of the 2012 International Conference on Communication, Electronics and Automation Engineering. Springer, Berlin / Heidelberg, pp. 1185–1190.
Yann-Gaël, G., 2004. A reverse engineering tool for precise class diagrams. In: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative research. IBM Press, Markham, Ontario, Canada, pp. 28–41.
Yoon, J., Blumer, A., Lee, K., 2006. An algorithm for modularity analysis of directed and weighted biological networks based on edge-betweenness centrality. Bioinformatics 22, 3106–3108.
Zamani, S., Lee, S.P., Shokripour, R., Anvik, J., 2014. A noun-based approach to feature location using time-aware term-weighting. Inf. Softw. Technol. 56, 991–1011.
Zimmermann, T., Nagappan, N., 2008. Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th International Conference on Software Engineering. ACM, pp. 531–540.



Chun Yong Chong obtained his M.Sc. degree in Computer Science from University of Malaya, Malaysia, in 2012. He is currently a Ph.D. candidate in the Software Engineering Department at University of Malaya. His current research interests include reverse engineering, software clustering, software remodularization, and graph theory.

Sai Peck Lee is a professor at Faculty of Computer Science and Information Technology, University of Malaya. She obtained her Master of Computer Science from University of Malaya, her Diplôme d'Études Approfondies (D.E.A.) in Computer Science from Université Pierre et Marie Curie (Paris VI) and her Ph.D. degree in Computer Science from Université Panthéon-Sorbonne (Paris I). Her current research interests in Software Engineering include Object-Oriented Techniques and CASE tools, Software Reuse, Requirements Engineering, Application and Persistence Frameworks, Software Traceability and Clustering. She has published an academic book, a few book chapters as well as more than 100 papers in various local and international conferences and journals. She has been an active member in the reviewer committees and programme committees of several local and international conferences. She is currently in several Experts Referee Panels, both locally and internationally.