Software Systems as Complex Networks

Lian Wen
Griffith University
l.wen@griffith.edu.au

Diana Kirk
Griffith University
d.kirk@griffith.edu.au

R. G. Dromey
Griffith University
g.dromey@griffith.edu.au

Abstract

As software systems become larger and more complex, in order to understand, manage and evolve these systems, we need better ways of characterizing and controlling their macroscopic properties. We suggest complex network theory may be useful for these purposes. In recent years, researchers have shown that many complex systems from different disciplines can be investigated as complex networks and most of them comply with a scale-free network model. We explore the view that a software system can be studied as a network with a number of components (classes) connected by dependency (integration) relationships; we call this network the Component Dependency Network (CDN). The CDNs of several Java libraries and applications have been examined and all of them exhibit some scale-free characteristics. This result has some practical value including that it allows us to identify important components (classes) and thereby assists software maintenance and reengineering. We have built a tool to study software systems as complex networks. In the paper we also suggest ways of controlling and changing how systems evolve in order to improve their understandability and maintainability.

1. Introduction

As software intensive systems become larger and more complex there is a need to better understand the macroscopic properties of these systems if we are to make better-informed decisions about reengineering, maintaining and evolving such systems. Because of scale effects traditional ways of viewing and characterizing large systems do not appear to be adequate for our needs. There is therefore a need to explore techniques from other disciplines that have had to deal with systems of large-scale complexity. *Complex network theory* [2][19] is therefore an obvious candidate discipline for us to explore.

In recent years, researchers have investigated complex systems from different domains as *complex networks* [1][18]. Those networks include both those occurring in nature, for example, neuron activity, cellular metabolisms, protein folding and social networks, and those created by people, for example, electricity supply networks, the web and the internet. Even though they represent totally different types of systems, they share many common properties such as power-law degree distribution [1], small world property [19] and high level of clustering [2][3]. The similarity between different complex networks inspires the idea that network growth mechanisms represent some basic natural tendency to create order from chaos.

The recent progress in complex network theory motivates us to study software systems as complex networks. A software system can be treated as a network of components connected by dependency relationships [7]. The importance of relationship is stressed by the Object-Attribute-Relation (OAR) model [4], which shows that human memory and knowledge are represented by connections of synapses between neurons, rather than by the neurons themselves [5]. Similarly, for software systems, some high level functions are realized by the relationships of low level components. In this paper, the term component indicates an abstract form of software or even hardware entities as in the behavior tree approach [6]. A component can be a traditional component in CORBA or a Java bean, a class in an object-oriented system, a sub-system of a software system and even a hardware device. Similarly, the relationship between two components is also abstracted as a dependency relationship. We call this network a component dependency network (CDN). In this paper, we have studied the degree distributions of the CDNs of several Java libraries and applications and found that all of them show evidence of scale-free characteristics. That means that all the degree distributions follow a power-law distribution. A similar power-law distribution is also observed in applications written in C++ and Smalltalk [9].

This power-law distribution of CDNs provides some evidence to support our initial conjecture that the

Proc. 6th IEEE Int. Conf. on Cognitive Informatics (ICCI'07)
D. Zhang, Y. Wang, and W. Kinsner (Eds.)
1-4244-1327-3/07/$25.00 ©2007 IEEE

evolution of software systems is like that of other kinds of complex systems. Not only does this result inspire some philosophical consideration of software systems, but it also has practical value.

In this paper, based on CDNs, we propose a method, which applies a webmining technique [20], to identify important classes (components) for large software systems. This analysis is static. Compared to a dynamic approach [8] based on the runtime method call, the proposed approach is more suitable for investigating libraries or systems where an overall dynamic analysis is impossible.

The organization of the paper is as follow. In section 2, we briefly introduce complex networks and describe some properties they exhibit. In section 3, we present some evidence to support the claim that software systems adhere to complex network principles and show scale-free network properties. In section 4, we introduce the proposed method to identify important classes in a software system. Finally, some discussions are provided in section 5 and a brief summary is presented in the last section.

2. Complex networks

For many years, networks have been treated as either *random networks* or as highly structured *lattices*. The random network theory of Erdös and Rényi has dominated scientific thinking about networks since its introduction in 1959 [2]. In random networks, edges linking vertices are assumed to be randomly placed and the number of edges attached to each vertex (the *degree*) has been shown to follow a normal distribution i.e. tends to be narrowly distributed about some average. A random network is characterised by short average path length. For a lattice, each vertex has the same degree and the network exhibits long path length. For example, in the 1-dimensional lattice depicted in Figure 1, each vertex is associated with exactly 4 edges.

However, these two traditional models cannot be used to explain many real large complex networks such as human social networks. Therefore, two new network models have been proposed; the first one is small world networks [18] and the second is scale-free networks [3].

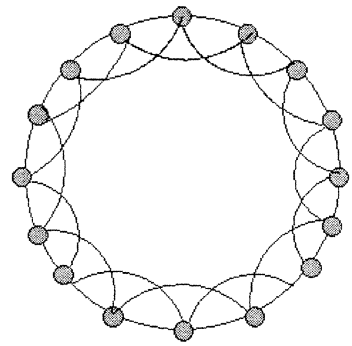

Figure 1. Lattice with degree 4

A small world network has a short *diameter*[1] and high *clustering coefficient*[2]. A simple method to generate a small world network is through a dynamic random 'rewiring' procedure on a lattice. For example, in Figure 2, three new connections have been added in the original lattice. The new network has a smaller diameter and keeps a high clustering coefficient. This kind of network was originally studied in human social networks [19].

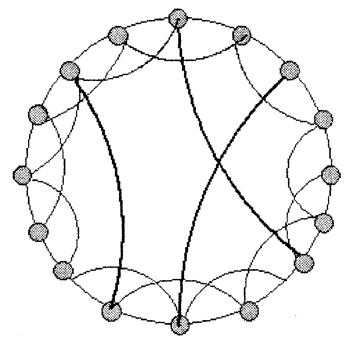

Figure 2. Lattice with rewiring

The scale-free network model was introduced in 1990s [2]. The most important feature of a scale-free network is the power-law degree distribution (see

[1] The diameter of a network is defined as the longest distance between two nodes in the network.

[2] The clustering coefficient is defined as: in a network, for each node i, suppose there are k_i other nodes connected to it and there are n_i links between those k_i nodes, then the coefficient for node i is $C_i = 2\, n_i / k_i\,(k_i - 1)$. The clustering coefficient of the whole network is the average of all nodes' clustering coefficient.

Figure 3[3]), on contra to the normal distribution of a random network. The direct effect of a power-law distribution is that there are a few nodes, called hubs, with a much larger number of connections compared with the average; it seems that there is no upper limit of the possible links a hub may have as the size of the network increased, and that is the reason why the property is called scale-free. A scale-free network usually exhibits following properties [2]:

1. **Power-law degree distribution**: the degree distribution has a long decreasing tail, which follows the power-law.
2. **Hubs**: A few nodes have a much larger number of links than most other nodes.
3. **Small world**: The average distance between nodes in a scale free network is very small compared to the network's size.
4. **Clustering**: The clustering coefficient of some scale-free networks is found to be much larger than that of a random network with the same number of nodes and links.
5. **Efficiency of Spread**: In a scale-free network, via the hubs, information can spread through a network much more efficiently than through a random network [16].
6. **High error tolerance**: A scale-free network can tolerate a much higher error rate than a random network, if the errors happen in randomly selected nodes or links.
7. **Vulnerable to well organized attacks**: Scale-free networks are highly tolerant to random errors, but may have weak points: the hubs.

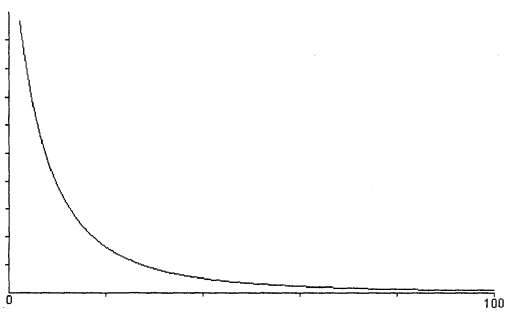

Figure 3. Power-law distribution

The significance of the scale-free network model is due to the fact that many real world complex networks are discovered to comply with this model. Those networks include cellular metabolisms [11], chemical reaction networks, the internet, the protein regulatory network [12], research collaboration networks, social networks and the World Wide Web [3]. The fact that so many different kinds of system show strikingly similar topological properties has spawned huge interest across many disciplines and has caused researchers to postulate the existence of some basic organisational principle. A breakthrough in the study of complex networks may result in good progress in the research of many different fields. In this paper, we suggest that the scale-free network property also exists in software systems. This implies that the research results relating to complex networks in other disciplines can also be useful in software engineering.

In the next section, we will present some test results and show that some software systems exhibit scale-free properties.

3. Evidence

A software system can be treated as a network of components connected with dependency relationships. Here, the concept of a component is a high level functional abstraction. For different types of software system, the implementation of components and the mechanisms for connecting components are different.

In Java and other object oriented systems, we treat a public class or a public interface as a component. In this section, we have tested the component dependency network (CDN) of several Java libraries and applications. All of them show scale-free properties. Combined with other work on C++ and Smalltalk [9], these results support our initial conjecture.

3.1. A Tool to retrieve the CDN of software systems written in Java

For a Java system, the classes are arranged in a hierarchy of packages. At source code level, a Java package is a directory in the file system, and each public class of the package has a corresponding Java file in that directory. Some packages may have sub-packages that are represented by sub-directories. In the component dependency network (CDN) of a software system written in Java, each public class (or a public interface) is a node. Suppose C_1 and C_2 are two classes of a Java system. If in the source code of C_1, C_2 is explicitly referred to, we say class C_1 is directly dependent on C_2. We represent this relationship by a directed link from node C_1 to node C_2 in the CDN. Some links are bi-directional, which means the two linked classes are dependent on each other.

We have developed a tool to explore the topological structure of a Java package's CDN (include sub-packages). The tool can be freely downloaded

[3] $y \propto x^{-\gamma}$, x is the number of connections, y is the number of nodes, and γ is the distribution exponent

from the web [24]. After selecting a target Java package, this tool can perform the following tasks:

1. Scan and parse all the Java source code under the given directory and the sub-directories to create the nodes and the dependency links.
2. Calculate common statistical properties of the CDN.
3. Visualize the CDN.
4. Draw histograms of the degree distribution and infer the parameters of the power-law distribution.
5. Display the rank of class.
6. Display the dependency tree for any selected Java class.

3.2. Test results

We have tested eight different Java applications and libraries. Five of them are from Sun's Java release j2sdk1.4.1_2, one is from an open source project apache ant [25], and the other two are from small Java projects written by one of the authors of this paper. The CDN of Java package java.awt and apache ant are shown in Figure 4 and Figure 5. The positions of nodes in the CDNs are calculated using a force-directed algorithm [26].

The statistical results of the tested Java packages are listed in Table 1. In this table, n is the number of nodes, l is the total number of links, l_d is the total number of bi-directional links, k is the average degree, std is the standard deviation of the degree distribution, p is the probability of having a link between two arbitrary nodes if on a random network, C is the clustering coefficient, d and $d_{\max}$ are the average distance and the maximum distance between nodes. Finally the power-law distribution exponent is shown as γ_{in}, γ_{out} and γ_{tot}, which represent the numbers of incoming links, outgoing links and total links respectively.

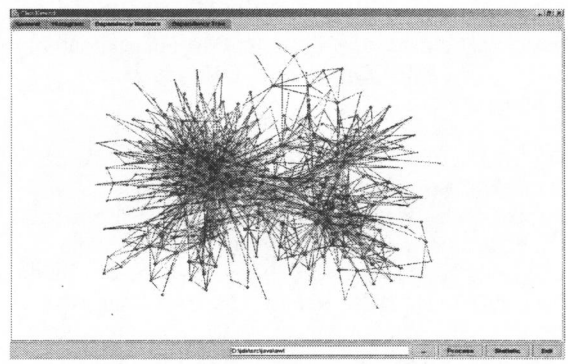

Figure 4. The CDN of Java package java.awt

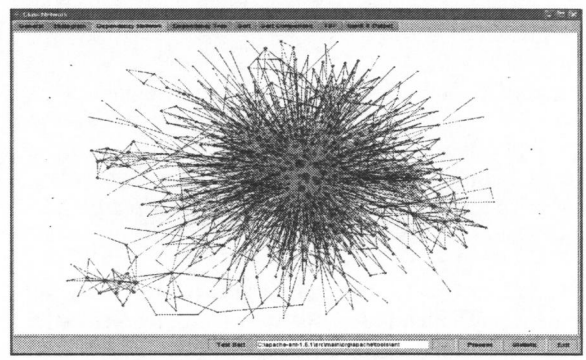

Figure 5. The CDN of apache ant

Table 1 Statistical results of the eight tested Java packages

#	Package	n	l/l_d	k	std	p	C	d	$d_{\max}$	γ_{in}	γ_{out}	γ_{tot}
1	java.awt	345	1721/151	4.99	12.72	0.03	0.63	2.99	7	1.58	1.11	1.42
2	java	1172	9453/374	8.7	31.45	0.01	0.57	2.58	6	1.23	1.46	1.43
3	javax	909	4683/124	5.15	15.89	0.01	0.61	4.03	13	1.09	1.21	1.74
4	com	642	2535/132	3.95	13.43	0.01	0.61	2.83	7	1.34	1.14	1.22
5	org	1083	7286/172	6.73	31.01	0.012	0.61	2.48	6	1.19	1.25	1.34
6	ant	628	2788/79	4.44	21.70	0.014	0.65	2.73	8	1.19	1.00	1.93
7	netp	79	126/8	1.59	3,71	0.051	0.69	3.01	8	1.09	1.08	1.87
8	classnet	36	61/6	1.70	3.59	0.10	0.73	2.90	6	1.47	1.70	4.30[4]

[4] For all the rest of the testing data, the value of the power law parameter γ is between 1 and 2, the reason for this extra large value is probably caused by the size of the test sample is too small. There are only 36 nodes in the CDN of classnet package.

From Table 1, it can be seen that the standard deviation of the degree distribution is about three times the size of the average degree. This is significantly different from a normally distributed random network model (in a normal distribution, the standard deviation is equal to the mean). The clustering coefficient is about 20 times larger than p, whereas in a random network, the clustering coefficient is approximately equal to p [1]. Therefore these CDNs are clearly not random networks.

For the eight Java CDNs tested, the degree distributions for incoming links, outgoing links and total links have been examined. All 24 distributions show characters of a power-law distribution. Some of the distributions are shown in Figure 6 - Figure 8 both in linear and logarithmic scales. The x-axis represents the number of input (output or total) links on a node and the y-axis represents the number of nodes. The curved line shows the function of the power-law distribution $y = A \times x^{-\gamma}$, where A and γ are constants for a given CDN.

Figure 6 and Figure 7 show strong evidence of power-law distribution. In Figure 8, even though the distribution is not smooth due to the small size of the sample, a long decreasing tail can be clearly observed. This is characteristic of a power-law distribution and indicates a trend towards the scale-free property.

From our testing result, we argue that the CDN for most Java systems, independent of the functionality of the systems, are scale-free networks.

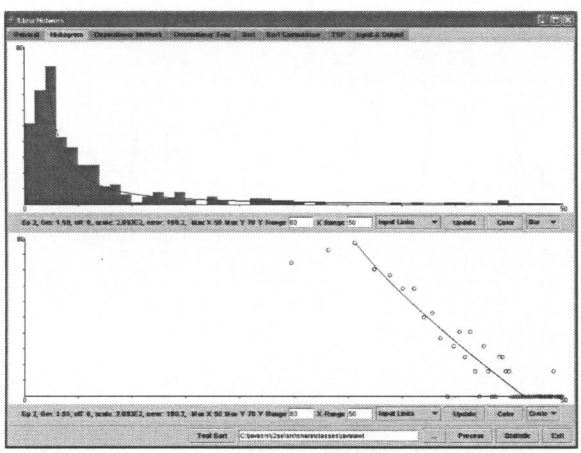

Figure 6 Degree distribution of incoming links of package java.awt.

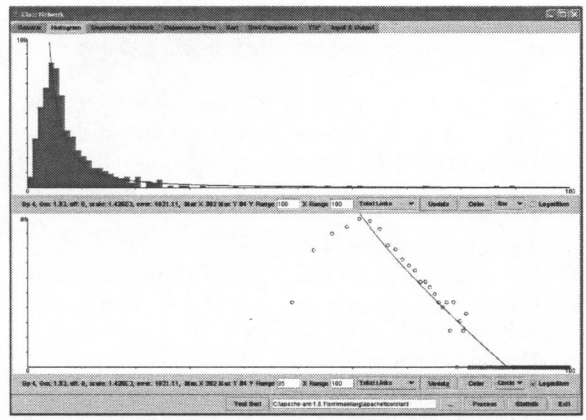

Figure 7. Degree distribution of the total links of apache ant.

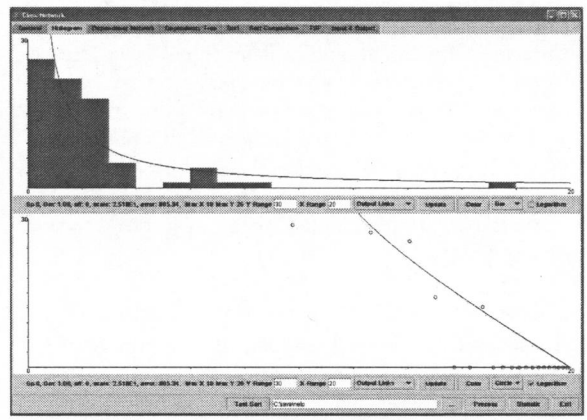

Figure 8. Degree distribution of outgoing links of Java package netp.

3.3. Other evidence

The power-law distribution has also been found in software systems written in C++ and Smalltalk. In Chidamber's paper [9], the authors introduce a metric called coupling between object classes (CBO) to measure OO systems. The concept is similar to the number of dependent components in this paper. In that paper, two histograms of CBO distribution, which come from a C++ system and a Smalltalk system, have been presented, and both of them show power-law distribution features. But that paper does not treat the dependency or coupling relationship as a network and does not claim the power-law distribution can be a general form for the dependence network for different types of software systems.

In this section, we have shown some evidence to support the conjecture that for most software systems,

the CDN is a scale-free network. From this conjecture, several interesting questions are raised.

1. What is the fundamental reason for a CDN to be scale-free?
2. Is a scale-free network the optimal topological structure for a CDN? If it is not, what is the optimal structure?
3. What are the practical benefits of knowing that the topological structure of a CDN is scale-free?

The first two questions, which are related to the origin and the advantages of scale-free networks, have already drawn many discussions in other disciplines [2][13][15]. Some of the discussions are applicable to software engineering. In Section 5 of this paper, we will also provide some discussions. However, a full answer of these two questions will be a fundamental contribution not only to software systems but also to complex systems in general and it is out of the scope of this paper.

To answer the third question, understanding the properties of a scale-free network may help to answer some interesting questions. For example, why do software systems work properly most of the time with known and unknown defects, but sometimes a very small defect may cause the whole system to crash? We believe that properties 6 and 7 of a scale-free network presented in section two may give some answer. *A scale-free network is high error tolerance for random errors but vulnerable to well organized attacks.* For more practical benefit, we propose that the scale-free property of a CDN may help people to identify important components and it may reduce software maintenance and re-engineering costs. The details are presented in the next section.

4. Practical usage

In the previous section, some evidence has been provided to support the conjecture that a CDN of a software system is a scale-free network. In this section, we propose that this knowledge may help people to identify some important components (classes) for legacy software systems.

Reverse engineering and software maintenance are two important activities in the field of software engineering. In order to facilitate and support these activities, researchers study the properties of code in an attempt to better understand the software systems represented by the code. Some aspects of the research include identification of key classes [8], classification of subsystems [14][17], and location of features [10].

A dynamic approach based on runtime method call has been proposed to identify important classes for OO systems [8]. The main idea of this approach is to execute some scenarios of the targeted system on a special platform so that all the runtime method calls can be recorded, and then analysed to identify some important classes.

There are three limitations of the dynamic approach. The first limitation is that this method is not suitable for analysing software libraries. The second limitation is that, for some large complex systems, it is not easy to cover all the major functions of the system through only a few scenarios. Finally, for some software systems, there may be no such special platform on which all the runtime method calls of a running system can be recorded. To address these limitations, a static approach that identifies important components (classes) through the analysis of a CDN, which can be retrieved from the source code, can be a good supplement.

A scale-free network has only a small numbers of highly-connected nodes; we propose that those nodes are more likely to be important components (classes).

A CDN is a directional network, so it can be useful to separate the incoming connections from the outgoing connections. Apart from the count of connection numbers, a webmining technique [20] has been applied to obtain a more sophisticated measure.

The idea of the webmining technique is not only to count the number of connections, but also to evaluate the quality of the connections. In a network, each node i has two associated values: the weight of hub (denoted by $h(i)$) and the weight of authority (denoted by $a(i)$) and they satisfy the following equations:

$h(i) = \sum a(j)$, if a link exists from node i to node j.

$a(i) = \sum h(j)$, if a link exists from node j to node i.

Therefore, for nodes with high weight of hub, not only they have many outgoing links, but also those links are linked to nodes with high weight of authority. Similarly, for nodes with high weight of authority, they should have many incoming links coming from nodes of high weight of hub.

Based on the number of incoming and outgoing links and the weight of hub and authority, the top 10 "important" classes from package java have been calculated and listed in Table 2.

Table 2. The top ten classes in package java based on different criteria

#	Incoming Links	Weight of authority	Outgoing Links	Weight of hub
1	String (628)	String (0.517)	Toolkit (121)	Component(0.114)
2	Object (401)	Object (0.409)	Component (105)	ObjectStreamClass(0.105)
3	IOException (261)	IOException (0.274)	ObjectStreamClass(60)	Toolkit(0.096)
4	IllegalArgumentException(241)	IllegalArgumentException(0.263)	Window(60)	Window(0.095)
5	System (163)	System (0.196)	Container(60)	ObjectInputStream(0.094)
6	Exception (131)	ClassNotFoundException(0.174)	ObjectInputStream(52)	Container(0.091)
7	Serializable (124)	Serializable (0.149)	KeyboardFocusManager(45)	Font(0.090)
8	NullPointerException (119)	Class (0.146)	Font(44)	ClassLoader(0.088)
9	ClassNotFoundexception(115)	Integer (0.144)	MetaData(43)	Security(0.087)
10	Class (113)	ObjectInputStream (0.132)	ClassLoader(42)	Beans(0.083)

From Table 2, it can be seen that the list of classes with highest numbers of income links is similar to the list of classes with highest weight of authority. A similar relationship can be seen between the classes with highest numbers of outgoing links and those with highest weight of hub. Generally, there are two different types of important classes. The first type is of simple but frequently reused class, and for this type of important classes, they usually have high number of incoming links and high weight of authority. Examples are "String" and "Object". The second type is of control or high level classes; these kinds of classes are very complex, provide many functions and require the support of many other classes. They usually have a high number of outgoing links and high weight of hub. Examples are "Component" and "Window".

Table 3 Classes in package java with the weight of authority ranking from 501 to 510

#	Weight of authority
501	FlatteningPathIterator(0.003)
502	ScrollPane(0.003)
503	ServerSocketChannel(0.003)
504	ICC_ColorSpace(0.003)
505	BeanContextServicesListener(0.003)
506	GlyphVector(0.003)
507	NameGenerator(0.003)
508	UnresolvedPermissionCollection(0.003)
509	AppletStub(0.003)
510	PaintContext(0.003)

Because there is no universal standard for the importance of classes in package java, it may be difficult to claim that all the classes listed in Table 2 are important. However, those with knowledge about the Java language will agree that at least half of the classes in the top 10 weight of authority are common used classes such as "String", "Object", "IOException", "System" and "Integer". Comparing the list of the top 10 weight of authority with that of classes with the weight of authority ranking from 501 to 510 (Table 3), which is in the middle of the whole list as there are more than 1000 public classes and interfaces in package java, it is clear that the classes in the top 10 list are much more frequently used and therefore more important.

In this section, a static approach has been proposed to identify important components (classes) in a software system. This approach is based on the assumption that for a software system, the CDN is usually a scale-free network and the degree distribution follows a power-law distribution. This feature implies that a small number of components have a higher number of connections and therefore having a high weight of authority or high weight of hub. This approach is also based on the assumption that important components do usually have high weight of authority or high weight of hub. The first assumption has been supported by testing results shown in the previous section, and the second assumption has been validated by a test on package java in this section.

The static approach for identifying important components in software systems is only one example to demonstrate the practical usage of the complex network theory in software engineering. We believe that there could be far more fundamental contributions to software engineering if we dig deeper into the complex network theory.

5. Discussion

5.1. The origin of scale-free networks

Why are many complex networks scale-free, what is the underlying mathematical model and how can we explain it?

Barabási's [2] explains this phenomenon by using a dynamic preference model. A brief explanation of this model is as follows. A complex network is evolved

from a primary network with a single node or a few nodes. When a new node is added into the network, it must create at least one link to connect the new node to the existing nodes. If the probability of connecting the new node to a particular node is the same among the existing nodes, then gradually the early nodes will have more links than the later nodes. However, this model cannot explain why a few nodes end up with many more links than the others, so a modified version has been introduced. In the new version, the probability of creating a link between the new node and an existing node depends on the number of links on the existing node. The more links an existing node has, the higher the probability that it will connect to the new node.

In software systems, the structure of the relationships between software components is interesting because this structure may affect either the behaviour of an individual component or of the system as a whole [19]. The dynamic preference model explanation has some relevance for software systems. When a system gradually increases in size, it is possible that programmers tend to use the most familiar components and this tendency results in these components becoming hubs or authorities.

5.2. Progressive activities and anti-regressive activities

It is well known that there are two different types of activities in a software development cycle: the progressive activities and the anti-regressive activities [21][22]. Progressive activities directly contribute to the implementation of software's functionalities but also increase the software's complexity or entropy. When a certain level of complexity is reached, the system may be difficult to maintain.

Anti-regressive activities are defined as those activities that do not directly increase the functions of a software product but improve its manageability so the software itself has the potential to grow in the future. These kinds of activities include updating of system documentation, rewriting of modules and complexity control.

From the architectural point of view, the progressive activities usually increase the complexity of a dependency network by adding new nodes and new links in the CDN. However, anti-regressive activities such as re-constructing the architecture or re-writing some modules may result in removing some links and nodes in the CDN and eventually reducing the complexity of the system.

The subtle relationship between the complexity of a CDN and the progressive and anti-regressive activities implies the possibility of using the CDN's complexity as an indicator or guideline for the software's manageability or the efficiency of the anti-regressive activities.

5.3. Is a scale-free network the best form of a CDN?

Even though the CDNs of all the tested software systems tend to be scale-free, is a scale-free network the best form for a CDN?

As we know, anti-regressive activities can reduce the complexity of a CDN. Based on our previous study, the topological structure of software's CDN can be independent of its functional requirements and can be normalized into a tree [7]. Therefore, theoretically, the anti-regressive activities can, under certain conditions, make the system's dependency network into a simple form such as a tree. Even though we may not be able to find an example of a tree-formed CDN in existing large systems, in principle it is possible to reach a tree structure using practically any anti-regressive activity. Another way to achieve a tree structure is to build a system from scratch. We can start with one component; whenever a new component is introduced, we connect it to only one of the existing components. If new connections have to be added in the existing CDN, we have to remove the same number of existing connections. (It is possible to use a tool to identify the most efficient connections and remove extra connections). If this procedure is strictly followed, the structure of the CDN will be a tree.

Besides a tree-formed CDN, a layered CDN is another simple form of architecture style. The layered architecture is not a new concept [23]. However, in our design, the structure is slightly different; we limit that a component depend on only components of one level lower. There will be no direct dependency relationships within one level or cross two levels. Figure 9 is an example of a layered CDN.

To build a system of a layered CDN, we can initially create all the base level components, and then based on those base level components, we create the second level components and then the third level components etc, and finally the system component. Of course, we can also use the top-down approach or even mix the top-down and bottom-up approaches together. The main idea is during the whole process, the layered architecture style has to be kept. A tool may be required to monitor the evolution of the CDN and

make sure the architecture style can be maintained during the system's lifecycle.

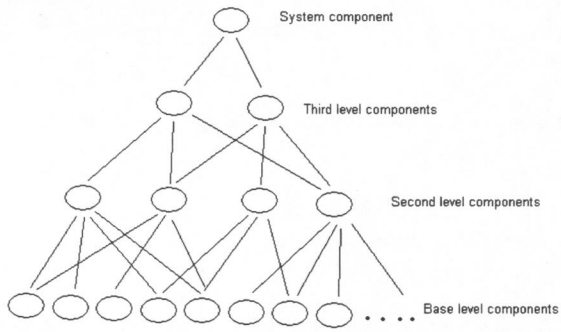

Figure 9. A software system of a layered CDN

One advantage of a layered CDN is that a layered CDN will result fewer levels in the component dependency trees. For a system without the layer style restriction, the component dependency tree can be very deep such as the tree in Figure 10. "Button" is only a primary component in package "java.awt", but the depth of the component dependency tree is about 40. However, for a system with layered CDN, the maximum level of the component dependency trees is the number of layers of the system that can be smaller than 10 or even 5. Therefore, if one component has been changed, the ripple effects will be limited within a much smaller range. This property will reduce the cost of the software maintenance.

From the discussion above, we know that theoretically, we can build software systems with a simpler structure than that implied by a scale-free network. If complexity is the major concern, we may have a better structure of a CDN.

5.4. Different ways to define the connections

In this paper, we define a connection between two components (classes in Java) as explicitly referring the other class's name in the source code of one class. This kind of definition is static, syntactic and structural. It is possible to extend the definition of a connection to different measures, for example from static to dynamic, from syntactic to semantic, and from structural to functional. This is an area for future research.

6. Summary

In this paper, the topological structures of CDNs of eight Java packages have been studied. The interesting discovery is that all of them clearly show scale-free properties. Based on this result and independent results from other research, we conjecture that the component dependency networks for most software systems may be scale-free. Based on this conjecture, we have proposed a static method to identify important components from a system's component dependency network. Also, we suggest ways of controlling and changing how systems evolve in order to improve their understandability and maintainability.

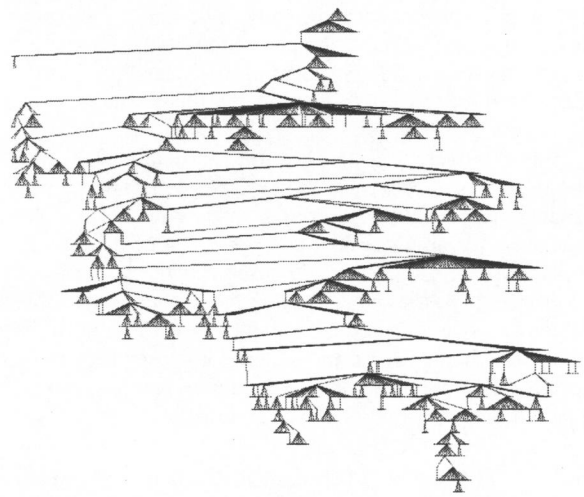

Figure 10. The component dependency tree of class "Button" in package "java.awt"

Parallel studies between different disciplines frequently inspire new ideas. The similarity between the topological structure of a CDN and other types of complex networks implies that the laws, which work behind the evolution of software systems, could be the same as those working behind the evolution of other complex systems such as human society and biological systems. Continuous studies may reveal more commonalities among the structure and evolution of those different complex systems and therefore provide new approaches to study software systems.

7. Acknowledge

The authors would like to acknowledge the Australian Research Council (ARC) Centre for Complex Systems for its support to this work.

8. References

[1] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks", *Reviews of Modern Physics,* vol. 74, pp. 47-97, 2002.

[2] A.-L. Barabási, "Linked", Perseus Publishing, 2002.

[3] A.-L. Barabási and E. Bonabeau, "Scale-Free Networks", *Scientific American*, May 2003.

[4] Wang, Y, "The OAR model for knowledge representation", the 19th IEEE Canadian Conference on Electrical and Computer Engineering, pp 1696-1699, 2006

[5] Wang, Y., "The Theoretical Framework of Cognitive Informatics", *The International Journal of Cognitive Informatics and Natural Intelligence*, 1(1) pp. 1-27, Jan, 2007

[6] Geoff R. Dromey, "From Requirements to Design: Formalising the Key Steps", (Invited Keynote Address), *IEEE International Conference on Software Engineering and Formal Methods*, SEFM'2003, pp. 2-11, Brisbane, September, 2003.

[7] Lian Wen, Geoff R. Dromey, "Architecture Normalization for Component-based Systems", *Electronic Notes in Theoretical Computer Science*, vol.160, pp. 335-348, 2006.

[8] Andy Zaidman, Toon Calders, Serge Demeyer, Jan Paredaens, "Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process", *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pp. 134-142, 2005.

[9] Shyam R. Chidamber, Chris F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, 20(6): 476-493, 6, 1994.

[10] Thomas Eisenbarth, Rainer Koschke and Daniel Simon, "Locating Features in Source Code", *IEEE Transactions on Software Engineering* (29/3), IEEE Computer Society, March 2003.

[11] H. Jeong, B. Tombor, R. Albert, Z.N. Oltvai and A.-L. Barabasi, "The large-scale organization of metabolic networks", *Nature*, 407, pp. 651, 2000.

[12] H. Jeong, S.P. Mason and A.-L. Barabasi, "Lethality and centrality in protein networks", *Nature*, 411, pp. 41, 2001.

[13] Beom Jun Kim, Ala Trusina, Petter Minnhagen and Kim Sneppen, "Self Organized Scale-Free Networks from Merging and Regeneration", *The European Physical Journal B*, http://arxiv.org/abs/nlin/0403006, 2005.

[14] Arun Lakhotia, "A Unified Framework for Expressing Software Subsystem Classification Techniques", *Journal of Systems and Software* (36), 1997.

[15] Kwangho Park and Ying-Cheng Lai, "Self-organized scale-free networks", *Physical Review* (E 72), 026131, 2005.

[16] Romualdo Pastor-Satorras and Alessandro Vespignani, "Epidemic Spreading in Scale-Free Networks", *Physical Review Letters*, 86(14), pp. 3200-3203, 2001.

[17] Vassilios Tzerpos and R.C. Holt, "ACDC: An Algorithm for Comprehension-Driven Clustering", *Proceedings of the Seventh Working Conference on Reverse Engineering*, IEEE Computer Society, 2000.

[18] D.J. Watts and S.H. Strogatz, "Collective dynamics of small-world networks", *Nature*, 393, pp. 400-442, 1998.

[19] Duncan J. Watts, "Six Degrees: The Science of a Connected Age", William Heinemann, 2003.

[20] Jon M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment", *Journal of the ACM*, 46(5) 604-632, 1999.

[21] Lehman, M.M., "*FEAST/2 Final Report – Grant Number GR/M44101*", 2001

[22] Lehman, M.M., "*Programs, Cities, Students, Limits to Growth?*", Inaugural Lecture, 1974

[23] Stafford, J. A., Wolf, A. L., "*Software Architecture*", Component-Based Software Engineering, putting the pieces together, Chapter 20, 2001

[24] Classnet, the tool to explore the CDN of a Java system, http://www.sqi.gu.edu.au/gse/tools/classnet.html, 2007

[25] Apache Ant, http://ant.apache.org/, 2007

[26] Cruz, I. F., Tamassia, R., 1998, "*Graph Drawing Tutorial*", http://www.cs.brown.edu/people/rt/papers/gd-tutorial/gd-constraints.pdf