Software Evolution Characterization

Kecia A. M. Ferreira
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, MG, Brazil
kecia@dcc.ufmg.br

Mariza A. S. Bigonha
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, MG, Brazil
mariza@dcc.ufmg.br

Roberto S. Bigonha
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, MG, Brazil
bigonha@dcc.ufmg.br

Bárbara M. Gomes
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, MG, Brazil
barbarag@dcc.ufmg.br

ABSTRACT

The study of software evolution has been the subject of much researches in the last decades, whose results reveal that a software system has continuing growth, continuing changes, increasing complexity and declining quality. However, the knowledge about how this process occurs is not consolidate yet. The present work provides a better understanding of software evolution process by analising it through the view of networks. A software system can be modeled as a graph whose nodes represent the software modules and whose edges represent the relationships between them. The study was performed on a set of 16 open source software sytems and a commercial application developed in Java. The results of this study reveal important facts about the evolutive nature of this type of software, such as: density of software systems tends to decrease, the diameter of software networks is short, classes with large in-degree tend to keep this property and their quality tends to degrade. Our study also identifies the macroscopic structure of software system what can support software engineers in the task of management and maintenance of software systems.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

ICSE 2011 Waikiki, Honolulu Hawaii

Keywords

software evolution, object-oriented software, complex networks

1. INTRODUCTION

The phenomenon of rotting design is a classical problem in Software Engineering. Despite all the knowledge about high-quality software construction consolidated in well-known principles, criteria, rules, design-patterns and techniques [20, 8, 25], it is known that as a software system evolves and changes, its architecture becomes more complex and rigid and, due to this design degradation, the program becomes increasingly hard to maintain. The Lehman Laws [16] describe this evolutive nature of software by postulating that a software system grows and suffer maintenances continuingly, has increasing complexity and decreasing quality throughout its evolution.

In the last years, there have been a lot of investments by researchers in the characterization of software evolution. Most of this works are concerned in investigating whether the Lehman's laws are applied in open source software, especially in the aspects of growth and complexity. Growth has been usually evaluated by means of metrics such as LOC or number of files, while complexity has been evaluated by means of McCabe or Halstead complexity metric. A few researches have used other software metrics to study software evolution, for instance: number of deleted/added/changed files, coupling and cohesion metrics. Most recently, the concepts of complex networks has been timidly applied to understand the behavior and the nature of software structure. A common finding of such works is that the in-degree of nodes in the network of modules within a software system follows a power-law and this network seems to conform to the so-called small-world phenomenon. An example of the potential application of the complex networks analysis is a study of Zimmerman and Nagappan that identified correlation between the number of defects in Windows Server 2003 and measures from network analysis, such as centrality and closeness.

These works bring important revelations towards a better

understanding of the real structure of the software systems. However, there is still a great lack of substantial knowledge about the evolution of design of software systems. Among the many open-questions in this topic are the following ones: the evolution of modules that have high in-degree in terms of in-degree, internal quality and stability; the behavior of network measures, such as diameter, density and the size of the largest strongly connected component; the existence of a general macroscopic structure of the network of modules within software systems and its evolution.

In the present paper, we describe the results of an empirical study that aims to investigate those issues. The data set analyzed in this work is from 16 open source object-oriented software systems, in a total of 109 versions of the programs, and from one commercial object-oriented software system. Our analysis yields valuable insight into the evolution of software system structure: the classes with higher in-degree tend to keep this position with the growth of the software system, tend to gain more methods, and have declining internal cohesion; the network of modules within a software system has short diameter, has shrinking density, and its largest strongly connected component enlarges with the growth of the software system. Even more interesting, our analysis reveals an intriguing picture of the macroscopic structure of software systems. The findings of this study are a step ahead towards a better understanding of the laws that control the evolution of software systems structures. Moreover, they can be used in order to directing maintenance tasks and test plans.

This paper is organized as follow: Section 2 provides a background of concepts used in our study; Section 3 is a review of related works; Section 4 describes the methodology and the data used in the study; Section 5 reports the experiments and its results; Section 6 brings the conclusions and indication of future works.

2. BACKGROUND

An object-oriented software system can be modeled as a directed graph (a network) in which the classes are the vertices and a connection between two classes is an edge. We consider that a class A is connected to another class B if A uses a field or a method of B or if A extends B. In this situation, there is an edge from A to B. In the present study, software evolution is evaluated by means of software metrics and network metrics that are described in this section. We also give a background about the network analysis terminology and the concepts used in this work.

2.1 Software Metrics

The following software metrics are used in order to evaluate the evolution of classes of a software system:

- Number of public methods: it is the number of public methods defined in the class.
- Number of public fields: it is the number of public fields defined in the class.
- Class cohesion: there is several class cohesion metrics proposed in the literature, however there is no consensual way to measure cohesion yet. In this work we use the metric Cohesion by Concern (CoCo) [6]. This metric is given by 1/C, where C is the number of disjointed sets M of methods within the class. Each set consists of similar methods. Two methods are similar when they use a common field or a common method of its class. For instance, if there are 2 sets in a class, CoCo will result in 0,5. This indicates that the class has 2 concerns. If there is only one set in a class, CoCo will result in 1, which indicates high cohesion.

2.2 Network Metrics

The following network metrics are used in order to evaluate the evolution the software systems:

- Network density: in a network with a edges and n vertices, this metric is given by $a/(n(n-1))$ [17]. In the context of software metrics, this metric is called COF (coupling factor) [1].
- Diameter: the diameter of a network is the length, given in number of edges, of the longest geodesic path within the network. A geodesic path is the shortest path between two vertices. In a social network, for example, it is an indicator of how rapidly information would spread throughout the network [21].
- In-degree: the in-degree of a vertex is given by the number of vertices from which there is an edge that reach the vertex. In a software system network, it is the number of classes that use services of a class or extend it.

2.3 Complex Networks

The empirical observation of real networks yielded valuable comprehension of such networks. The work of Newman [21] presents a wide review about the advances in this field called Complex Networks. The study of the properties of networks includes such concepts as the small-world phenomenon, degree distributions, scale-free networks and models of network growth.

A power-law is a probability distribution function in which the probability of a random variable X takes a value x is proportional to a negative power of x, denoted by $P(X = x) \propto cx^{-k}$. Networks with power-law degree distribution are referred as scale-free networks. In a scale-free network there is a great number of vertices with low degree and a small portion of them with high degree. There has been a spate of interest in such networks in the literature, since power-law degree distribution has been observed in a wide range of networks like the Web, the Internet, metabolic networks, telephone calls graphs and software system networks [2, 18, 24, 26, 27, 5]. An important property of a scale-free network is its resilience to the removal of their vertices. TA study of vertex deletion in the Internet and Web concluded that such networks are resilient against random failure of vertex in the network, whereas the target removals at the highest degree vertices in the network are destructive [21]. Since software systems networks are also scale-free, this property can be applied to them: an error or maintenance in a class with high in degree could widely affect other classes in the system.

The small-world phenomenon refers to a characteristic of networks whose most pairs of vertices are connected by a short path. This characteristic is related to the easiness of information propagation in the network. Depending on the kind of the network, information should take different meaning, such as spread of disease in a population, a rumor in a

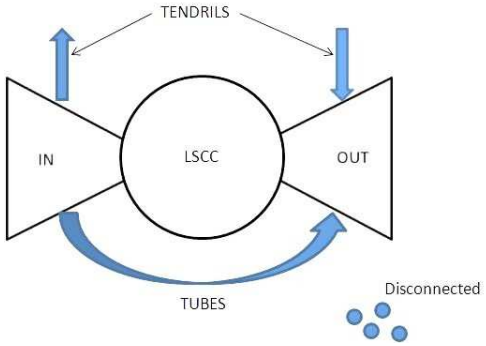


Figure 1: The bow-tie model of the Web

social network and error propagation in a software system network.

Models of networks help us understand network topology and the processes taking place on networks. In this work, we explore the structure of software systems networks by the light of concepts and characteristics of complex networks.

2.4 The Bow-tie Model

The study of Broder et al. [3] infers that the macroscopic structure of the Web can be modeled by a picture known as bow-tie. This picture, shown in Figure 1, suggests that web pages can be divided into five groups: SCC, in, out, tendrils, tubes and disconnected. Their analysis reveals that in the Web graph there is a central core in which all of pages can reach one another, which they called the giant strongly connected component (SCC). Another group of pages can reach the ones in SCC but cannot be reached from them. This group is called in. Out consists of pages that can be reached from SCC but cannot link it back. Tendril consists of pages that cannot reach the SCC and are not reachable from it; pages in tendrils can be reached from IN or can reach OUT, without passing through SCC. There is a group of pages in tendrils that can be reached from in, then be connected to another tendril, leading into OUT. This group of pages is called tubes.

In the present work we investigate how well the bow-tie model fits to the software system network. Our analysis reveals a simpler picture that can represent the way that classes in a object-oriented software system connect one to another.

3. RELATED WORKS

Software evolution has been studied extensively. One of the most noted works in this field resulted in the well-known Lehman's laws that include: continuing change, increasing complexity, continuing growth and declining quality [16]. Many investigators have recently studied whether the Lehman's laws can be applied to open-source software systems. A case study carried out by Godfrey et al. [9] with the Linux kernel concluded that the system grows continually in a geometric rate. Lee et al. [15] analyzed the evolution of JFreeChart, an open source library, based on size, coupling and cohesion metrics. The results of the work point out that the evolution of the target software follows some of the Lehman's laws. Mens et al. [19] studied the evolution of Eclipse by means of software metrics, such as number of added, changed and deleted files and number of errors. They found evidences of continuing growth and increasing complexity in Eclipse. It was also found that the added classes have higher fan-in and lower fan-out coupling comparing to the removed classes. Israeli and Feitelson [11] used software metrics in order to analyze the Linux kernel evolution. The results of their study support most of the Lehman's laws; however it was observed that the functions within the program have a decreasing average complexity. Xie et al [28] evaluated the evolution of 7 open source software systems. The results of their study demonstrate that the following Lehman's laws are applied to open-source software systems: continuing change, increasing complexity, self regulation and continuing growth. In addition they observed that most of modifications occur in a small portion of source code.

Software evolution has been commonly studied by means of software system growth. Koch [14] analyzed the growth of a large sample of open source software systems. The results of the study indicate that the mean growth rate is linear or tends to decrease over time, but a significant percentage of projects exhibit superlinear growth. Herraiz et al. [10] carried out a comparative study of two software metrics commonly used for characterizing the evolution of software: number of lines of code and number of files. They analyzed a package in Debian GNU/Linux and concluded that both metrics have the same behavior. The results of the study also confirm the findings of Godfrey et al.[9].

Other approaches have been used in the study of software evolution and software characterization. Many researchers have identified that in-degree distribution in software system network follow a power-law [13, 2, 18, 24, 26, 27, 5]. Olbrich et al. [22] investigated the evolution of code smells within a system and their impact on the change behavior of the infected system elements. They analyzed two open-source software systems and considered two code smells: *God Class*, which refers to those classes that take too many responsibilities and *Shotgun Surgery*, that occurs when a change in the code of the class causes changes in many other classes. They found that classes infected with these code smells have a high change frequency. Nevertheless they did not point out the way those classes suffer changes. Jenkins and Kirk [12] evaluated software evolution by using complex network theory. Their study was performed over some released versions of a component from the Sun Java2 Runtime Environment (rt.jar) and concluded that the degree distribution in the network of software class dependencies follows power law. They propose an instability metric that they claim to be conformed with the growth process of the software system. Zimmermann and Nagappan [29] investigated correlation between measures from network analysis, such as centrality and closeness, and the number of defects in Windows Server 2003. In their study, they built a dependency graph of the software system, and then they collected complexity measures and network measures. They concluded that the network measures used in the study can predict defects for binaries of Windows Server 2003 and that result can support managers in the task of allocating resources.

Despite the notable contribution of the works carried out in order to characterize software evolution, there are still open questions about this phenomenon. Most of the researches in this field concentrate in studying the growth of software systems in terms of lines of code and number of modules or files and some of them evaluate software evo-

Table 1: Software systems analyzed in the study

Name	*Category*	*# downloads/week*	*Time of life*	*#classes*	*#versions*	*#analyzed versions*
JEdit	Text editor	9.138	2001 a 2009	377 a 1124	13	13
Dr Java	Development	3.837	2002 a 2009	596 a 3692	10	10
Java Groups	Cooperation	465	2003 a 2009	696 a 1137	40	13
KoL Mafia	Game	1.007	2004 a 2009	39 a 1109	13	13
DBUnit	Database	448	2002 a 2009	198 a 369	25	5
FreeCol	Game	7.452	2003 a 2010	112 a 5902	27	5
JasperReports	Development	5.542	2001 a 2010	525 a 5304	50	5
JGNash	Financial	822	2002 a 2010	782 a 3603	40	5
Java msn library	Communication	271	2004 a 2010	494 a 872	10	5
Jsch	Security	2.304	2004 a 2009	202 a 271	29	5
JUnit	Development	1.834	2000 a 2009	78 a 230	18	5
Logisim	Education	1.590	2005 a 2009	908 a 1185	28	5
MeD's Movie Manager	Storage	1.169	2003 a 2010	64 a 517	60	5
Phex	Network	1.084	2001 a 2009	393 a 1352	26	5
Squirrel sql	Database	7.270	2006 a 2010	424 a 1223	26	5
Hibernate	Database	12.906	2004 a 2010	956 a 2446	53	5
Commercial - frontier layer	Commercial application	-	2005 - 2010	1100 a 1246	18	18
Commercial software - model layer	Commercial application	-	2005 - 2010	3343 a 4031	18	18

lution by means of software metrics. The present work is aimed at investigating software evolution from the viewpoint of complex networks. This approach is not new, since a few previous works analyzed software system as complex network. The results of the work described in this paper, however, lead to a better understanding of the behavior of the object-oriented software systems and leads to identify a novel generic macroscopic picture of model software systems.

4. METHODOLOGY

The selection of the open-source software systems analyzed in the study was based on the following criteria: time of life, quantity of versions or releases and category. The data was extracted from www.sourceforge.net, which classifies the programs in categories, such as development, games and communication. For each category, up to 10 software systems were selected, by satisfying the following criteria: they are developed in Java, they have 5 versions or releases at least and they have 4 years of life at least. Another criterion was the availability of bytecodes because the tool used to perform the measurements evaluates the compiled code, not the source code. The initial survey resulted in 108 programs. Among them, we selected by category those with biggest number of versions or releases, with biggest time of life and most popular. Popularity was evaluated by means of the number of downloads per week. This last selection resulted in 16 programs whose data are shown in Table 1. Data were gathered from sourcerforge.net from September 2009 up to April 2010.

In order to observe the existence of relevant difference between two consecutives versions of a program, we initially analyzed data from all the versions of three programs: JEdit, DrJava and Kolmafia. For JavaGroups, that has a large number of versions, we selected 13 of them: the first one, the last, and 11 intermediate versions, by observing a period of release approximately equal between two consecutives versions. We observed that the results of the subsequent versions are very close. Due to this, for the other software systems, it was selected 5 versions: the first one, the last, and three intermediate versions, by observing a period of release approximately equal between them.

The commercial software system analyzed in this work is developed by a Software Engineering laboratory of an important Brazilian university. The laboratory provides software and consulting solutions for different market segments and most of its clients are Brazilian government organizations. The software system selected for analysis is one of the oldest

Table 2: Evolution of the open source software systems

Software	*Version*	*#Classes*	*#Connections*	*COF*	*Diameter*
DBUnit	2.0	198	429	0,011	9
	2.2.1	289	666	0,008	11
	2.4.0	332	769	0,007	13
	2.4.4	347	780	0,006	17
	2.4.7	369	815	0,006	16
FreeCol	0.1.0	44	112	0,05900	5
	0.5.0	416	1899	0,011	12
	0.6.0	611	2609	0,007	11
	0.8.0	927	5150	0,006	13
	0.9.2	1087	5902	0,005	14
Jasper Reports	0.4.0	242	525	0,009	8
	1.0.0	574	1316	0,004	9
	2.0.0	1104	2435	0,002	13
	3.0.0	1233	3038	0,002	13
	3.7.1	1629	5304	0,002	13
JGNash	1.10.0	743	2757	0,005	16
	1.11.1	782	2443	0,004	17
	1.50.0	942	2659	0,003	12
	2.00.0	2716	7374	0,001	24
	2.20.0	3603	12978	0,001	24
Java msn library	10a1	171	494	0,017	10
	10a2	186	516	0,015	7
	10b1	203	615	0,015	7
	10b2	218	662	0,014	9
	10b3	270	872	0,012	9
JUnit	3.4	78	138	0,023	5
	3.8	101	182	0,018	6
	4.0	92	197	0,02	6
	4.5	188	352	0,01	8
	4.8.1	230	421	0,008	10
JavaGroups	2.2	696	1935	0,004	10
	2.2.1	849	2880	0,004	10
	2.2.5	829	2059	0,003	10
	2.2.6	832	2074	0,003	10
	2.2.7	857	2201	0,003	10
	2.2.8	810	2621	0,004	8
	2.2.9	922	2621	0,003	8
	2.3	959	2756	0,003	9
	2.4.1	1013	3075	0,003	7
	2.5.1	967	3736	0,004	8
	2.6.1	1012	3639	0,003	8
	2.7.0	1041	3688	0,003	11
	2.8.0	1137	3875	0,003	9
KolMafia	0.2	39	83	0,056	7
	0.4	75	222	0,04	9
	1.0	143	508	0,025	10
	2.0	191	726	0,02	11
	4.0	342	1399	0,012	12
	5.0	334	1780	0,016	11
	6.0	388	2102	0,014	10
	7.0	498	2970	0,012	12
	9.0	616	3410	0,009	11
	10.0	708	4004	0,008	13
	11.0	757	4578	0,008	14
	12.0	772	5357	0,009	12
	13.7	1109	7373	0,006	13

Software	*Version*	*#Classes*	*#Connections*	*COF*	*Diameter*
LogSim	2.0.0	908	3294	0,004	13
	2.1.0	993	3940	0,004	14
	2.1.5	1018	4141	0,004	14
	2.2.0	1054	4439	0,004	14
	2.3.3	1185	4609	0,003	14
MeD's Movie	1.6	64	149	0,037	6
Manager	1.7	73	168	0,032	6
	2.0	517	1067	0,004	10
	2.8	458	1465	0,007	12
	2.9.13	608	1845	0,005	13
Phex	0.6	393	1078	0,007	8
	2.0.0	897	3215	0,004	18
	2.8.0	1205	4352	0,003	16
	3.0.0	1419	6036	0,003	19
	3.4.2	1352	5480	0,003	20
Squirrel-sql	1.0	424	717	0,004	15
	2.0	729	1592	0,003	13
	2.6	940	1765	0,002	14
	3.0	1134	2570	0,002	16
	3.1	1223	2989	0,002	16
JSch	0.1.1.4	80	202	0,032	4
	0.1.20	83	204	0,028	5
	0.1.26	94	210	0,024	5
	0.1.34	109	271	0,023	5
	10.1.42	117	385	0,02	5
Hibernate	3.0	956	2739	0,003	19
	3.1	1118	3746	0,003	20
	3.2	1302	4102	0,002	23
	3.3.0	1690	5707	0,002	21
	3.5.1	2446	5980	0,001	21
JEdit	2.4	377	1192	0,009	8
	2.5	422	1474	0,008	8
	3.1	426	1595	0,009	8
	3.2	449	1672	0,008	8
	4.0	554	2059	0,007	9
	4.1	618	2393	0,006	12
	4.1-8	646	2550	0,006	12
	4.2	805	3255	0,005	10
	4.3	810	3276	0,005	10
	4.3.4	867	3444	0,005	10
	4.3.9	954	3671	0,004	10
	4.3.13	1008	3885	0,004	13
	4.3.18	1124	4261	0,003	12
DrJava	1011	596	1773	0,005	10
	2148	1064	3393	0,003	14
	1826	1108	3680	0,003	12
	2304	1512	4569	0,002	18
	2332	1622	5259	0,002	19
	1750	2036	8287	0,002	21
	1406	2187	9562	0,002	23
	1942	3003	9732	0,001	17
	r4592	3421	117000	0,001	14
	r4756	3692	13627	0,001	16

Table 3: Evolution of the commercial software system

Layer	*Version*	*#Classes*	*#Connections*	*COF*	*Diameter*
Frontier	V1	1100	2418	0,002	10
	V10	1162	2698	0,002	10
	V18	1246	1551	0,001	10

Layer	*Version*	*#Classes*	*#Connections*	*COF*	*Diameter*
Model	V1	3343	28420	0,003	14
	V10	3796	28812	0,002	14
	V18	4031	32490	0,002	14

Table 4: In-degree evolution - JEdit 2.4 and 4.3.18

Class	*in-degree*
org.gjt.sp.jedit.EditAction	127
org.gjt.sp.jedit.View	124
org.gjt.sp.jedit.textarea.JEditTextArea	120
org.gjt.sp.jedit.jEdit	101
org.gjt.sp.jedit.Buffer	66
org.gjt.sp.jedit.textarea.InputHandler	52
org.gjt.sp.util.Log	46
org.gjt.sp.jedit.GUIUtilities	33
org.gjt.sp.jedit.MiscUtilities	28
org.gjt.sp.jedit.syntax.TokenMarker	24

Class	*in-degree*
org.gjt.sp.jedit.jEdit	282
org.gjt.sp.util.Log	159
org.gjt.sp.jedit.GUIUtilities	135
org.gjt.sp.jedit.View	102
org.gjt.sp.jedit.Buffer	73
org.gjt.sp.jedit.MiscUtilities	62
org.gjt.sp.jedit.io.VFSManagerX	51
org.gjt.sp.jedit.textarea.JEditTextArea	49
org.gjt.sp.jedit.buffer.JEditBuffer	45
org.gjt.sp.jedit.bsh.SimpleNode	44

and the largest made by the laboratory. The program

Table 7: In-degree evolution - Commercial software 1.0 and 1.18

Class	*in-degree*
A	808
B	558
C	551
D	314
E	291
F	287
G	283
H	271
I	265
J	248
-	-
-	-

Class	*in-degree*
A	912
C	631
B	595
X	385
D	347
E	341
F	317
H	295
G	291
Y	285
I	275
J	258

was built using the three-tier architecture and has more than 6,000 classes, which are divided into 6 packages. In this work we analyzed data from two of those packages separately that implement the frontier layer and the model layer. Data of the commercial software system are shown in Table 1.

Software measurements were collect by a tool called Connecta [4] that generates a file in appropriate format for Pajek [23], a network analysis tool.

5. EXPERIMENTS AND RESULTS

In this section, we present and analyze the results of our experiments in the following sequence: the growth of software systems, the network density evolution, the network diameter evolution, the in-degree evolution, the evolution of classes with high in-degree, and the macroscopic view of the software network. Data of the software systems evolution are shown in Table 2 and 3.

5.1 Software Systems Growth

Table 8: Instability of a class of Kolmafia with highest in-degree

Version	*in-degree*	*CoCo*	*# public fields*	*# public methods*
0.2	7	0,5	0	18
2.0	69	0,33	0	30
5.0	142	0,143	0	74
11.0	145	0,067	8	85
13.7	264	0,05	17	78

We analyze the size of a software system by means of the number of classes. The number of classes in an open source software system grows drastically. In 50% of the analyzed programs, the final version has more than twice the number of classes of the first version. Our findings are according to others works which claim that continuing growth is a dominant characteristic of open source software system [9, 14, 19, 11]. This characteristic is also observed in the commercial software system analyzed in this work, however in a smaller scale. A possible explanation for this fact is that an open source software system is on an environment that may be more dynamic than most of the commercial software.

5.2 Software Network Density

Our study points out that the density of the network of classes within a software system decreases as the software system grows. In terms of software construction, this means that a new class inserted in the software system tends to be connected to a very low number of other classes.

5.3 Diameter

The small-world effect is related to the fact that most pairs of vertices seem to be connected by a very short path. This effect has as consequences the determination of some behaviors of the network. For instance, in a social network the small-world effect implies that the propagation of infor-

Table 5: In-degree evolution - Kolmafia 6.0 and 12.0

Class	*in-degree*
net.sourceforge.kolmafia.KoLmafia	69
net.java.dev.spellcast.utilities.LockableListModel	34
net.sourceforge.kolmafia.KoLRequest	30
net.sourceforge.kolmafia.AdventureResult	26
net.java.dev.spellcast.utilities.ActionVerifyPanel	26
net.sourceforge.kolmafia.KoLFrame	26
net.sourceforge.kolmafia.KoLFrame$KoLPanel	25
net.sourceforge.kolmafia.AdventureFrame	24
net.sourceforge.kolmafia.KoLCharacter	20
net.sourceforge.kolmafia.KoLSettings	16

Class	*in-degree*
net.sourceforge.kolmafia.KoLmafia	264
net.sourceforge.kolmafia.KoLConstants	255
net.sourceforge.kolmafia.persistence.Preferences	226
net.sourceforge.kolmafia.utilities.StringUtilities	203
net.sourceforge.kolmafia.RequestThread	193
net.sourceforge.kolmafia.AdventureResult	188
net.sourceforge.kolmafia.KoLCharacter	187
net.sourceforge.kolmafia.RequestLogger	165
net.java.dev.spellcast.utilities.LockableListModel	151
net.sourceforge.kolmafia.textui.command.AbstractCommand	149

Table 6: In-degree evolution - JUnit 4.0 and 4.8.1

Class	*in-degree*
org.junit.runner.Description	12
org.junit.runner.notification.RunListener	9
org.junit.runner.Runner	8
org.junit.runner.notification.RunNotifier	8
junit.framework.TestResult	8
org.junit.runner.notification.Failure	8
org.junit.runner.Request	7
org.junit.runner.manipulation.Filter	6
org.junit.runner.notification.RunNotifier$SafeNotifier	6
junit.framework.TestSuite	5

Class	*in-degree*
org.junit.runner.Description	18
org.junit.runners.model.Statement	16
org.hamcrest.BaseMatcher	15
org.junit.runners.model.FrameworkMethod	15
org.junit.runner.notification.Failure	11
org.junit.runner.notification.RunListener	10
org.junit.runner.notification.RunNotifier	10
org.junit.runners.model.RunnerBuilder	9
org.junit.runners.model.TestClass	9
org.junit.runner.manipulation.Filter	9

Table 9: Instability of classes with in-degree in the commercial software

Class	*Version*	*in-degree*	*CoCo*	*# public fields*	*# public methods*
A	1.0	808	1	0	23
	1.18	912	1	0	25
B	1.0	558	0,071	0	70
	1.18	595	0,067	0	85
C	1.0	551	0,045	0	93
	1.18	631	0,037	1	114
D	1.0	314	0,036	0	105
	1.18	347	0,031	0	116
E	1.0	291	0,05	0	104
	1.18	341	0,048	0	105

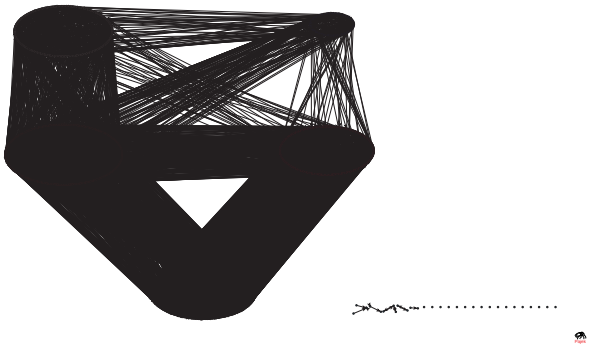
mation will be very fast. If the subject of the study of the network is the spread of diseases, the small-world effect implies the time it takes for a disease to spread throughout a population [21]. The diameter, which is the length of the longest shortest path between two vertices in the network, is a metric that indicates this effect. The results of our experiments reveal that the diameter of a software network grows slowly as the size of system grows and, thus this kind of network has small diameter. This revelation leads some interesting insight about the dynamics of processes taking place on such networks. An error in a class would spread very fast through the classes in a software system. Besides, a modification in a class would rapidly spread, demanding modifications throughout the entire software system.

5.4 In-degree

The analysis of our results reveals that the classes with highest in-degree tend to keep this condition as the software system grows. For all the software systems analyzed in this study, we verified the behavior of the 10 classes with the highest in-degree. We observed that this group of classes is roughly the same throughout the software life. Tables 4, 5, 6 and 7 show the data of evolution of the classes with highest in-degree in 4 software systems, including 2 applications, 1 library and the commercial system. The conjunction of this finding with the fact that a new class inserted in the software system tends to be connected to a very low number of other classes leads to an valuable revelation about the process of software system growth: a new class inserted in the system is preferentially attached to a class that has high in-degree.

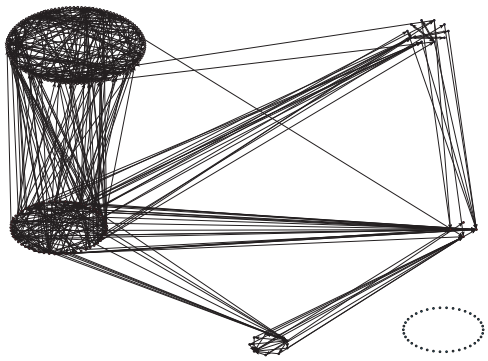
5.5 Evolution of Classes with High In-degree

One can argue that classes with high in-degree is stable, since those classes are provider of services and so should be well defined, constructed and tested. Stability, here, is defined as the lack of modifications in a class during the life of the software system. Intuitively it will be possible to conclude that if the system is well designed and the open-closed principle was appropriately applied, those classes will suffer little modifications or none. However our finds shows that the opposite occurs. Classes with high in-degree are extremely unstable. We evaluated the modifications of a class between two consecutives versions by means of three metrics: the number of public fields, the number of public methods, and cohesion. Table 8 shows data of a class of an open-source software system and Table 9 shows data of a class of the commercial application.In each new version, the classes with higher in-degree grow in number of public



(a)

Figure 2: Hibernate (version 3.5.1) network modeled by little house



(a)

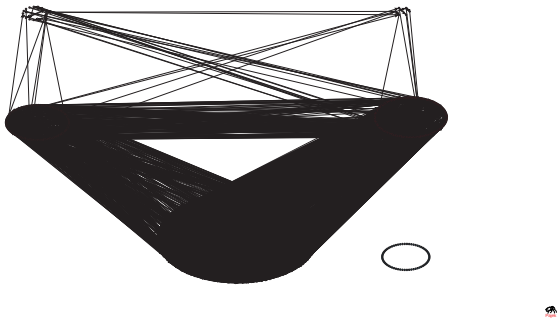
Figure 3: JUnit (version 4.8.1) network modeled by little house

methods and sometimes in number of public fields. Moreover, their cohesion decreases over the time. A reasonable explanation for this process is that, how these classes are great services providers, the usual practice is to keep them in this status by including new services in them in order to attend the new classes. This causes the degradation of the classes cohesion, that influences the deterioration of the system.

By those results, we conclude that the process of software evolution occurs in the following way: as a new class in inserted in the system, instead of refactoring the system [7], the common practice is usually to aggregate new services in the older classes. This leads to the swelling of the classes that already have lot of clients, and so they become less cohesive and have its in-degree always growing. The non-refactoring practice can hence be the cause of the small-world effect in software systems

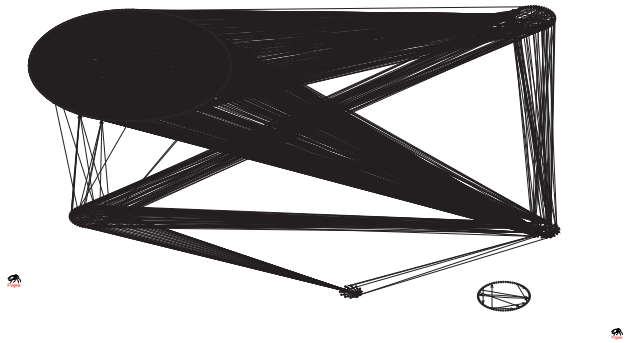
5.6 The Macroscopic Structure of the System Network

For the purpose of investigate whether there is a general macroscopic structure of the software networks, we first fitted some versions of the software systems analyzed in this



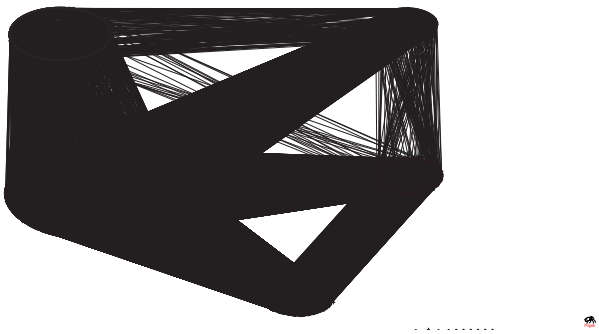
(a)

Figure 4: Kolmafia (version 13.7) network modeled by little house



(a)

Figure 5: The frontier layer of the commercial software (version 1.18) modeled by little house



(a)

Figure 6: The model layer of the commercial software (version 1.18) modeled by little house

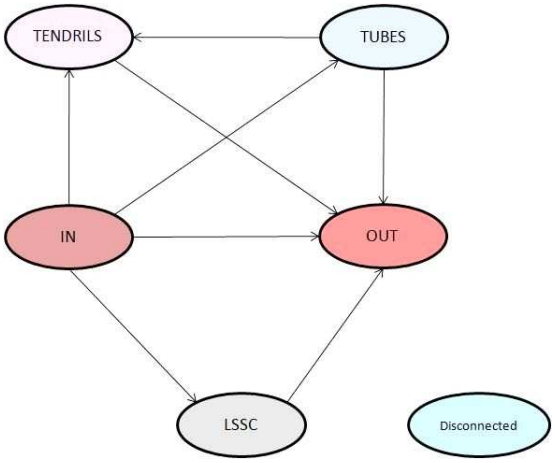


Figure 7: The generic macroscopic structure of system software network

work to the bow-tie model. This analysis was carried out using Pajek, which also generates an image of the network and allows manipulation of it. Each group from the bow-tie model corresponds to a component in the network. We drew the picture of network by grouping the nodes into their respective components. By manipulating those images, we find out that the connections between the components within the network form an interesting image that match a well-known graph. Figures 2, 3 and 4 show the resulting images of 3 open-source software systems: a framework, a library and an application. Figures 5 and 6 show the resulting image of the frontier layer and the model layer of the commercial application. The macroscopic structure of software networks that we identified is shown in Figure. We call it the *little house*. We preserve the same terminology of components employed in the bow-tie model. The SCC is the central core in which every single class reaches the other ones within the component. IN is the group of classes that cannot be reached for any class within the system and can reach classes in SCC, tendril, out and tubes. OUT is the group of classes that can be reached for all the other components but cannot reach any class within the system. Classes that connect a class within IN to a class within OUT constitute tendril component. This result is massively observed in the data set evaluated in this work what evidences its consistence. New questions arise from this finding: the kind of classes that compose a component; the size of the giant strongly connected component and the manner it evolves; the implications of this model in the management and in the maintenance of software systems.

SCC plays a central role in the system since its classes are strongly connected one to another, what can make this component hard to be understood, tested and maintained. Table shows the growth of SCC in number of classes and in the percentile of the classes within the software system that compose SCC. From this data we draw that the number of classes in SCC tends to increase. 11 software systems of the sample increased 3 times or more in number of classes and in 7 of those programs are observed that the percentile of classes in SCC also increased substantially.

It could be thought that the connections among the identified components in a software network should be related to the multi-tier architecture. However we did not found evidences to support this hypothesis from our experiments. A counter-example of this appears in the analysis of the data from the commercial software of our study. This system was constructed under the multi-tier architecture and we analyzed the frontier layer and the model layer separately. In both cases, the relationship among classes within the system can be modeled by the small house.

The macroscopic structure identified in software systems brings novel information to software engineers about the nature of their work subject, especially in the sense of software maintenance and test. The presence of a giant strongly connected component emphasizes the need of systematic approach of maintenance tasks in the classes of this component, because a modification in a class within this component can be widely spread throughout the entire system. Knowing the way that classes are connected one to another can lead improvements in test techniques in such way that test tasks can be more efficient. Furthermore the model can be use as basis to generate artificial data to be used by software engineering researchers that usually face problems with finding data from software systems to validate their algorithms and models.

6. CONCLUSION

This paper presented the results of a study about software evolution characterization based on software metrics, and concepts and metrics of network. We analyzed 16 open-source software systems and 1 commercial application, in a total of 114 versions. The empirical observation of data shows that: the density of software network tends to decrease as the software grows; the diameter of such networks is short; the classes with highest in-degree tend to keep this status; such classes are unstable, since they grows in number of public methods and sometimes in number of public fields, and their internal cohesion degrade. Those observations yield important insight about the nature of software evolution. How the density tends to decrease and the classes with highest in-degree tend to have even higher in-degree, we infer that the common practice is to insert new requirements into such classes instead of refactor the system in order to introduce the new requirements. Thus the non-refactoring practice could be the reason of the small-world phenomenon in software networks and its implications. The small diameter of the software networks can lead a process in which a error or a maintenance task performed in class within the system spreads widely through it.

Our investigations reveal an interesting picture that model the macroscopic structure of software networks, which we called the little house. This macroscopic view of the systems can support software engineers in the management, maintenance and test tasks. It also can be applied in the construction of artificial data that can improve researches in software engineering.

There is much to be done in understanding the processes taking place in software systems. Further works needs to be carried out in order to expose details about the nature of the classes that compose each component in macroscopic structure identified in this paper as well the forces that make this kind of relationship among classes appears.

7. ACKNOWLEDGMENTS

This work was sponsored by FAPEMIG-Brazil, as part of the project CONNECTA Process: CEX APQ-3999-5.01/07.

8. REFERENCES

[1] R. Abreu, Fernando Brito; Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of 4th Int. Conf. of Software Quality*, McLean, VA, USA, Outubro 1994.

[2] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Undertanding the shape of java software. In *OOPSLA'06*, Oregon, Portland, USA, Outubro 2006.

[3] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *WWW9 Conference*, pages 309–320, May 2000.

[4] K. A. M. FERREIRA. *Avaliação de Conectividade em Sistemas Orientados por Objetos*. DCC/UFMG, Belo Horizonte, Junho 2006.

[5] K. A. M. Ferreira, M. A. S. Bigonha, R. da S. Bigonha, H. Corrêa, and L. F. de O. Mendes. Valores referência de métricas de software orientado por objetos. In *Simpósio Brasileiro de Engenharia de Software*, Fortaleza, Ceará, 2009.

[6] K. A. M. Ferreira, H. Correa, M. Bigonha, and R. Bigonha. *Cohesion by Concern*. Technical Report - Federal University of Minas Gerais, Brazil, 2009.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] M. Godfrey and Q. Tu. Growth, evolution, and structural change in open source software. In *Proc. of the IWPSE*, pages 103–106, Vienna, Austria, 2001.

[10] I. Herraiz, G. Robles, and J. u. M. Gonzalez-Barahon. Comparison between slocs and number of files as size metrics for software evolution analysis. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 206–213, Washington, DC, USA, 2006. IEEE Computer Society.

[11] A. Israeli and D. G. Feitelson. The linux kernel as a case study in software evolution. *The Journal of Systems and Software*, 83(3):485–501, 2010.

[12] S. Jenkins and S. R. Kirk. Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Inf. Sci.*, 177(12):2587–2601, 2007.

[13] L. Jing, H. Keqing, M. Yutao, and P. Rong. Scale free in software metrics. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 229–235, Washington, DC, USA, 2006. IEEE Computer Society.

[14] S. Koch. Software evolution in open source projects—a large-scale investigation. *J. Softw. Maint. Evol.*, 19(6):361–382, 2007.

[15] Y. Lee, J. Yang, and K. H. Chang. Metrics and evolution in open source software. In *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, pages 191–197, Washington, DC, USA, 2007. IEEE Computer Society.

[16] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. of the Fourth Intl. Software Metrics Symposium (Metrics'97)*, 1997.

[17] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1), March 2007.

[18] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18(1), Setembro 2008.

[19] T. Mens, J. Fernández-Ramil, and S. Degrandsart. The evolution of eclipse. In *Proc. 24th Int'l Conf. on Software Maintenance*, pages 386–395, October 2008.

[20] B. MEYER. *Object-oriented software construction*. Prentice Hall International Series, Estados Unidos, 2 edition, 1997.

[21] M. E. J. Newman. The structure and function of complex networks. *SIAM Reviews*, 45(2):167–256, 2003.

[22] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 390–400, Washington, DC, USA, 2009. IEEE Computer Society.

[23] PAJEK. *Networks / Pajek Program for Large Network Analysis - for Windows*. Acesso em Setembro de 2009.

[24] A. Potantin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in oo programas. *Communications of the ACM*, 48(5):99–103, Maio 2005.

[25] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 7th edition edition, 2009.

[26] D. Puppin and F. Silvestrini. The social network of java classes. In *SAC'06*, pages 1409–1413, Dijon, França, 2006.

[27] R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Proceedings of 3rd International Workshop on Source Code Analysis and Manipulation (SCAM)*, Setembro 2003.

[28] G. Xie, J. Chen, and I. Neamtiu. Towards a better undertanding of software evolution: An empirical study on open source software. In *ICSM*, Edmonton, Canada, 2009.

[29] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 531–540, New York, NY, USA, 2008. ACM.