Software Evolution Characterization - A Complex Network Approach

Kecia A. M. Ferreira[1], **Mariza A. S. Bigonha**[2], **Roberto S. Bigonha**[2], **Bárbara M. Gomes**[1]

[1]Department of Computing – CEFET-MG
Av. Amazonas, 7675 – Nova Gameleira – Belo Horizonte-MG – Brazil

[2]Department of Computer Science – UFMG
Av. Antônio Carlos, 6627 – Pampulha – CEP: 31270-010 – Belo Horizonte-MG

{kecia,mariza,bigonha}@dcc.ufmg.br

Abstract. *Software evolution has been the subject of research in the last decades, revealing 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. This paper presents the results of a study about software evolution characterization based on concepts of Complex Networks. We analyzed 16 open-source software systems and one commercial application, in a total of 129 versions. The results of this study show that: the density of a software network decreases as the software system grows; the diameter of such networks is short; the classes with higher in-degree keep this status; such classes are unstable and their internal cohesion degrades. Our investigations also revealed an interesting picture which models the macroscopic structure of software networks. We called it* the little house.

1. Introduction

Despite all the knowledge about high-quality software construction consolidated in well-known principles, criteria, rules, design-patterns and techniques, 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's Laws [Lehman et al. 1997] describe this evolutive nature of software by postulating that every software system grows and suffers maintenances continuously, has increasing complexity and decreasing quality throughout its evolution.

Most of the researches carried out to describe software evolution are concerned in investigating whether the Lehman's laws are applied in open source software, especially in the growth and complexity aspects [Koch 2007, Xie et al. 2009, Israeli and Feitelson 2010]. Growth has been usually evaluated by means of metrics such as LOC or number of files [Godfrey and Tu 2001, Herraiz et al. 2006], while complexity has been evaluated by means of McCabe or Halstead complexity metric [Mens et al. 2008, Israeli and Feitelson 2010]. A few researches have used other software metrics to study software evolution, for instance: number of deleted/added/changed files [Mens et al. 2008], coupling and cohesion metrics [Lee et al. 2007]. Recently, the concepts of complex networks have been timidly applied to understand the behavior and the nature of software structure [Jing et al. 2006, Jenkins and Kirk 2007, Louridas et al. 2008, Zimmermann and Nagappan 2008]. Common findings in such

works are that the *in-degree* of vertices 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 [Newman 2003]. However, there is still a great lack of solid knowledge about the evolution of software systems design.

In this work, we carried out an exploratory study to investigate how the design of object-oriented program evolves, by applying Complex Network concepts. An object-oriented program can be modeled as a network in which the vertices correspond to classes, and the edges correspond to relationships between classes. Such networks are referenced in this paper as *software networks*. The aim of this study is to get insights on how software networks evolve in terms of: *density*, which measures how connected one to another are the vertices in a network; *diameter*, which is a measure of the distance between vertices in a network; *in-degree*, which is the number of vertices in the network which depend upon a given vertex. These metrics are considered in this work because they express properties of the dependence among vertices in the network, which is an important aspect of the quality of software design. We also investigate how the internal quality and the size of the central classes of a software system evolute over the time. The central classes of a system are those which have a high number of dependent classes, i.e, a high *in-degree*. The research questions investigated in this study are the following: (1)How the density of software networks evolves? (2)Is the diameter of software networks short? (3)Which are the central classes in a software system? (4)Does the internal quality of such classes degrade over the time? (4)Is there a generic macroscopic figure of software network?

The data set analyzed in this work is from 16 open source object-oriented software systems and from one commercial object-oriented software system, in a total of 129 versions of the programs. Our analysis yields a novel insight into the evolution of software system structure: the classes with a higher in-degree tend to maintain this status as the software system grows, they also gain more methods and have declining internal cohesion; the network of modules within a software system has short *diameter* and shrinking *density*. Even more interesting, our analysis reveals the picture of the macroscopic structure of software systems. By analysing this pattern of software networks, we also find that such networks have a strongly connected component which enlarges as the software system grows. The findings of this study identify properties of software design evolution which are not described by previous works. They can be used for improving software development tasks, such as maintenance and test plans.

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

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 paper.

2.1. Metrics

Coupling among modules is an important aspects of software design quality, because the high coupling among modules in a software system makes the software design more complex and rigid. Due to the importance of this aspect, we consider the following network metrics which evaluate the connectivity among vertices in a network:

- Network density: in a network with a edges and n vertices and without self loops, this metric is given by $a/(n(n-1))$ [Leskovec et al. 2007]. In the context of software metrics, this metric is called COF (coupling factor) [Abreu and Carapua 1994].
- 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 [Newman 2003].
- In-degree: the in-degree of a vertex is given by the number of vertices from which there is an edge that reaches the vertex [Newman 2003]. In a software system network, it is the number of classes that use services of a given class or extend it.

In this work, we explore how the central classes evolve in terms of size and internal quality. The size of a class is evaluated by means of the number of public methods and number of public fields. We considered these metrics because they represent the size of the interface of the class, what will reveal whether a class is changed over the time to adapt to connections with other classes. The internal quality of a class is evaluated by means of a cohesion metric, because cohesion is one of the most important aspects of modularity. There are 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 Responsibility (COR) [Ferreira 2011], which is a different interpretation of LCOM4 [Hitz and Montazeri 1995]. This metric is given by 1/C, where C is the number of disjointed sets 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 the class. If a method *a* is similar to a method *b*, and *b* is similar to a method *c*, then *a* is also similar to *c*. For instance, if there are two sets in a class, COR will result in 0,5. This indicates that the class has 2 responsibilities. If there is only one set in the class, COR will result in 1, indicating a high cohesion.

2.2. Complex Networks

Empirical observation of real networks yielded valuable comprehension of such networks. The work of Newman [Newman 2003] presents a wide review about advances in the field of Complex Networks. The study of properties of networks includes concepts such as the small-world phenomenon, degree distributions, scale-free networks and models of network growth. Models of networks help us understanding network topology and the processes taking place inside networks. In this work, we explore the structure of software system networks using concepts and characteristics of complex networks.

Networks with power-law degree distribution are referred as scale-free networks. A power-law is a probability distribution function in which the probability of a random

variable X take a value x is proportional to a negative power of x, denoted by $P(X = x) \propto cx^{-k}$. In a scale-free network there is a large 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 [Wheelson and Counsell 2003, Puppin and Silvestri 2006, Baxter et al. 2006, Louridas et al. 2008]. An important property of a scale-free network is its resilience to the removal of their vertices. A study on vertex deletion in the Internet and Web shows 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 [Newman 2003]. Since software systems networks are also scale-free, this property might 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 in which most pairs of vertices are connected by a short path. This is related to the easiness of information propagation in the network. Depending on the kind of network, information should assume different meaning, such as the spread of a disease in a population, dissemination of a rumor in a social network, or an change propagation in a software system network.

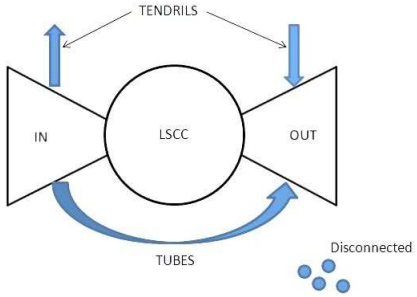


Figure 1. The bow-tie model of the Web

2.3. The Bow-tie Model

Broder et al. [Broder et al. 2000] have concluded that the macroscopic structure of the Web can be modeled by a picture known as bow-tie, shown in Figure 1. By this model, web pages can be divided into five groups: *LSCC*, *in*, *out*, *tendrils*, *tubes* and *disconnected*. It reveals that in the Web graph there is a central core in which all pages can reach one another. This central core is called the giant strongly connected component (LSCC). Another group of pages can reach the ones in LSCC but cannot be reached by them. This group is called *in*. *Out* consists of pages that can be reached from *LSCC* but cannot link it back. *Tendril* consists of pages that cannot reach *LSCC* and are not reachable by it; pages in tendrils can be reached by *in* and 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*. This model has important applications in studies of the web, such as the analysis of web algorithms and the prediction of the evolution of web structures. 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 classes in a object-oriented software system connect to one another.

3. Related Work

One of the most noted works in the field of software evolution resulted in the Lehman's laws which include: continuing change, increasing complexity, continuing growth and declining quality [Lehman et al. 1997]. Many researchers have recently studied whether these laws can be applied to open-source software systems. Mens et al. [Mens et al. 2008] 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. Israeli and Feitelson [Israeli and Feitelson 2010] used software metrics in order to analyze Linux kernel evolution. The results of their study support most of Lehman's laws, however they observed that functions within the program have a decreasing average complexity. Xie et al. [Xie et al. 2009] evaluated the evolution of 7 open source software systems. The results of their study demonstrate that the following Lehman's laws are applicable 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 the source code.

Software evolution has been usually studied by means of software system growth. Koch [Koch 2007] analyzed the growth of a large sample of open source software systems, concluding 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. [Herraiz et al. 2006] 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.

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 follows a power-law [Wheelson and Counsell 2003, Baxter et al. 2006, Louridas et al. 2008]. Jenkins and Kirk [Jenkins and Kirk 2007] 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 [Zimmermann and Nagappan 2008] found that measures from network analysis, such as centrality and closeness, can predict defects for binaries of Windows Server 2003.

Despite the notable contribution of the works carried out to characterize software evolution, there are still open questions about this phenomenon. Most of the researches in this field are limited to studying the growth of software systems in terms of lines of code and number of modules or files. A few of them evaluate software evolution by means of other software metrics. Since a class is the basic component of an object-oriented software, we investigate how software systems evolve in terms of number of classes. We study how two important network measures, density and diameter, behave as the software system grows. Classes with higher in-degree play a central role in the software system. We explore how such classes evolve in terms of in-degree, internal cohesion, number of public methods and number of public fields. The results of the research carried out reveal important properties of the software system evolution process. We also identify

Table 1. Software systems analyzed in the study

Name	*Category*	*# downloads/week*	*Age*	*#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	10	10
Commercial - model layer	Commercial application	-	2005 - 2010	3343 a 4031	10	10

and analyze the macroscopic structure of object oriented software systems.

4. Methodology

The selection of the open-source software systems analyzed in the study was based on the following criteria: age, quantity of versions or releases, and category. The data were 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, satisfying the following criteria: they were developed in Java, they have at least 5 versions or releases and they are 4 years old 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 highest popularity, highest number of versions or releases and with highest age. Popularity was evaluated through 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 to April 2010.

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

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	0.4.0	242	525	0,009	8
Reports	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	10a1	171	494	0,017	10
library	10a2	186	516	0,015	7
	10b1	203	615	0,015	7
	10b2	218	662	0,014	9
	10b3	270	872	0,012	9

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	1.6	64	149	0,037	6
Movie	1.7	73	168	0,032	6
Manager	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	1.0	424	717	0,004	15
sql	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

The commercial software system analyzed in this work is developed by a software engineering laboratory of an important Brazilian university. This laboratory provides software and consulting solutions for different market segments. Most of its clients are Brazilian government agencies. The software system selected for analysis is one of the oldest and largest made by the laboratory. The program was built using the three-tier architecture and has more than 6,000 classes, which are divided into 6 packages. We analyzed data from two of those packages which implement the frontier layer and the model layer. These layers were analyzed separately for the convenience of the laboratory, which performed the data collection. Data of the commercial software system are shown in Table 1.

Software measurements were collected by Connecta [Ferreira 2006] which generates a file in appropriate format for Pajek [PAJEK 2010], a network analysis tool.

Table 3. Evolution of the open source software systems

Software	*Version*	*#Classes*	*#Connections*	*COF*	*Diameter*
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*
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

5. Experiments and Results

In this section, we present and analyze the results of our experiments. Data on the software systems evolution are shown in Tables 2, 3 and 4.

5.1. Software Systems Growth

We analyze the size of a software system through its 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 in the first version. Our findings accord to other works which claim that continuing growth is a dominant characteristic of open source software systems [Godfrey and Tu 2001, Koch 2007,

Table 4. 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

Mens et al. 2008, Israeli and Feitelson 2010]. This characteristic is also observed in the commercial software system analyzed in this work, however at a smaller scale. A possible explanation for this fact is that an open source software system is in an environment that may be more dynamic than most of the commercial software.

5.2. Diameter

The small-world has as consequence some network behaviors. For instance, in a social network the small-world effect implies that the propagation of information will be very fast. If the subject of the study is the spread of diseases, the small-world effect implies the time it takes for a disease to spread throughout a population [Newman 2003]. The diameter is a metric that indicates this effect. The results of our experiments reveal that the diameter of a software network is small initially and grows only slowly, so it remains rather small. This reveal that the distance between two classes is small. Hence, a change, for instance, in a class would widely spread, demanding changes throughout the software system.

5.3. 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 into the software system tends to be connected to a very low number of other classes.

Table 5. The highest in-degree evolution - Freecol 0.1.0 and 0.9.2

Class	*in-degree*
net.sf.freecol.client.FreeColClient	84
net.sf.freecol.common.model.Unit	84
net.sf.freecol.client.gui.Canvas	80
net.sf.freecol.common.model.Player	78
net.sf.freecol.common.model.Game	67
net.sf.freecol.common.model.Tile	61
net.sf.freecol.client.gui.i18n.Messages	59

Class	*in-degree*
net.sf.freecol.client.FreeColClient	214
net.sf.freecol.client.gui.Canvas	208
net.sf.freecol.client.gui.i18n.Messages	174
net.sf.freecol.common.model.Player	158
net.sf.freecol.common.model.Unit	148
net.sf.freecol.common.model.Tile	131
net.sf.freecol.common.model.Game	131

5.4. In-Degree

The analysis of our results reveals that the classes with highest in-degree maintain this property as the software system grows. We observed that the group of the 10 classes with the highest in-degree is roughly the same throughout the software life. Tables 5, 6 and 7 show the data of the classes with the highest in-degree in Freecol, Hibernate, and

Table 6. The highest in-degree evolution - Hibernate 3.0 and 3.5.1

Class	*in-degree*
org.hibernate.HibernateException	86
org.hibernate.util.StringHelper	81
org.hibernate.dialect.Dialect	58
org.hibernate.engine.PersistenceContext	54
org.hibernate.MappingException	49
org.hibernate.util.ArrayHelper	47
org.hibernate.engine.Cascades	46
org.hibernate.Hibernate	45
org.hibernate.util.ReflectHelper	42
org.hibernate.AssertionFailure	38

Class	*in-degree*
org.hibernate.HibernateException	174
org.hibernate.util.StringHelper	139
org.hibernate.dialect.Dialect	97
org.hibernate.MappingException	87
org.hibernate.mapping.PersistentClass	78
org.hibernate.util.ReflectHelper	73
org.hibernate.Hibernate	68
org.hibernate.AssertionFailure	65
org.hibernate.util.ArrayHelper	59
org.hibernate.mapping.Property	59

Table 7. The highest 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

the commercial system. This finding, associated to 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 a 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 Higher In-Degree

One can argue that classes with high in-degree are stable, since those classes are services provider of services and so should be well defined, constructed and tested. Stability, here, is defined as the low frequency 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 [Meyer 1997] was appropriately applied, those classes will suffer none or little modifications. Nevertheless our findings show that the opposite occurs. Classes with a higher in-degree are extremely unstable. In the commercial software system, however, this property is moderate. 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

Table 8. Evolution data of the class from KoLmafia with the highest in-degree

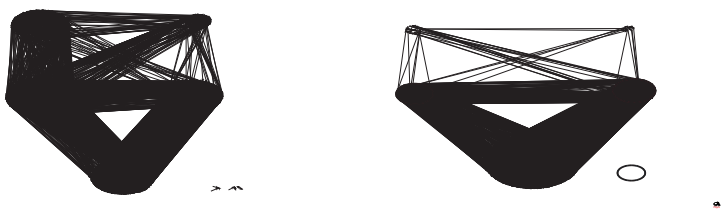
Version	in-degree	COR	public fields	public methods
0.2	7	0,5	0	18
2.0	69	0,33	0	30
5.0	142	0,143	0	74
11.0	145	0,067	8	85
13.7	264	0,05	17	78

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

Class	*Version*	*in-degree*	*COR*	*# 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

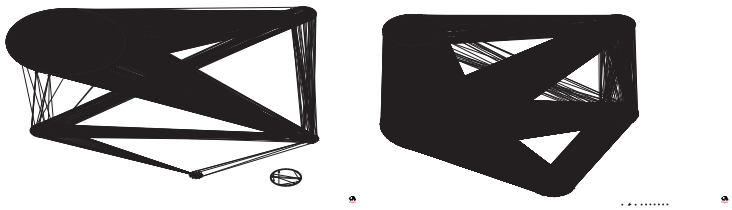
application. In each new version, the classes with higher in-degree grow in number of public methods and sometimes in number of public fields. Moreover, their cohesion decreases over time. A reasonable explanation for this behavior is that due to the fact that these classes are such great service providers, keeping them in this status by including new services to attend new classes is the usual practice adopted. This causes the degradation of the class cohesion, which influences system deterioration.

By those results, we inferred that the process of software evolution occurs in the following way: as a new class is inserted in the system, instead of refactoring the system [Fowler 1999], the common practice is usually to aggregate new services in the older classes. This leads to the swelling of the classes which have already a lot of clients, so they become less cohesive and have a growing in-degree. The non-refactoring practice might hence be the cause of the small-world effect in software systems



(a) (b)

Figure 2. (a) Hibernate (version 3.5.1) network and (b) Kolmafia (version 13.7) network modeled by little house



(a) (b)

Figure 3. (a) The frontier layer and (b) the model layer of the commercial software (version 1.18) modeled by little house

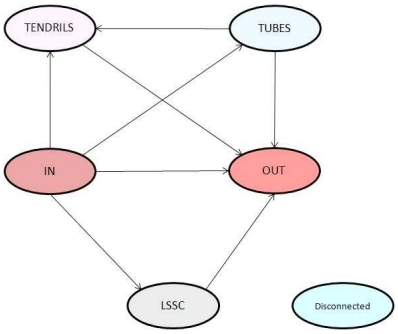


Figure 4. *Little house* – The generic macroscopic structure of software network

5.6. The Macroscopic Structure of Software System Networks

For the purpose of discovering a general macroscopic structure of software system networks, we first fitted to the bow-tie model some versions of the software systems analyzed in this work. This analysis was carried out by using Pajek, which also generates an image of the network and allows its manipulation. 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 matches a well-known graph. Figure 2 shows the resulting images of Hibernate and Kolmafia. Figure 3 show the resulting images of the frontier layer and the model layer of the commercial application. The macroscopic structure of software networks we identified is shown in Figure 4. We call it the *little house*. This result was massively observed in the data set evaluated in this work. We preserved the same terminology of components employed in the bow-tie model. The *little house* model is a graph in which a node corresponds to a specific group of classes. In each of these groups, except the *disconnected* node, classes are freely connected one to another. The *little house* is constituted the following nodes. **LSCC**: is the largest strongly connected component of the software system. In this component, any class can reach all the other classes of *LSCC*. Therefore, every class in *LSCC* depends upon all the other classes in *LSCC*, directly or indirectly. **In**: classes from *in* can use any other class of the software system, but they are not used by the classes which are not in this component. **Out**: classes from *out* can be used by any other class of the software system, but they use only classes which are in this component. **Tendril**: classes from *tendril* use only classes from this component or from *out*. Besides, a class from *tendril* can be used only by classes from *tendril*, *tubes* or *in*. **Tubes**: classes from *tendril* use only classes from this component, *out* or *tendril*. Besides,

Table 10. Evolution of LSCC

Software	LSCC first version (#classes)	LSCC first version (% of the size)	LSCC last version (#classes)	LSCC last version (% of the size)
Jsch	16	20	16	14
LogSim	289	32	303	25
Jml	17	10	27	10
JavaGroups	9	2	13	1
DBUnit	16	8	24	7
Hibernate	100	10	477	20
Squirel	40	10	579	47
Junit	22	28	9	4
Jedit	69	20	395	35
Phex	165	41	403	29
Jgnash	276	35	322	11
DrJava	43	2	968	26
Jasper	20	8	150	9
MovieManager	50	78	92	18
FreeCol	5	11	758	70
KolMafia	15	38	748	67

a class from *tubes* can be used only by classes from *tubes* or *in*. **Disconnected**: a class in this component have no connection with other classes.

LSCC 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. Data of the evolution of this component, shown in Table 10, indicates that LSCC enlarges over the time. Eleven software systems of the sample increased three times or more in number of classes and in seven of those programs the percentile of classes in LSCC 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 find evidences to support this hypothesis from our experiments. A counter-example of this appears in the analysis of the data from the commercial software. 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 *little 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 testing. 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 system. Knowing the way classes are connected one to another can lead to improvements in test techniques in such way those test tasks can be more efficient. Furthermore the model can be used 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 presents the results of a study about software evolution characterization based on concepts of Complex Networks. We analyzed 16 open-source software systems and one commercial application, in a total of 129 versions. The empirical observation of data shows that: the density of software network decreases as the software system grows; the diameter of such networks is short; the classes with higher in-degree keep this status; such classes are unstable, since they grow in number of public methods and sometimes in number of public fields, and their internal cohesion degrades. Those observations yield important insight about the nature of software evolution. How the density tends to decrease and the classes with higher in-degree tend to have even higher in-degree, we inferred that the common practice is to insert new requirements into such classes instead of refactoring the system in order to introduce the new requirements. Thus the non-refactoring practice might be the reason of the small-world phenomenon in software networks and its implications. The small diameter of a software network, for instance, can lead to ripple effects of errors or maintenance changes.

Our investigations revealed an interesting picture which models the macroscopic structure of software networks. We called it *the little house*. We envision that the results of this study can be used to improve software development tasks, such as maintenance and test plans, and also can be applied in the construction of artificial data to support research in Software Engineering. There is much to be done in understanding the processes taking place inside software systems. Further works need to be carried out to expose details about the nature of the classes which compose each component in the macroscopic structure identified in this paper as well as the forces that make appear this kind of relationship among classes.

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

References

Abreu, F. B. and Carapua, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of 4th Int. Conf. of Software Quality*, McLean, USA.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. (2006). Understanding the shape of java software. In *OOPSLA '06*, pages 397–412, Portland, USA. ACM.

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

Ferreira, K. A. M. (2006). *Avaliao de Conectividade em Sistemas Orientados por Objetos*. Master Thesis - DCC/UFMG., Belo Horizonte, Brazil.

Ferreira, K. A. M. (2011). *Um Modelo de Predio de Amplitude da Propagao de Modificaes Contratuais em Software Orientado por Objetos*. PhD Dissertation - DCC/UFMG., Belo Horizonte, Brasil.

Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code*. Addison Wesley.

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

Herraiz, I., Robles, G., and Gonzalez-Barahon, J. M. (2006). Comparison between slocs and number of files as size metrics for software evolution analysis. In *CSMR '06*, pages 206–213, Washington, DC, USA. IEEE Computer Society.

Hitz, M. and Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. In *Int. Symposium on Applied Corporate Computing*, Monterrey, Mxico.

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

Jenkins, S. and Kirk, S. R. (2007). Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Information Sciences: an International Journal*, 177(12):2587–2601.

Jing, L., Keqing, H., Yutao, M., and Rong, P. (2006). Scale free in software metrics. In *COMPSAC '06*, pages 229–235, Washington, USA.

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

Lee, Y., Yang, J., and Chang, K. H. (2007). Metrics and evolution in open source software. In *QSIC '07*, pages 191–197, Washington, USA.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. In *Metrics'97*.

Leskovec, J., Kleinberg, J., and Faloutsos, C. (2007). Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1.

Louridas, P., Spinellis, D., and Vlachos, V. (2008). Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18:2:1–2:26.

Mens, T., Fernndez-Ramil, J., and Degrandsart, S. (2008). The evolution of eclipse. In *Proc. 24th Int'l Conf. on Software Maintenance*, pages 386–395.

Meyer, B. (1997). *Object-oriented software construction*. Prentice Hall International Series, Estados Unidos, 2 edition.

Newman, M. E. J. (2003). The structure and function of complex networks. In *SIAM Reviews*, volume 45, pages 167–256.

PAJEK (2010). *Networks / Pajek Program for Large Network Analysis - for Windows*. http://vlado.fmf.uni-lj.si/pub/networks/pajek/.

Puppin, D. and Silvestri, F. (2006). The social network of java classes. In *SAC '06*, pages 1409–1413, New York, NY, Estados Unidos. ACM.

Wheelson, R. and Counsell, S. (2003). Power law distributions in class relationships. In *SCAM'03*, Amsterd, Holanda.

Xie, G., Chen, J., and Neamtiu, I. (2009). Towards a better understanding of software evolution: An empirical study on open source software. In *ICSM'09*.

Zimmermann, T. and Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In *ICSE '08*, pages 531–540, Leipzig, Alemanha. ACM.