The Evolving Structures of Software Systems

Kecia Aline Marques Ferreira
Roberta Coeli Neves Moreira
Department of Computing
Federal Center for Technological Education of Minas Gerais
Belo Horizonte, Brazil
kecia@decom.cefetmg.br, robertacoelineves@gmail.com

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

***Abstract*—Software maintenance is an important problem because software is an evolving complex system. To make software maintenance viable, it is important to know the real nature of the systems we have to deal with. Little House is a model that provides a macroscopic view of software systems. According to Little House, a software system can be modeled as a graph with five components. This model is intended to be an approach to improve the understanding and the analysis of software structures. However, to achieve this aim, it is necessary to determine its characteristics and its implications. This paper presents the results of an empirical study aiming to characterize software evolution by means of Little House and software metrics. We analyzed several versions of 13 open source software systems, which have been developed over nearly 10 years. The results of the study show that there are two main components of Little House which suffer substantial degradation as the software system evolves. This finding indicates that those components should be carefully taken in consideration when maintenance tasks are performed in the system.**

***Keywords*-software metrics; software evolution; complex networks;**

I. Introduction

It is estimated that about 70% of the total cost of a software system is due its maintenance, and a large amount of the effort in this phase is spent in program comprehension. Many factors contribute to making software maintenance laborious and costly. Among them is the evolutive nature of software structures, which is characterized by declining quality and increasing complexity [1]. It is widely known that as a software system evolves, its structure becomes more rigid, its complexity tends to grow, and the software maintenance may become a very difficult task.

Concerning this issue, many works have been carried out in order to provide a better comprehension about the software evolution process. Many of them have investigated software evolution by means of size and complexity, using mainly LOC (lines of code) and McCabe or Halstead complexity metrics. However, especially for object-oriented software, those metrics may not provide a proper assessment of software evolution, and many other software metrics have been proposed in the last years. An emerging research area has considered software system as a Complex Network. It is a consensus in the works carried out in this field that the software system networks exhibit the main characteristic of a Complex Network, the node degree distributions are modeled by a power-law [2]. In spite of the notable efforts in order to understand the nature of software systems structures and their evolution, the knowledge about the characteristics of the real software systems we have to maintain is still deficient.

Aiming to improve the insight into the process of software structures evolution, this work presents the results of an experimental study about object-oriented software evolution by using a novel approach. The evolution of software systems is analyzed by means of a model, called Little House [3], and the thresholds of six object-oriented software metrics, namely number of afferent couplings, LCOM, COR (Cohesion of Responsibility) [4], DIT, number of public fields and number of public methods. Little House is defined in a previous published work of the authors in order to provide a macroscopic view of software structures. This model is based on the well-known Bow-Tie model, which is used to represent the Web graph. According to Little House, classes of an object-oriented system can be grouped in five components, named *In*, *Out*, *LSCC*, *Tendrils* and *Tubes*. In the present work, we investigate how the components of Little House evolve in terms of software metrics, regarding the thresholds of such metrics.

The remaining of this paper is organized as follows. Section II discusses the related work. Section III describes the software metrics used in this work, as well as their thresholds. Section IV describes the method used to perform the data gathering and analysis. Section V shows and discusses the results of the study. Section VI brings the conclusion and indications of future work.

II. Related Work

The Lehman's laws postulate that software system evolution has the following main characteristics: continuing change, increasing complexity, continuing growth and declining quality [1]. Koch [5] analyzed the growth of a large sample of open source software systems, concluding that the mean growth rate of a software system is linear or tends to decrease over time. Herraiz et al. [6] studied the evolution of Eclipse by means of software metrics and found evidences of continuing growth and increasing complexity. Israeli and Feitelson [7] used software metrics in order to analyze the Linux kernel evolution. The results of their study support most of Lehman's laws. Xie et al. [8] evaluated the evolution of seven open source software systems, and have found that the following Lehman's laws are applicable to open-source software systems: continuing change, increasing complexity, self regulation and continuing growth.

Recently, the concepts of Complex Networks have been applied to characterize software systems. A system can be represented by a graph in which modules are the nodes, and the relationship between two modules are the edges. In an object-oriented system, the classes can be taken as the nodes. An important metric in this case is the in-degree of the node, which indicates the number of other classes that depend upon a given class. Many researchers have identified that the in-degree distribution in software system networks follows a power-law [9], [10], [11]. In a power-law distribution, there is a large number of occurrences of low values, and there is a very few occurrences of high values. In an object-oriented software, this means that there are few classes with high in-degree, while most of the classes have very low in-degree. Some studies have used Complex Network in order to evaluate software structures. Zimmermann and Nagappan [12] found that measures from network analysis, such as centrality and closeness, can predict defects for a software system. Jenkins and Kirk [13] 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 a power law. They propose an instability metric that they claim to be conformed with the growth process of the software system. Although these studies have revealed important features of software structures and of software evolution process, the knowledge we have so far on this subject is still incipient. The research described in this paper aims to characterize the software evolution process by detailing how the degradation of the software structures occurs. For this purpose, a new approach is used.

The present work is based on a recent published work of the authors [3] in which the software evolution is characterized by means of software metrics and Complex Networks concepts. One of the main contributions of that work is the definition of Little House, a model for the macroscopic structure of object-oriented software systems. In the present work, the evolution of software systems is analyzed according to the Little House model. This model, which is detailed in Section II-A, is a "picture" of the software system. The aim of the present study is to investigate how this "picture" evolves by means of software metrics.

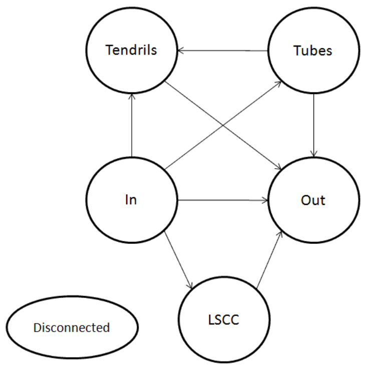


Figure 1. *Little House* – The macroscopic structure of software networks

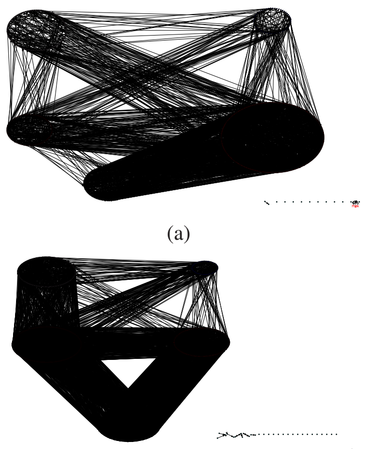


Figure 2. Hibernate modeled by Little House (a) version 3.0 and (b) version 3.5.1

A. The Little House Model

Little House [3] is a macroscopic view of software system structures. Its definition was based in the Bow-Tie model [14], which represents the way pages in Web are connected one to another. Bow-Tie provides a macroscopic view of the Web, and reveals that there is a giant group of pages that are strongly connected one to another.

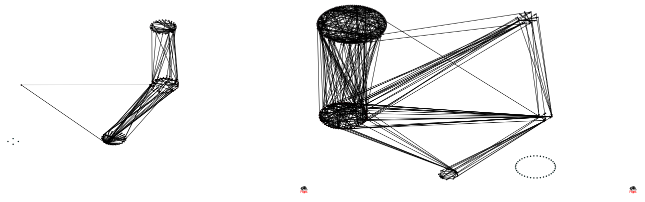
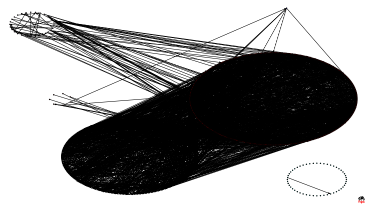
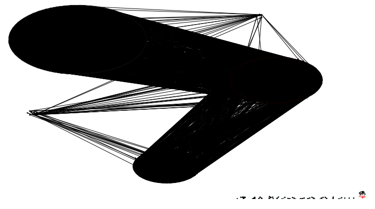


Figure 3. JUnit modeled by Little House (a) version 3.4 and (b) version 4.8.1



(a)



(b)

Figure 4. JGNash modeled by Little House (a) version 1.10.0 and (b) version 2.30.0

Little House is a graph composed by six nodes, and each of them corresponds to a distinct group of classes. The model is depicted in Figure 1. The components of Little House are described following.

- **LSCC**: is the largest strongly connected component of the software system. In this component, any class can reach all the other classes of *LSCC*. *LSCC* plays a central role in the system since its classes are strongly connected one to another, what might make this component hard to be understood, tested and maintained.
- **In**: classes from *In* can use any other class of the software system, but they are not used by classes from the other components.
- **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.
- **Tendrils**: classes from *Tendrils* use only classes from this component or from *Out*. Besides, a class from *Tendrils* can be used only by classes from *Tendrils*, *Tubes* or *In*.
- **Tubes**: classes from *Tubes* use only classes from this component, *Out* or *Tendrils*. Besides, a class from *Tubes* can be used only by classes from *Tubes* or *In*.
- **Disconnected**: a class in this component has no connection with classes of the other components.

Figures 2, 3 and 4 show the depicted figure of Little House of Hibernate, JUnit and JSNash, in their first and last versions analyzed in this work. This model is intended to improve tasks such as program analysis, program visualization, testing, refactoring and maintenance. However, to achieve this aim, it is necessary to investigate the characteristics and implications of the model. The present research is concerned on investigating how each component of Little House evolves as the system grows.

III. The Software Metrics Set

Six object-oriented software metrics were considered in this research. They are chosen because they evaluate important factors of software quality. Moreover, there are thresholds proposed for them in the literature [15]. The software metrics are described following.

- *# Afferent Couplings (AC)*: this metric is the number of incoming couplings of a class, providing an indicator of the number of classes that depend upon a given class. To compute this metric, the following types of connections between two classes were considered: inheritance, use of public fields and method call. In the case of inheritance, when a class *A* is a superclass of *B*, then there is a directed coupling from *B* to *A*, i.e., an incoming coupling in *A*.

- *Lack of Cohesion in Methods LCOM* [16]: this metric has been widely considered as a flawed way to measure class cohesion. Nevertheless, LCOM is implemented in a large number of tools [17]. Due to this, LCOM is considered in this work.

- *Cohesion of Responsibility (COR)*: this metric is defined and evaluated in a recent published work of the authors [4]. COR is given by 1/C, where C is the number of disjointed sets of methods within the class. Each set consists of methods which are similar. Two methods of a class *A* are similar when they use common fields or common methods of *A*. 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 of similar methods 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 degree.

- *Depth of Inheritance Tree (DIT)* [16]: this metric gives the maximum distance of a class from the root class in the inheritance tree of the system. Although inheritance is a powerful technique for software reuse, its immoderate use makes software design more complex [18]. Therefore, DIT may be used as an indicator of the difficulty of understanding a class.

- *# Public Fields (PF)*: is the number of public fields defined in a class. Avoiding public fields in programs is widely considered as a good practice because the use of public fields can reduce the modularity of the program [19], [20].

- *# Public Methods (MF)*: is the number of public methods defined in a class. This metric is an indicator of the interface size of a class. Large classes are

Table I
THE SOFTWARE METRICS THRESHOLDS

Level	*Metric*	*Reference Values*
System	COF	Good: up to 0.02 - Regular: 0.02 to 0.14 - Bad: greater than 0.14
Class	# Afferent couplings	Good: up to 1 - Regular: 2 to 20 - Bad: greater than 20
Class	# Public fields	Good: 0 - Regular: 1 to 10 - Bad: greater than 10
Class	# Public methods	Good: 0 to 10 - Regular: 11 to 40 - Bad: greater than 40
Class	DIT	Typical value: 2
Class	LCOM	Good : 0 - Regular: 1 to 20 - Bad: greater than 20
Class	COR	Good: 1 - Regular: 0,2 to 0,5 - Bad: less than 0,2

considered as a code smell because they are harder to understand and maintain [19].

A. The Software Metrics Thresholds

Ferreira et al. [15] have proposed and evaluated thresholds for object-oriented software metrics, defining three ranges of reference values for the metrics: *good*, which refers to the most common values of the metric; *regular*, which is an intermediate range of values with a low but not irrelevant frequency; and *bad*, that refers to values with quite rare occurrences. These thresholds were derived empirically, having as basis the metric values of 26,000 classes from open-source programs. The purpose of these thresholds is to establish a benchmark for the evaluation of software systems based on what is done in practice.

In the present research, the metric thresholds are applied to evaluate classes within a given component of Little House. Table I shows the thresholds of the software metrics considered in the present research.

IV. METHODS

Data from the first and the last versions of 13 open-source software systems developed in Java were analyzed in this study. The last version considered for each software is up to 2009/2010. Table II shows the data of the set of programs. They are developed in Java, they have at least 5 versions or releases, and they are 4 years old at least. Another criterion of the sample selection was the availability of the bytecodes of the programs because the tool used to perform the measurements has as input the bytecodes.

The measures and the graph of the software systems were gathered by a tool called Connecta [21], which generates graphs representing the software networks and exports them as a file in an appropriate input format for Pajek [22], a network analysis tool. Using Pajek, the software network was fitted to the Bow-Tie model [14] and, then, the result of this fitting was processed in order to be adapted to Little House.

After identifying the classes for each component of Little House, the following steps were performed:

Table II
SOFTWARE SYSTEMS ANALYZED IN THE STUDY

Name	*Category*	*downloads/week*	*Age*	*#classes*	*#versions*
DBUnit	Database	448	2002 - 2009	198 - 369	25
FreeCol	Game	7,452	2003 - 2010	112 - 5902	27
Hibernate	Database	12,906	2004 - 2010	956 - 2446	53
JasperReports	Development	5,542	2001 - 2010	525 - 5304	50
Java Groups	Cooperation	465	2003 - 2009	696 - 1137	40
JGNash	Financial	822	2002 - 2010	782 - 3603	40
Java msn library	Communication	271	2004 - 2010	494 - 872	10
Jsch	Security	2,304	2004 - 2009	202 - 271	29
JUnit	Development	1,834	2000 - 2009	78 - 230	18
Logisim	Education	1,590	2005 - 2009	908 - 1185	28
MeD's Movie Manager	Storage	1,169	2003 - 2010	64 - 517	60
Phex	Network	1,084	2001 - 2009	393 - 1352	26
Squirrel sql	Database	7,270	2006 - 2010	424 - 1223	26

- for each metric, the percentile of classes in the component whose measures fall in the *good* range of the metric threshold was computed;
- the first and the last version was compared in order to compute the difference between the percentiles. When the result is positive, it means that in the last version, the percentile of classes having *good* measures is larger then in the first version, i.e., the proportion of such classes increases within the component as the system evolves. On the other hand, when the result is negative, it means that the component in the last version has a lower percentile of classes with *good* measures, indicating that the component has degraded over the evolution of the system.

The purpose of such analysis is to observe how the Little House components evolve. It is expected to identify if there is a single component or a group of components that suffer a higher level of degradation comparing with the other ones.

V. RESULTS

To assess the evolution of a Little House component in a program, we computed the difference of percentiles of *good* classes between the last and the first versions in the component, for each software metric. Tables III-VIII show the resulting data gathered in the experiment. For example, as shown in Table III, in the case of AC metric this difference in LSCC of DBUnit is -39,58. This means that the ratio of the *good* classes in LSCC of DBUnit decreased as the program increased. Empty entries in the tables are used to indicate that the first version of the program does not have the corresponding component. For instance, the first version of Freecol does not have *Tubes*.

For each component and for each software metric, we have computed the median, the mean, the standard deviation and the 90% confidence interval of the mean for the sample. The data are shown in Tables III-VIII. The results indicate that the Little House components evolve according to the following patterns.

- Disconnected: in general, there is a very few variation in the quality of classes in this component as the system evolves. The use of inheritance is an aspect in which a degradation was observed because the rate of classes evaluated as *good* decreased by 1.5 in the final version. In 62% of the programs from the sample, the rate of classes with DIT up 2, that is the typical values of this metric, decreased. The rate of classes with *good* values of PM decreased by 1.8.

- In: the proportion of classes evaluated as *good* by all metrics increased with the growth of the system. A possible explanation for this behavior is that as the classes from *In* are not used by any other component of Little House, they might have a high level of stability.

- Tubes: the rate of classes with *good* values of COR decreased by 7.2, while the rate of classes evaluated as *good* by the other metrics increased. A peculiarity about *Tubes* is that the first versions of some programs do not have this component, but the latest versions have it.

- Tendrils: the rate of classes with *good* values of AC decreased by 3, while the rate of classes evaluated as *good* by the other metrics increased. *Tendril* is the component that can work as a bridge between *In* and *Out* and also between *Tubes* and *Out*. Perhaps for this reason, the number of AC of its classes tends to increases.

- LSCC: according to LCOM and COR, the ratio of classes with *good* cohesion in LSCC increased over the system growth. The ratio of classes with DIT up to 2 increased by 3.5. That is, in terms of the aspects evaluated by DIT, LCOM and COR, the evolution of LSCC leads to an increase on the proportion of classes evaluated as *good* in this component. However, the opposite occurs with the level of connectivity of the component. The proportion of classes with *good* values of AC decreased by 11, indicating that the role of the classes from LSCC becomes even more critical as the system evolves. The rate of classes with *good* values of PF and PM decreased by 5.5 and 5.2, respectively. Hence, classes from *LSCC* tends to have even more public methods and public fields as the system grows. This enlargement in size of the interfaces of classes can be a direct cause of increasing the connectivity degree of *LSCC*.

- Out: this is the only component of Little House in which the rate of class with *good* values of COR and LCOM decreased, by 6.2 and 10, respectively. This result shows that, in general, the internal cohesion of classes from *Out* degrades. The rate of classes with *good* values of PF decreases in 7.7. Classes from *Out* are used by other components, but do not use classes from other components. Hence, a reasonable cause for this result is that new services and public fields are grafted in these classes, so that their internal cohesion decreases.

Table III
THE EVOLUTION OF LITTLE HOUSE COMPONENTS ACCORDING TO AC (#AFFERENT COUPLINGS)

Program	***Disc.***	***LSCC***	***In***	***Out***	***Tubes***	***Tendr.***
DBUnit	0,00	-39,58	0,67	19,37	35,71	12,50
FreeCol	0,00	1,21	-23,08	9,64	-	-8,33
Hibernate	-0,38	-3,57	7,82	-6,77	-3,93	-15,19
Jasper	0,00	32,67	10,00	10,34	14,81	-8,74
JGroups	4,63	-73,50	74,72	-13,49	51,85	-8,29
JGNash	-0,86	-15,62	-20,83	-10,78	-69,52	-12,13
Jml	0,00	-19,61	21,34	-10,10	2,78	-17,05
Jsch	-1,89	-12,50	6,48	11,67	50,00	31,43
Junit	0,00	-13,13	-29,09	-36,54	-	-23,47
Logisim	0,26	-2,38	-6,04	1,65	-	-4,23
Med's	-	-5,33	-6,67	22,83	-	50,00
Phex	0,00	6,12	5,33	13,49	90,00	-14,55
Squirrel	2,04	1,94	-19,51	-9,05	-4,17	-22,00
Median	0,00	-5,33	0,67	1,65	14,81	-8,74
S	1,62	25,01	26,63	16,53	45,47	21,73
Mean	**0,32**	**-11,02**	**1,63**	**0,17**	**18,62**	**-3,08**
CI (90%)	-0,91	-42,99	-17,62	-8,42	-33,37	-13,82
	1,14	4,45	15,02	7,02	42,27	6,74

These findings show empirically that *LSCC* and *Out* are the two main critical components in Little House. Although the sample was not large, the results expose a tendency of the pattern formation of the software structures throughout its evolution. In this study, we investigated how the components of Little House evolve by means of software metrics. Although this work is not concerned on determining the causes of the observed phenomenon, this is a relevant question to be further studied. For instance, it is important to know why and how the internal cohesion of classes from *Out* declines over the system growth. Determining these causes may make Little House a powerful approach of software evaluation.

VI. CONCLUSION

Understanding software system structures is essential to achieve effective software maintenance. However this issue is extremely difficult to tackle due to the high complexity of software systems. Aiming to improve the understanding and the maintenance of software, the Little House model was defined in a recently published work of the authors. Little House is a model for the macroscopic topology of structures of object-oriented software systems. According to this model, a software system can be depicted as a

Table IV
THE EVOLUTION OF LITTLE HOUSE COMPONENTS ACCORDING TO DIT

Program	***Disc.***	***LSCC***	***In***	***Out***	***Tubes***	***Tendr.***
DBUnit	-0,65	0,00	0,00	36,01	0,00	2,08
FreeCol	-7,14	31,03	38,46	19,17	-100,00	-70,83
Hibernate	-0,38	0,00	0,00	-0,83	0,00	-1,28
Jasper	-0,37	0,00	0,00	0,16	0,00	2,73
JGroups	-1,16	3,42	96,50	-26,19	100,00	-3,62
JGNash	-6,76	-2,20	0,00	5,39	93,33	-2,93
Jml	0,00	0,00	14,29	1,58	0,00	0,00
Jsch	0,00	0,00	0,00	5,00	0,00	5,71
Junit	0,00	4,55	-1,82	-5,77	-	89,47
Logisim	0,80	-2,39	5,65	-1,90	-	4,29
Med's	-	6,89	-20,00	-8,87	-	-28,57
Phex	-2,74	-4,80	-8,58	5,12	90,00	-7,27
Squirrel	-0,08	8,78	-2,44	-0,63	20,83	38,00
Median	-0,38	0,00	0,00	0,16	0,00	0,00
S	2,67	9,11	29,39	14,40	60,62	35,78
Mean	**-1,54**	**3,48**	**9,39**	**2,17**	**20,42**	**2,14**
CI (90%)	-5,13	-3,86	-6,16	-9,61	-39,70	-33,25
	0,07	7,79	23,47	5,30	53,43	20,01

Table V
THE EVOLUTION OF LITTLE HOUSE COMPONENTS ACCORDING TO LCOM

Program	***Disc.***	***LSCC***	***In***	***Out***	***Tubes***	***Tendr.***
DBUnit	10,61	33,33	10,38	-39,92	-42,86	-16,05
FreeCol	-5,36	11,58	46,15	3,32	-	8,33
Hibernate	-1,27	7,24	10,13	15,72	-4,26	11,84
Jasper	5,13	3,33	32,50	-10,10	-12,50	-8,97
JGroups	6,20	16,24	49,60	4,76	85,19	11,22
JGNash	-6,43	-3,57	-8,93	-14,86	26,67	0,34
Jml	13,93	0,22	6,11	9,94	-11,11	23,85
Jsch	-20,49	-6,25	-5,37	-1,67	50,00	-31,43
Junit	0,00	-13,13	-61,82	-21,15	-	-17,51
Logisim	3,92	-7,80	-5,26	-3,46	-	-1,53
Med's	-	23,56	23,33	-41,89	-	71,43
Phex	5,13	5,82	28,64	4,79	20,00	20,00
Squirrel	0,00	33,64	56,10	14,25	-41,67	19,00
Median	1,96	5,82	10,38	-1,67	-4,26	8,33
S	9,01	15,12	31,40	18,80	42,13	25,45
Mean	**0,95**	**8,02**	**13,97**	**-6,17**	**7,72**	**6,96**
CI (90%)	-3,55	2,09	6,47	-21,09	-29,93	-7,28
	4,82	12,33	28,66	5,30	34,16	20,44

Table VI
THE EVOLUTION OF LITTLE HOUSE COMPONENTS ACCORDING TO COR

Program	***Disc.***	***LSCC***	***In***	***Out***	***Tubes***	***Tendr.***
DBUnit	-0,06	25,00	-10,60	-35,23	-68,57	-28,92
FreeCol	-5,36	27,10	30,77	18,95	-	8,33
Hibernate	-1,46	4,71	10,97	17,08	-9,79	14,02
Jasper	4,76	-15,00	30,00	-36,09	-20,22	-8,09
JGroups	2,73	-2,56	46,58	-0,79	44,44	10,98
JGNash	-4,07	-6,08	-4,76	-12,61	-66,67	7,06
Jml	13,93	9,80	-3,44	18,71	19,44	16,37
Jsch	-38,54	12,50	-5,37	6,67	0,00	2,86
Junit	75,00	73,23	23,03	50,00	-	44,74
Logisim	4,63	-7,08	-15,46	-2,55	-	-2,70
Med's	-	26,44	30,00	-49,06	-	71,43
Phex	-0,82	5,74	28,64	0,03	20,00	18,18
Squirrel	-0,15	22,87	56,10	11,80	16,67	27,00
Median	-0,11	9,80	23,03	0,03	0,00	10,98
S	25,57	22,67	23,00	27,06	39,08	24,59
Mean	**4,22**	**13,59**	**16,65**	**-1,01**	**-7,19**	**13,94**
CI (90%)	-3,43	-6,93	0,87	-25,81	-62,95	-11,33
	17,35	22,81	28,50	14,31	20,48	26,33

Table VII
THE EVOLUTION OF LITTLE HOUSE COMPONENTS ACCORDING TO PF (# PUBLIC FIELDS)

Program	***Disc.***	***LSCC***	***In***	***Out***	***Tubes***	***Tendr.***
DBUnit	0,59	2,08	-3,94	4,70	-14,29	0,98
FreeCol	-12,50	21,13	-7,69	-9,95	-	0,00
Hibernate	2,61	-4,22	-7,30	-3,34	-14,23	-6,54
Jasper	15,27	-1,33	-10,00	-24,05	-29,63	-20,94
JGroups	3,30	-53,85	92,37	5,56	66,67	5,82
JGNash	-1,79	-5,18	-4,17	-7,86	-22,86	-9,75
Jml	4,64	-22,22	10,67	-0,99	27,78	-8,70
Jsch	1,28	-6,25	2,59	0,00	50,00	5,71
Junit	-2,63	0,00	0,00	-21,15	-	96,49
Logisim	2,33	5,42	3,61	-0,41	-	0,21
Med's	-	-8,00	-6,67	-19,81	-	0,00
Phex	-7,07	4,05	-5,88	-11,73	0,00	-9,09
Squirrel	32,96	-2,59	-2,44	-10,94	0,00	-14,00
Median	1,81	-2,59	-3,94	-7,86	0,00	0,00
S	11,50	17,46	26,94	9,72	33,68	29,13
Mean	**3,25**	**-5,46**	**4,70**	**-7,69**	**7,05**	**3,09**
CI (90%)	-5,32	-25,04	-19,57	-13,79	-36,95	-12,03
	9,10	3,64	17,96	-2,97	28,48	17,56

graph with five components, that are also graphs representing groups of classes.

This work carried out an experimental study in order to expose characteristics of software evolution by means of Little House and software metrics. Six software metrics were analyzed in this work, namely: number of afferent couplings, LCOM, COR, DIT, number of public fields and number of public methods. In particular, the study evaluated how the rates of classes having *good* measures into each Little House component evolve over the system growth.

The analysis of the results suggests that there are two components of Little House that are more critical, not only due to the way they evolve, but also due to their central role in the system: *LSCC* and *Out*. *LSCC* is the strongly connected component of the system. As the system grows, the ratio of classes with high values of AC in *LSCC* tends to increase. This might make modifications and error in classes from *LSCC* more impacting on the system as it evolves. All the components of Little House depend upon *Out*. This feature makes *Out* a fundamental component in the system. The results show that the system evolution causes a slight, but not negligible, degradation of the internal quality of classes from *Out*.

Further works are needed in order to achieve an even more detailed knowledge about Little House. For example, it is important to verify if there is any correlation between the components of Little House and error proneness or change propagation. Another important issue to be investigated is the qualitative characteristics of classes within each component of Little House. We envision that answering questions

Table VIII
THE EVOLUTION OF LITTLE HOUSE COMPONENTS ACCORDING TO PM (# PUBLIC METHODS)

Program	***Disc.***	***LSCC***	***In***	***Out***	***Tubes***	***Tendr.***
DBUnit	-0,65	-2,08	-3,05	-8,22	-8,57	97,01
FreeCol	7,14	6,02	-11,54	33,47	-	0,00
Hibernate	8,78	0,26	11,78	11,14	2,13	7,68
Jasper	1,38	5,33	65,00	-4,61	22,38	-0,15
JGroups	-10,00	-46,15	75,20	12,70	100,00	9,52
JGNash	-1,12	-8,58	-12,50	-6,92	51,43	-12,95
Jml	-12,38	-10,46	-2,50	-2,48	-5,56	-4,01
Jsch	0,00	-6,25	-1,11	1,67	50,00	0,00
Junit	0,00	-6,57	-3,64	-5,77	-	92,11
Logisim	-0,64	-1,80	-10,20	0,28	-	-1,32
Med's	-	-7,78	0,00	-10,38	-	0,00
Phex	-6,16	10,93	-9,90	8,96	90,00	-5,45
Squirrel	-8,16	0,16	45,12	-6,20	-12,50	-6,00
Median	-0,64	-2,08	-2,50	-2,48	22,38	0,00
S	6,40	13,85	30,29	12,14	42,88	36,40
Mean	**-1,82**	**-5,15**	**10,97**	**1,82**	**32,15**	**13,57**
CI (90%)	-3,98	-21,84	-7,77	-5,50	-2,84	-12,29
	1,76	2,43	25,40	8,07	56,24	20,47

like those will lead to a more deeper comprehension about software structures, what may make Little House a robust approach to understanding, maintenance, testing and visualization of software systems.

REFERENCES

[1] 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.

[2] M. E. J. Newman, "The structure and function of complex networks," in *SIAM Reviews*, vol. 45, no. 2, 2003, pp. 167–256.

[3] K. A. M. Ferreira, M. A. Bigonha, R. S. Bigonha, and B. M. Gomes, "Software evolution characterization - a complex network approach," in *X Brazilian Simposium on Software Quality - SBQS'2011*, Curitiba, Paraná, Brazil, 2011.

[4] K. A. M. Ferreira, M. A. Bigonha, R. S. Bigonha, H. C. Almeida, and R. C. N. Moreira, "Métrica de coesão de responsabilidade - a utilidade de métricas de coesão na identificação de classes com problemas estruturais," in *X Brazilian Simposium on Software Quality - SBQS'2011*, Curitiba, Paraná, Brazil, 2011.

[5] S. Koch, "Software evolution in open source projects—a large-scale investigation," *J. Softw. Maint. Evol.*, vol. 19, no. 6, pp. 361–382, 2007.

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

[7] A. Israeli and D. G. Feitelson, "The linux kernel as a case study in software evolution," *The Journal of Systems and Software*, vol. 83, no. 3, pp. 485–501, 2010.

[8] 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.

[9] R. Wheelson and S. Counsell, "Power law distributions in class relationships," in *Proceedings of 3rd International Workshop on Source Code Analysis and Manipulation (SCAM)*, Sept. 2003.

[10] 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, Oct. 2006.

[11] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," vol. 18, no. 1, Sept. 2008.

[12] 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*. New York, NY, USA: ACM, 2008, pp. 531–540.

[13] S. Jenkins and S. R. Kirk, "Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution," *Inf. Sci.*, vol. 177, no. 12, pp. 2587–2601, 2007.

[14] 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*, 2000, pp. 309–320.

[15] K. A. M. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, pp. 476–493, 1994.

[17] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2008, pp. 131–142.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999.

[20] B. Meyer, *Object-oriented software construction*, 2nd ed. USA: Prentice Hall International Series, 1997.

[21] K. A. M. Ferreira, *Avaliação de Conectividade em Sistemas Orientados por Objetos*, Master Thesis - Federal University of Minas Gerais. Belo Horizonte, Brazil, June 2006.

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