A Generic Macroscopic Topology of Software Networks - a Quantitative Evaluation

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

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

***Abstract*—The dependence among modules in a software system usually is represented as a network, in which the nodes are the modules, and the edges are the connections between the modules. The relationships among modules in software systems are hard to assess, especially in large programs. Knowing the nature of the software system structures is very important to improve maintenance tasks and other challenging tasks in software development. A previous work of the authors of this paper has defined a model to the topology of software networks, named Little House. This model is a generic macroscopic view of software systems, and it is an adaptation of the well-known Bow-tie model. According to Little House, a software network can be partitioned into six components, in such a way there is a special pattern of connections among them. This paper describes the results of a quantitative evaluation of Little House. The aim of this work is to investigate whether the components of Little House can be described by any pattern of software metric values. The results of this evaluation indicate that in the software systems developed currently there are two main components of Little House that have critical values of metrics. This finding suggests that classes from those components should be carefully considered when maintenance tasks are performed in the program.**

***Keywords*-software metrics; complex networks; software topology; software visualization.**

I. Introduction

The Software Engineering community have defined principles and processes to be adopted in order to develop high quality software systems. However, we know too little about the real nature of the software systems we have to deal with. Knowing the way the software systems are really constructed is essential for challenging tasks in software development, such as maintenance, testing and refactoring. The lack of this knowledge is detrimental to such tasks, because a substantial portion of them depends upon understanding software structure [1].

A software system can be viewed as a network in which the nodes correspond to the modules of the system, and an edge corresponds to a relationship between two of those modules. In an object-oriented software system, the classes can be taken as the modules. In this case, if a class A uses a class B, then there is an edge from A to B. Following this idea, some researches have been carried out in order to identify the nature of networks constituted by the modules within the software systems. The main findings of those works indicate that the in-degree distribution in software system networks follows a power-law [2], [3], [4]. However, the knowledge about the real structural pattern of software systems is still incipient.

Aiming to overcome this problem, a model called Little House has been defined in a previous published work of the authors [5]. Little House is a generic model of software system networks. According to Little House, the network constituted by the modules in a software system has six components, which are connected one to another by following a specific pattern. Little House is intended to be used to improve tasks such as maintenance and refactoring. Nevertheless, to achieve this aim, the model needs to be deeply investigated in order to identify the characteristics of its components and to determine its implications. For this purpose, the research presented in this work carried out an empirical characterization of Little House by means of software metrics. The software metrics considered in this analysis evaluate size, information hiding, class cohesion and inheritance, because these are important software design properties related to software maintenance. The aim of this work is to investigate whether there are components of Little House which concentrate classes with critical values of software metrics. The results of this work show that there are two main critical components in Little House. Although this work does not exhaust all the necessary analysis on the implications of Little House, their results indicate that the model may be used as a robust tool to assist in understanding and maintaining programs, for instance, by aiding to identify critical parts of the program graphically.

The remaining of this paper is organized as follows. Section II discusses the related work. Section III describes the Little House model. Section IV describes the method used to perform the data collection and analysis. Section V shows and discusses the results of the study. Section VI discusses the limitations of the work. Section VII brings the conclusions and indications of future work.

II. RELATED WORK

Software structure visualization is a way to aid understanding programs [6]. In order to achieve a mental model of software systems, some metaphors have been used to represent programs. For instance, Wettel and Lanza [7] have defined a 3D approach for software visualization by representing a software system as a city. The aim of their approach is to show a "beautiful picture" of the system and represent it as a city in which the classes are the buildings and the packages are the blocks. However, directed graphs seems to be the most common approach to represent the relationship among software entities. In an object-oriented software system, these fundamental entities may be the classes. Using a directed graph to represent an object-oriented program, classes will be the nodes, and a relationship between two classes will be an edge. For example, if a class *A* depends upon a class *B*, hence there will be an edge from *A* to *B*. As pointed by Ball and Eick [8], graph layout is one of the main problems in software visualization because nodes and edges must be showed in such way the graph can clearly show the system structure. This problem is particularly difficult and important in the case of large systems. The present work is related to the topic of software visualization because it evaluates Little House[5], which provides a visual representation of software networks. Little House uses graphs to represent such structures in a macro level, what may be useful to cope with large systems. Little House is described in details in Section III.

More than just a tool for software visualization, Little House is a model that may help understanding the real structures of software systems. Some initiatives for investigating the real structures of software systems have occurred in the recent years. For instance, the concepts of Complex Networks [9] have been applied to characterize software structures. Many researches have confirmed that software structures have characteristics of scale-free networks [2], [3], [4]. In a scale-free network, there is a large number of nodes with a low degree, and a very small portion of nodes with a high degree. Wen et al. [10] have identified characteristics of scale-free networks in Java programs, and Yao et al. [11] have found that the scale-free properties become more evident as the software system grows.

Other approaches to characterize software structures have been also applied. Baxter et al. [3] have characterized the shape of Java programs by means of software metrics. They have found that most software metrics follow a power-law. Lindsay et al. [12] have carried out an empirical analysis in order to identify the characteristics of large classes in object-oriented programs. They have concluded that most large classes have poor design.

Those works expose important characteristics of software systems. However, this is not enough to provide a solid knowledge about the nature of the programs we have to maintain. The present work is based in Little House model, and aims to characterize the components of the model by means of software metrics in order to advance the knowledge about real software structures.

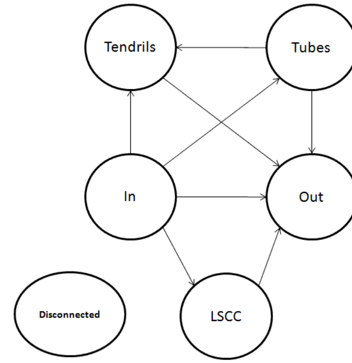


Figure 1. *Little House* – The generic macroscopic topology of software networks

III. THE LITTLE HOUSE MODEL

Little House is a model for the macroscopic topology of object-oriented software systems [5]. It was defined having as basis the well-known Bow-tie model [13], which represents the Web graph. Little House is an adaptation of the Bow-tie picture, in order to make clearer the relationships among the components of the model. The Little House model is a graph in which a node corresponds to a specific group of classes. In each of these components, classes are freely connected one to another. The model is depicted in Figure 1. Figure 2 shows the software networks of Hibernate and JUnit modeled by Little House. The components of Little House are the following:

- **In**: classes from *In* can use classes from other components. However, they are not used by classes that are outside of *In*. So, classes from *In* can demand services from classes of other components, but they only provide services to classes inside *In*. *In* is, then, the *user* component of the system.

- **LSCC (Largest Strongly Connected Component)**: this component is the *core* of the 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. Classes from *LSCC* are strongly connected one to another, what might make this component hard to be understood, tested and maintained.

- **Out**: classes from *Out* can be used by any other class of the software system, but they use only classes which are inside this component. That is, *Out* is the

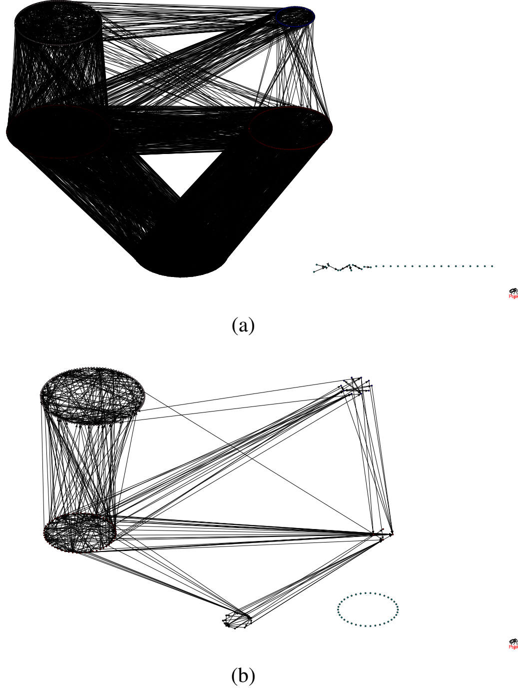


Figure 2. Software networks modeled by Little House: (a) Hibernate (version 3.5.1); (b) JUnit (version 4.8.1)

Table I
SOFTWARE SYSTEMS ANALYZED IN THE STUDY

Name	***Category***	***Age***	***#classes***	***#versions***
DBUnit	Database	2002 - 2009	198 - 369	25
FreeCol	Game	2003 - 2010	112 - 5902	27
Hibernate	Database	2004 - 2010	956 - 2446	53
Jasper Reports	Development	2001 - 2010	525 - 5304	50
Java Groups	Cooperation	2003 - 2009	696 - 1137	40
JGNash	Financial	2002 - 2010	782 - 3603	40
Java msn	Communication	2004 - 2010	494 - 872	10
Jsch	Security	2004 - 2009	202 - 271	29
JUnit	Development	2000 - 2009	78 - 230	18
Logisim	Education	2005 - 2009	908 - 1185	28
MeD's	Storage	2003 - 2010	64 - 517	60
Phex	Network	2001 - 2009	393 - 1352	26
Squirrel sql	Database	2006 - 2010	424 - 1223	26

provider component of the system.

- **Tubes**: classes from *Tubes* use only classes from this component, *Out* or *Tendril*. Besides, a class from *Tubes* can be used only by classes from *Tubes* or *In*.

- **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*.

- **Disconnected**: a class in this component has no connection with classes from other component.

Little House was defined to provide a macroscopic view of software systems, so that software engineers can use it to analyze the software systems they deal with. For instance, when a software system has a large LSCC, it might be a sign of a high complexity and difficulty of maintenance, and changing a class from *Out* might imply in a larger impact than changing a class from *In*. This model may improve tasks such as software maintenance and refactoring. However, to achieve this aim, a detailed characterization of the model is necessary. In this vein, this work presents an empirical evaluation of Little House by means of software metrics.

IV. METHODS

The research question of this work is the following: "Which components of Little House have the highest percentage of classes with potential design deviances?". In this study, we apply software metrics to investigate this question.

In this work, data from 13 open-source software systems developed in Java were analyzed. The 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 their bytecodes because the tool used to perform the measurements evaluates the compiled code, not the source code. Table I shows the data of the set of programs used in this study.

The software measurements were collected by a tool called Connecta [14]. The software metrics considered in this study are described in Section IV-A. Connecta also generates graphs which represent the software networks, and exports them as a file in an appropriate input format for Pajek [15], a network analysis tool. Pajek has as entry a file which describes the nodes and the edges of a network. Using this tool, the software network was fitted to the Bow-tie model [13] and, then, the result of this fitting was processed in order to be adapted to Little House. In the sequence, the list of classes within each component of Little House was generated.

The analysis performed in this study is related to the identification of the components with the worst measurements

Table II
THE SOFTWARE METRICS THRESHOLDS

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

for each software metric considered in this research. Two previous analyses were performed in this study. In the first one, for each software and for each component of Little House, the mean value of the metric was calculated. In the second analysis, the median of the values were considered. These analyses did not lead to any distinguished result. A possible reason for that is the property of the distribution values of the measures. Most of the metrics used in this study are modeled for a power-law distribution [3], [16]. In such distribution, there is a large amount of occurrences of low values, and there is a very few occurrences of high values. Hence, the mean value in power-law distributions is not representative. The analysis of these data, hence, needs to consider this property.

To perform the data analysis, it was used the software metric thresholds proposed by Ferreira et al. [16]. The thresholds of the software metrics are discussed in Section IV-B. For each software and for each software metric, the number of classes within each component having the metrics in the *regular* or *bad* ranges was considered. The justification for considering those ranges is the premise that when a class has a metric classified as *bad* or *regular* is a sign that the class might have design flaws, whereas classes with a given metric in the *good* range do not present relevant design deviances regarding the evaluated factor. Although this research did not aim at determining whether this premise is true, the work of Ferreira [16] describes results of experiments that indicate that it can be accepted.

The aim of the analysis is to identify, for each software system, the components of Little House with the highest relative number of classes with a given software metric in the *regular* or *bad* ranges.

A. The Software Metrics Set

In this study, it was considered software metrics that evaluate the following software design properties: size, information hiding, class cohesion and inheritance. The main criterion for the selection of the metrics is the existence of thresholds proposed for them in the literature. The metrics are described following.

- Number of public methods (PM): it is the number of public methods defined in a given class. This metric is an indicator of the interface size of a class.

- Number of public fields (PF): it is the number of public fields defined in a given class. The use of public methods is recognized as a bad design, because it could introduce strong interdependence between the classes of a software system.

- Depth of Inheritance Tree (DIT) [17]: it is the maximum distance of a class from the root of the inheritance tree. The higher the DIT of a class, the higher the number of classes involved in the definition of the class.

- Number of Afferent Couplings (AC): it is the number of classes which depend upon a given class directly. When representing an object-oriented software system as a directed graph, this metric corresponds to the in-degree of a given node. In such graph, the classes are represented by the nodes, and the connections between the classes are represented by the edges. In this work, we consider that there is a connection from a class A to a class B in the following cases: A invokes a method of B; A uses a field of B; A is a subclass of B.

- Lack of Cohesion in Methods (LCOM) [17]: this metric is a measure for the lack of internal cohesion of a class. The higher the value of LCOM, the lower the level of internal cohesion. LCOM has been widely criticized in the literature. Nevertheless, it is implemented in many tools [18]. For this reason, LCOM was analyzed in the study.

- Cohesion by Responsibility (COR) [19]: 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.

B. The Software Metric Thresholds

Ferreira et al. [16] have proposed thresholds for six object-oriented software metrics. The derivation of the thresholds

is based on the statistical analysis of the data gathered from a large sample of open source Java projects. They observed that the values of some software metrics are modeled by a power law, what implies that the mean value of the metrics are not representative. They, then, analyzed the data in order to identify three ranges of values for each of the evaluated metrics: the values that have a high frequency, the values with an intermediate frequency, and the values with a very low frequency. These ranges of values are called *good*, *regular* and *bad*, respectively. The proposed thresholds are a benchmark for software evaluation by means of software metrics, since they indicate the common practice on software design.

The thresholds proposed by Ferreira et al. [16] are shown in Table II. For instance, the thresholds for AC (*number of afferent couplings*) are: up to 1 - *good*, 2 to 20 - *regular*, and greater than 20 - *bad*. This thresholds indicate that the developer should avoid classes having AC greater than 20, because classes with this characteristics appear in software systems rarely. This does not mean that a class with AC greater than 20 has necessarily a poor design. However, this should be taken as a warning sign of design deviances.

DIT is the only metric whose values are not modeled by a power-law. Hence, the mean value of DIT is representative. The mean value of DIT is 2. As the depth in the inheritance tree can make a class more difficult to be understood, it is recommended to keep DIT up 2.

Ferreira et al. [16] have analyzed the proposed thresholds empirically. The results of their experiments have shown that the *bad* range of the metrics can be used to identify classes with design deviances, as well as the *good* range is able to indicate classes with a good design. Using the same approach, the thresholds of COR was also derived and evaluated by Ferreira et al. [19]. The resulting thresholds of COR are also shown in Table II.

In this work, these thresholds are used in order to evaluate the components of Little House. We use the term *critical* to refer to the metric values which fall in the *bad* or in the *regular* ranges. In the case of DIT, the term *critical* refers to values greater than 2.

V. Results

In this section, we describe the results of the study. We first present the data about the size of the components of Little House. Further, we describe and analyze the data about the software metrics in Little House.

A. Size of the Components

Table III shows the size of the components of Little House for each software systems analyzed in this work. These data indicate that there is not a particular component that in general concentrates most classes in a system.

Tubes is the component of Little House whose classes link classes from *In* to classes from *Out*. These results indicate that this component has a small portion of classes of the system in general. The mean value of the proportion of classes within *Tubes* is 5, with a standard deviation of 6. *FreeCol* and *Meds̀ Movie Manager* indeed do not have the *Tubes* component.

Table III
Size of the components of Little House in percentage of number of classes

Name	***Disc.***	***LSCC***	***In***	***Out***	***Tubes***	***Tendrils***
DBUnit	14	8	22	20	19	17
FreeCol	5	70	2	21	0	2
Hibernate	21	20	24	15	3	17
Jasper Reports	17	9	3	21	10	40
Java Groups	16	1	62	1	3	17
JGNash	13	9	1	18	3	56
Java msn	13	10	31	25	13	8
Jsch	45	14	23	10	2	6
JUnit	17	4	24	2	4	49
Logisim	10	26	8	32	2	22
MeD's	6	18	1	30	0	45
Phex	11	30	1	53	1	4
Squirrel	15	47	3	29	2	4
$\overline{X}$	16	20	16	21	5	22
Standard deviation	10	19	18	14	6	19

In is the component of Little House whose classes are not used by classes from other components. From the viewpoint of the macroscopic topology of the system, it is the client component in the system, because it is the only component that requires the services of other components, but does not provide any service to them. The number of classes within *In* corresponds to 16% of the system in general.

LSCC, *Out* and *Tendrils* have the highest percentage of classes of the system, in general. This result emphasizes the importance of these components, especially *LSCC* and *Out*. *Out* has classes that are used by other components but do not use classes from other components, i.e., they are essentially service providers. Therefore, classes from *Out* have a central hole in the system. *LSCC* is the component whose classes are strongly connected one to another. Hence, the larger the number of classes in *LSCC*, the higher the number of classes affected by a change in a class of *LSCC*.

B. Software Metrics in Little House

This section describes the data of the software metrics observed in the Little House components for each program from the sample. The data summarization is presented in the next section.

Data from the measures of the components are shown in Tables IV and V. The data show the percentage of classes in each component whose measures fall in the *bad* or in the *regular* range of the metric threshold. For instance, in DBUnit, 70% of the classes have the *number of afferent coupling* (AC) in the *bad* or in the *regular* range of this metric. The data are from the latest versions of the software

Table IV
PERCENTAGE OF CLASSES WITH CRITICAL SOFTWARE METRIC VALUES

Name	*Component*	*AC*	*LCOM*	*COR*	*DIT*	*PF*	*PM*
DBUnit	Disc.	0	2	9	7	4	4
	LSCC	70	66	75	0	4	8
	In	36	54	75	0	8	14
	Out	52	69	80	7	9	8
	Tubes	44	50	78	0	14	8
	Tendrils	25	51	70	0	7	4
FreeCol	Disc.	0	20	20	7	12	7
	LSCC	39	48	52	9	19	14
	In	23	53	69	11	8	11
	Out	40	57	56	6	27	2
	Tubes	-	-	-	-	-	-
	Tendrils	8	9	9	71	0	0
Hibernate	Disc.	0	3	3	0	4	10
	LSCC	51	81	80	0	9	24
	In	28	71	70	0	10	15
	Out	64	56	50	1	13	10
	Tubes	49	68	71	0	16	16
	Tendrils	34	66	58	3	18	10
Jasper Reports	Disc.	0	0	0	0	15	11
	LSCC	47	87	90	0	1	25
	In	15	67	70	0	10	2
	Out	65	68	66	2	26	25
	Tubes	52	82	82	0	30	23
	Tendrils	18	71	68	2	21	5
Java Groups	Disc.	1	22	17	3	0	10
	LSCC	85	61	69	7	54	46
	In	25	51	53	3	7	24
	Out	78	43	50	36	21	14
	Tubes	47	15	56	0	33	0
	Tendrils	34	47	48	12	6	8
JGNash	Disc.	0	16	17	6	6	4
	LSCC	28	48	43	20	0	1
	In	0	28	28	0	0	0
	Out	45	57	54	15	9	7
	Tubes	0	100	0	100	0	0
	Tendrils	24	69	68	3	7	3

Table V
PERCENTAGE OF CLASSES WITH CRITICAL SOFTWARE METRIC VALUES

Name	*Component*	*AC*	*LCOM*	*COR*	*DIT*	*PF*	*PM*
Java msn library	Disc.	0	18	18	0	5	18
	LSCC	66	70	66	0	22	22
	In	7	87	89	0	4	10
	Out	73	73	58	1	18	19
	Tubes	30	78	86	0	6	6
	Tendrils	22	69	69	0	9	9
Jsch	Disc.	2	49	53	0	9	0
	LSCC	56	62	43	0	12	18
	In	18	70	71	0	8	11
	Out	58	42	33	25	0	8
	Tubes	50	50	0	0	0	0
	Tendrils	28	71	57	14	14	0
JUnit	Disc.	0	0	0	0	3	3
	LSCC	22	22	22	0	0	11
	In	29	61	44	2	0	4
	Out	75	75	50	25	25	25
	Tubes	40	90	8	0	10	0
	Tendrils	39	53	55	10	3	7
Logisim	Disc.	2	25	23	18	2	8
	LSCC	51	67	75	15	12	13
	In	21	74	84	2	4	10
	Out	53	50	52	14	10	11
	Tubes	73	47	63	0	5	10
	Tendrils	33	69	68	14	6	20
MeD's	Disc.	0	17	10	0	10	10
	LSCC	25	26	23	9	7	7
	In	7	27	20	20	7	0
	Out	47	51	49	19	20	10
	Tubes	-	-	-	-	-	-
	Tendrils	0	28	28	28	0	0
Phex	Disc.	0	8	9	3	12	5
	LSCC	29	57	56	14	3	4
	In	6	47	47	23	6	12
	Out	55	56	56	10	25	12
	Tubes	10	80	80	10	0	10
	Tendrils	14	60	62	7	9	6
Squirrel	Disc.	0	14	14	2	11	10
	LSCC	25	39	37	6	2	5
	In	19	43	43	2	2	5
	Out	43	41	38	17	14	9
	Tubes	37	74	49	12	0	0
	Tendrils	22	56	48	12	14	6

systems analyzed in this work. Following, we discuss the results for each software system analyzed in this study:

- DBUnit: *LSCC* and *Out* are the components with the highest relative number of classes with critical values of AC. All the components, except *Disconnected*, have a high percentage of classes with critical values of LCOM and COR. *Out* is the only component which have classes with critical values of DIT. However, the percentage of classes with critical values of DIT in *Out* is very low, about 7%. None of the components has a high percentage of classes with critical values of PF or PM.

- Freecol: *LSCC* and *Out* have the highest relative number of classes with critical values of AC. All the components, except *Tendrils*, have a high percentage of classes with critical values of LCOM and COR. *Tendrils* is the only component with a high percentage of classes having critical values of DIT, about 71%. Roughly 27% das classes of *Out* have critical values of PF. In all the components, just a few percentage of classes have critical values of PM.

- Hibernate: *LSCC*, *Out* and *Tubes* have the highest relative number of classes with critical values of AC. All the components, except *Disconnected*, have a high percentage of classes with critical values of LCOM and COR. *Tendrils* and *Out* are the only components with classes having critical values of DIT, 3% and 1%, respectively. There is no component that stands out about the percentage of critical PF. *LSCC* is the component with the highest percentage of classes having critical values of PM.

- Jasper Reports: *LSCC*, *Out* and *Tubes* are the components with the highest relative number of classes with critical values of AC. All the components, except *Disconnected*, have a high percentage of classes with

critical values of LCOM and COR. *LSCC* has the highest relative number of classes having critical COR, about 90%. *Tendrils* and *Out* are the only components with classes having critical values of DIT. Nevertheless, these percentages are very low, about 2%. In this software, *Out*, *Tubes* and *Tendril* have a high percentage of classes with critical PF, 26%, 30% and 21%, respectively. *LSCC*, *Out* and *Tubes* are the components with the highest percentage of classes with critical PM, 25%, 25% and 23%, respectively.

- Java Groups: *LSCC* and *Out* have the highest relative number of classes with critical values of AC. All the components have a relevant percentage of classes with critical values of LCOM and COR. *Out* have the highest percentage of classes with critical DIT, 36%. *LSCC* and *Tubes* exhibit the highest percentages of classes with critical PF, 54% and 33%, respectively. *LSCC* also has the highest percentage of classes with critical PM, 46%.

- JGNash: *Out* has the highest proportion of classes having critical AC. In this program, LCOM and COR exhibit a very disparate result: 100% of the classes from *Tubes* are critical according to LCOM, whereas none class of this component is critical according COR. An explanation for this particular result is that *Tubes* is a very small component in this program. Regarding the small size of *Tubes*, *LSCC* is the component with the highest percentage of classes having critical DIT, about 20%. None of the components present a high proportion of classes with critical values PF or PM.

- Java Msn Library: *LSCC* and *Out* have the highest relative number of classes with critical values of AC. All the components, except *Disconnected*, have a high proportion of classes with critical LCOM and COR. Only *Out* has classes with critical values of DIT, but in a very small proportion. *LSCC* and *Out* also have the highest relative number of classes with critical values of PF, 22% and 18%, respectively. *LSCC*, *Out* and *Disconnected* have the highest relative number of classes with critical values of PF, 22%, 19% and 18%, respectively.

- Jsch: *LSCC*, *Out* and *Tubes* are the components with the highest relative number of classes with critical values of AC. However, *Tubes* is very small in this program. *Out* is the component with the highest proportion of classes having critical DIT, 25%. *LSCC* and *Tendrils* are the components with the highest percentages of classes having critical PF, whereas *LSCC* and *In* have the highest proportion of classes with critical PF.

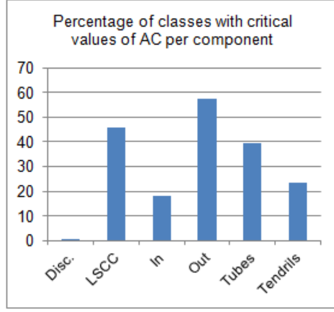
- JUnit: *Out* is outstandingly the component with the highest proportion of classes with critical AC, 75%. All the components have significant proportions of classes with critical values of LCOM and COR. However, *LSCC* is the component with the lower proportion of classes with critical LCOM and COR, 20%. In *Tubes*, the results of LCOM and COR did not show concordance. Similar to JGNash and Jsch, JUnit also has a very small *Tubes*. *Out* is the component with the highest proportion of classes having critical DIT, PM and PF.

- Logisim: *Tubes* has the highest proportion of classes with critical AC. However, *Tubes* is a very small component in this software system. Thus, *LSCC* and *Out* are the most relevant components regarding the proportion of classes with critical AC in this software system. All the components have substantial proportion of classes with critical LCOM and COR. With the exception of *In* and *Tubes*, which are very small components in this software system, all the other components have roughly equal proportions of classes with critical DIT. *LSCC* is the component with the highest proportion of classes having critical PF, whereas *Tendril* has the highest proportion of classes with critical PM.

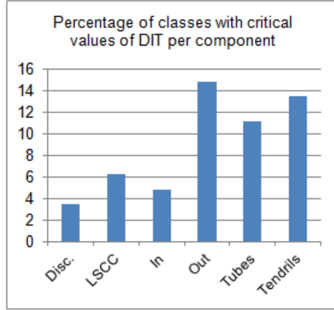
- Med's Movie Manager: *LSCC* and *Out* have the highest proportion of classes with critical AC. *Out* is the component with the highest proportion of classes with critical LCOM and COR, about 50%. *Tendrils* has the highest proportion of classes with critical DIT, 28%. *Out* is the component with the highest proportion of classes having critical values of PF and PM.

- Phex: *Out* has the highest proportion of classes with critical values of AC. All the components, except *Disconnected*, have significant proportion of classes with critical LCOM and COR. *In* and *LSCC* have the highest proportion of classes with critical DIT. However, *In* is a very small component in this software system. *Out* has the highest proportion of classes with critical PM and PF.

- Squirrel: *Out* has the highest proportion of classes with critical values of AC. Although *Tubes* has also a high proportion of classes with critical AC, it is a very small component of Squirrel. All the components, except *Disconnected*, have significant proportion of classes with critical LCOM and COR. *Out* and *Tendrils* are also the components with the highest proportion of classes having critical values of DIT, PF and PM. However, *Tendrils* is a very small component in this

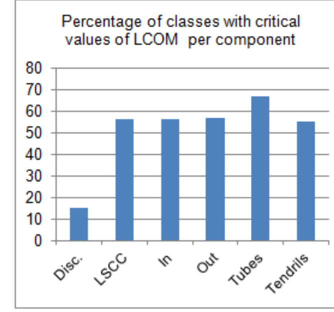


(a)

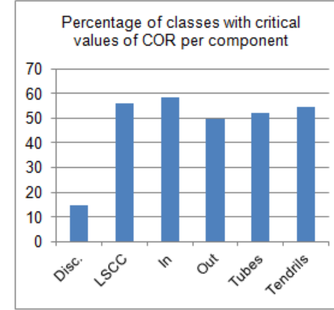


(b)

Figure 3. Mean percentage of classes with critical values of AF and DIT.



(a)



(b)

Figure 4. Mean percentage of classes with critical values of LCOM and COR.

program.

These results indicate that, although the programs analyzed in this work have different types and sizes, the distributions of their classes among the components of Little House are very similar, regarding the critical values of the software metrics.

C. Data Summarization

Figures 3, 4 and 5 summarize the data, showing the mean percentage of classes with critical values of the metrics per component.

The results of this study show that, in the case of the metric *afferent couplings*, *Out* and *LSCC* have the highest percentage of classes with *bad* or *regular* score. However, *Out* is the component with the highest percentage of classes in this situation. *Tubes* also has a high percentage of classes with critical AC.

There is no component of Little House that outstandingly has a higher percentage of classes with LCOM or COR in the *bad* or *regular* ranges when comparing with the other components. Moreover, most of the software systems analyzed in this work have a high proportion of classes with critical LCOM and COR in all their components. This characteristic might be due the evolution of the programs. The results of a previous research of the authors have shown that as a software system evolves, the internal quality of its classes degrades [19]. A curiosity on this result is that despite LCOM and COR are different metrics, they had the same behavior in this study.

Out and *Tendrils* are the components with the highest percentage of classes having DIT greater than the mean value of this metric. *Tubes* also has a high percentagem of classes with critical DIT. This result shows that, in general, classes from those components have a higher DIT value when compared to classes of the other components. Hence, *Out* and *Tendrils* are characterized by the use of inheritance, and *Out* is characterized by having deeper inheritance trees.

Out is the component with the highest percentage of classes having *number of public fields* in the *bad* or *regular* ranges. *LSCC* and *Tubes* have the second highest percentages of classes in this situation. *LSCC* is the component with the highest percentage of classes having *number of public methods* in the *bad* or *regular* ranges. *Out* has the second highest percentage of classes in this situation. This result indicates that classes from *LSCC* and *Out* are characterized by a large interface size, since they tend to provide a large number of services. This might be a reason for the inherent characteristics of these components: *Out* is the *server* component, and *LSCC* is the largest strongly connected component.

The main conclusion drawn from these results is that, in the practice, the software systems are structured in such a way their classes are distributed into six components, according to Little House. In most cases, two of those components,

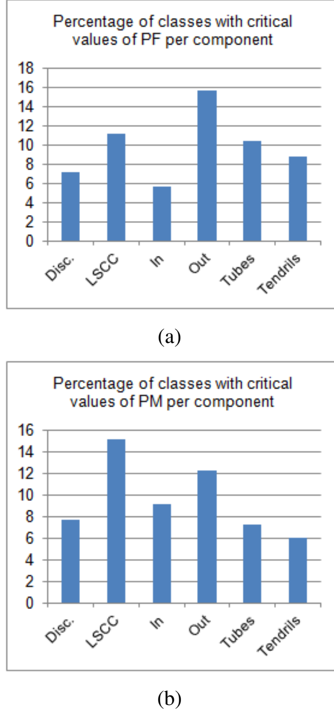


Figure 5. Mean percentage of classes with critical values of PF and PM.

Out and *LSCC*, have the highest percentage of classes with critical values of the software metrics. Considering that those metrics are able to assess the internal properties of the classes, this result means that those components are characterized by classes with potential design deviances. A reasonable implication of this result for the practice is that classes from *Out* and *LSCC* should be carefully considered in tasks such as refactoring and maintenance. Applying Little House might be useful to identify those classes graphically, and to explore the way they are connected to other classes in the system.

VI. Limitations

The analysis performed in this work is based in thresholds of software metrics. The evaluation of the thresholds are beyond the scope of this paper. Ferreira et al. [16] describe the results of an evaluation of the used thresholds. Nevertheless, those thresholds are a recent proposal and ideally they should be evaluated by the Software Engineering community, especially in real scenarios of Software Engineering.

To verify which components of Little House concentrate critical classes, we considered the *regular* and *bad* ranges of the metrics. That is, we assumed that if a class is not evaluated as *good*, it should be carefully considered when tasks such as maintenance are performed in the system. Although this assumption has not been empirically validated, it is an reasonable premise, since the *good* ranges of the metrics correspond to well known design principles.

The sample of programs analyzed in this work are from different application domain and of varying size. In spite of those differences among them, their results are very similar. Although the sample is relevant, it is not possible to ensure that the observed results can be generalized to other software systems.

VII. Conclusion

Little House is a model of the macroscopic topology of structures of object-oriented software systems. According to Little House, a software system can be modeled as a graph with six nodes or components. Little House is a practical view of how classes within a software system are connected one to another. This work carried out an empirical study in order to detail characteristics of Little House. In particular, this work aimed to characterize the components of Little House by means of software metrics.

The results of the present research showed that Little House have two critical components: *Out* and *LSCC*. *Out* is the component of Little House whose classes are used by classes from any component of Little House, but only uses classes from *Out*. *LSCC* is the largest strongly connected component of Little House. Hence, changes in classes from these components may have a large impact in the system as hole. *Out* and *LSCC* have the highest percentage of classes with software metric values considered as *bad* or *regular*. The main implication of this finding is that classes from these components should be carefully considered in tasks such as refactoring and maintenance.

This work and the previous one in which Little House was defined are a novel research on defining and characterizing a generic topology of software structures. In a recent work, we have investigated how software structures evolves by means of Little House [20]. The results of that work indicate that, as a software system evolves, *LSCC* and *Out* are the components that suffer more degradation. Nevertheless, future works are needed to provide a deeper characterization of Little House:

- A relevant question raised from this research is the correlation of Little House and error proneness. An empirical study needs to be carried out in order to determine if there is any component of Little House that has more error occurrences over the time.
- Another important issue is concerned with change propagation. This study revealed that *Out* and *LSCC* are the most critical components in terms of software metric values. A hypothesis drawn from this finding is that changes in classes from *Out* and *LSCC* may have a high impact in the system. However, it is necessary to verify this hypothesis empirically.
- It is also important to verify whether there will be any pattern of qualitative characteristic of the components of Little House. One of the questions to be answered in

this aspect is the relationship among Little House and patterns such as MVC.

- Moreover, the findings of the present work may be used for developing new approaches of software maintenance and refactoring.
- New approaches of software visualization may also be defined based on Little House.

References

[1] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Commun. ACM*, vol. 54, pp. 142–151, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1924421.1924451

[2] 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.

[3] 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.

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

[5] 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, pp. 41–55.

[6] Z. Sharafi, "A systematic analysis of software architecture visualization techniques," in *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ser. ICPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 254–257. [Online]. Available: http://dx.doi.org/10.1109/ICPC.2011.40

[7] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *In Proc. of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. Society Press, 2007, pp. 92–99.

[8] T. Ball and S. G. Eick, "Software visualization in the large," *Computer*, vol. 29, pp. 33–43, 1996.

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

[10] L. Wen, R. G. Dromey, and D. Kirk, "Software engineering and scale-free networks," *Trans. Sys. Man Cyber. Part B*, vol. 39, pp. 648–657, June 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1656753.1656759

[11] Y. Yao, S. Huang, Z.-p. Ren, and X.-m. Liu, "Scale-free property in large scale object-oriented software and its significance on software engineering," in *Proceedings of the 2009 Second International Conference on Information and Computing Science - Volume 03*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 401–404. [Online]. Available: http://dl.acm.org/citation.cfm?id=1584349.1585483

[12] J. Lindsay, J. Noble, and E. Tempero, "Does size matter?: a preliminary investigation of the consequences of powerlaws in software," in *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, ser. WETSoM '10. New York, NY, USA: ACM, 2010, pp. 16–23. [Online]. Available: http://doi.acm.org/10.1145/1809223.1809226

[13] 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.

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

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

[16] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 244–257, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.05.044

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

[18] 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.

[19] 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, pp. 9–23.

[20] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, and R. C. N. Moreira, "The evolving structures of software systems," in *Proceedings of the 3rd International Workshop on Emerging Trends in Software Metrics - WETSoM'12*, Zurich, Switzerland, 2012, pp. 1–7.