Power Laws in Smalltalk

Michele Marchesi, Sandro Pinna, Nicola Serra, Stefano Tuveri

Dipartimento di Ingegneria Elettrica ed Elettronica, Università di Cagliari, Piazza d'Armi, 09123 Cagliari, Italy
{michele,pinnasandro,nicola.serra,stefano.tuveri}@diee.unica.it

Abstract

Many real systems have been described as complex networks, where nodes represent specific parts of the system and connections represent relationships among them. Examples of such networks come from very different contexts. Traditional theories suggest to represent complex systems as random graphs, according to the models proposed by Erdős and Rényi. However, there is an increasing evidence that many real world systems behave in different ways displaying *scale-free* distributions of nodes degree. Random graphs are not suitable to describe such behaviors, thus new models have been proposed in recent years. Recently, it has emerged the interest in applying complex network theories to represent large software systems. Software applications are made of modules and relationships among them. Thus, a representation based on graph theory is natural. This work presents our study [1] in this context. We analyzed four large Smalltalk systems. For each one we built the graph representing all system classes and the relationships among them. We show that these graphs display scale free characteristics, in accordance to recent studies on other large software systems.

Key words: software graphs, smalltalk, power-laws, scale-free networks, software architecture

1 Introduction

In recent years, many studies have been carried out on the application of complex networks theory to physical, biological and human phenomena. In fact, many systems can be described as complex networks, whose nodes represent the elements of the system, and edges represent the interactions between them. Examples of such systems are living systems, whose nodes are macromolecules with different functions, connected through chemical links; the stock

[1] This study is part of MAPS research project (Agile Methodologies for Software Production, funded by FIRB research fund of MIUR.)

market, where many traders are connected through information and opinion exchanges; the Hollywood movie system, where stars take part together in a movie; the Web, where pages link other pages. Such networks grow and evolve increasing their complexity in an apparently disordered way.

Using the classical random graph theory developed by Erdős and Rényi [10][11] to model such systems looks sensible, but most of these systems in fact behave according to laws significantly different from those predicted by Erdős and Rényi theory.

More precisely, there is increasing evidence that several real networks behave as *small worlds*, simultaneously showing a short average minimum length path and a high clustering behavior [16]. Moreover, many real networks show interesting laws in the distribution of the number of links connected to a node. The tails of such distributions follow a *power law*, that is a significant deviation from the Gaussian behavior that would be expected if links were randomly added to the network [3]. The meaning of the term *small worlds* and the difference between *power law* and *gaussian law* will be clarified within the next section.

Surprisingly, such properties are common to very different real systems. This suggests that the modeled behaviors are independent of the particular nature of the represented real system. By the contrary, it seems like these are general and universal behaviors which raise as a consequence of the sole complexity of the system.

Large software applications are considered to be among the most complex artifacts ever produced by man [6], and consequently are good candidates to be modeled as complex networks, and to be studied with complexity theory. This is particularly true for object-oriented systems, where objects and classes are natural candidates to be represented as nodes, and the various possible relationships between them – such as inheritance, instantiation, composition, dependence – are represented as arcs connecting nodes. Thus, it is natural to try to describe such systems by complex systems theory models. Very recently, some studies have been performed on software systems showing that run-time objects [13] and static class structures of object oriented systems are in fact governed by scale free power law distributions [14][15]. Other studies have been performed by Myers [12] and Wheeldon and Counsell [17], leading to similar results. Most of these studies of complex software systems are based on C++ and Java code.

In this paper we present a study on the Smalltalk system, a very complex software application that is considered the most pure object-oriented system. Namely, we study the Squeak [9] and VisualWorks [8] Smalltalk implementations. The dynamic typing nature substantially differentiates Smalltalk by other languages such as C++ and Java. This makes the study of Smalltalk systems particularly interesting. Accordingly to our previsions and similarly to other software systems developed using different languages, we will show that Smalltalk systems display link distributions whose tails follow a *power law*. We think that this approach could be an important starting point to better understand the nature and evolution of large software systems. Therefore, with this paper we want to suggest a new point of view for reviewing well known properties of software systems and for leading to new results that are not reachable with the classic software engineering measurement approaches.

2 Small Worlds and Scale Free Networks

Traditionally, complex networks have been described as completely random or completely deterministic. Only recently it has been shown that many real technological, biological and social networks lie somewhere between these extremes. Erdős and Rényi pioneered the study of the behavior of random graphs. According to their model, a "random graph" is made starting from a set of n nodes and connecting each pair independently with a given probability p. Random graphs exhibit many interesting properties, depending on how large is the number of nodes n, and on the connection probability p. The probability distribution of the nodes degree, k in a random graph with n vertexes and z connections is the well-known Poisson distribution:

$$p_k = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} \cong \frac{z^k \cdot e^{-z}}{k!} \tag{1}$$

where the second approximation becomes exact as the number of nodes n becomes very large. Watts and Strogatz [16] observed that a very important property of random graphs is the small value for the average minimum length path, coupled with the low clustering coefficient. In a graph, the minimum length path between two nodes is the shortest path of consecutive edges connecting the two nodes. The average minimum length path is the average of such value taken among all possible node pairs. The clustering coefficient of a graph is a measure of how clustered, or locally structured, a graph is. Specifically, if node i is linked to K_i neighbor nodes, then we define the clustering coefficient for node i, C_i, as the fraction of the total possible $\frac{K_i \cdot (K_i - 1)}{2}$ arcs that are realized between i's neighbors. Empirical evaluation of many real networks shows that they are characterized by a small value of the minimum length path, and by a large clustering coefficient. This means that random graphs are not suitable to model these real-world networks. Watts and Strogatz defined as *small worlds*, the networks characterized by a small average minimum length path, and by a large clustering coefficient, and proposed an interesting method to build small words networks starting by a lattice network. The other main obstacle to apply Erdős and Rényi theory to real networks is the empirical observation that the predicted Poisson distribution of the nodes degree given in equation 1 does not fit the observed nodes degree distribution of a great number of real technological, social and biological networks. Barabasi and Albert [3] showed how this incongruence between the random graphs theory and real networks is due to two main factors not accounted for by the Erdős and Rényi model:

- Real networks expand continuously by the addition of new vertexes
- Connection between nodes are preferential rather than independent

The above conditions are demonstrated to be sufficient conditions for displaying a power law tail in the distribution of the node degrees:

$$p_k \propto k^{-\gamma} \tag{2}$$

The theoretical model provided by Barabasi asserts the value for the exponent to be $\gamma = 3$. Empirical studies on real networks such as the Web, the North American power grid, the Hollywood actors network and others, display values for the exponent very close to the theoretical γ.

It is very surprising how many real systems modeled as networks reflect this power law distribution in accordance with theoretical evidence. This suggests that many real systems share common properties and behaviors in their structure, which are independent of the particular nature of the system itself. Those properties seem to be related exclusively with the complexity of the system modeled. Software systems play an important role in this context, as they are obviously characterized by a high level of complexity.

3 Software Applications as Complex Systems

Software applications and architectures have become ever larger and more complex over the past years. So, it is sensible to apply complex systems and graph theories to model and to study large software applications. The study of software systems as complex networks has a great scientific interest. It is acknowledged that software networks play a special role in the context of random networks theory, as it seems reasonable that the small world and scale-free signature origin from different factors compared to other real networks. In fact, software is built up out of many interacting modules and subsystems at many detail levels (methods, classes, hierarchies, libraries, etc.), and good software is developed following collective and collaborative designs, design patterns and optimization principles, rather than an explicit preferential attachment. This suggests an alternative scenario for generating scale-free networks, which is more related to optimization rather than to Barabasi hypothesis[14]. This enforces the idea that the power law shape of degree distribution tails describes general behaviors of complex networks.

Moreover, the analysis of software systems structure and evolution as complex networks could also be of practical interest, as it might provide an alternative perspective able to help a better understanding of the mechanisms ruling software production or the relationships among network structure and software quality. For instance, an entire new bunch of metrics computed on the program graph could be introduced, and correlated with external software metrics.

One of the first studies of this kind[13] has shown that the graphs formed by run time objects, and by the references between them in object-oriented applications are characterized by a power law tail in the distribution of node degrees. Valverde and Sole[14][15] found similar properties studying the graph formed by the classes and their relationships in large object-oriented projects. They found that software systems are highly heterogeneous *small worlds* networks with scale-free distributions of connections degree. Wheeldon and Counsell[17] performed similar studies on Java projects. Myers[12] found analogue results on large C and C++ open source systems, considering the collaborative diagrams of the modules within procedural projects and of the classes within the OO projects. He also computed the correlation between complexity metrics and topological measures, reveling that nodes with large output degree tend to evolve more rapidly than nodes with large input degree.

Our study is a contribution to the open issues quoted above. This paper introduces a procedure for building the class graphs of a group of Smalltalk programs, and presents the related results. Smalltalk language provides a powerful environment to easily build and analyze graphs representing its own classes. Moreover, the dynamically typed structure of the language adds more interest as it represents a fundamental difference with respect to other related studied languages. On the other hand, for the same reason, it is hard to statically resolve all class relationships.

4 Building the Smalltalk Graph

A software system is made of modules and relationships between them. More precisely, a software application developed according to the object-oriented paradigm is made of classes and well defined relationships between classes. Representing such a software system as a graph is natural. In this graph, each node represents a class within the system, while the arcs between nodes represent the relationships among classes. An arc links a couple of nodes, one representing the starting class, and the other the ending class. The graph is thus an oriented graph. An arc has also a weight, which is a measure of the degree of the relationship between the two classes. While there may be many kinds of relationships between classes, for the sake of simplicity, in this study we have considered just two kinds of relationships: inheritance and dependence, each one giving a different contribution to the arc weight. The inheritance is represented by an arc with unitary weight, from the subclass to the superclass.

As regards dependence, we say that class A depends on class B when one of the following conditions holds:

- class A has an instance variable whose type is B (composition);
- a method of class A has a variable (parameter or local variable) whose type is B;
- an object of class B is obtained by a method call, or created, inside a method of class A.

The dependence analysis must consider that Smalltalk is not a typed language. It is a dynamic, implicitly typed language where objects, not variables, carry type information. This frees the programmer from declaring variable types, but embeds less information into the source code. None of the three conditions reported above can be easily tested by an analysis of Smalltalk code. We observe, however, that having access in the code of a method to a variable of a given class is significant only if one or more messages are sent to this variable. So, we define that class A depends on class B if a method of class B is called from within a method of class A. If the considered method has only one implementor, class B, the dependence link is clearly unambiguous. If the method has more than one implementor class, say n, it is not possible to ascertain with a static analysis which one is the right one.

In this situation, we decided to introduce a dependency towards all implementor classes, weighting such a dependency with weight $1/n$. If a method of class B is called by more than one method of class A, or more than once within the same method of class B, we introduce weighted arcs for every call. Inheritance is not considered in the dependency analysis. For instance, let us suppose that class C is subclass of class A, that class D is subclass of class B, and that a method of B – which is not overridden in class D – is called inside a method of A, which in turn is not overridden in class C. At run time, it may be possible that an object belonging to class C calls the method, and/or that such a method is in fact executed on an object belonging to class D. Nevertheless, in this case only a dependence between class A and class B is recorded. Note that this issue holds also for typed languages, such as Java or C++, when dynamic binding is used.

In conclusion, our graph is a collection of nodes:

$$G \equiv \{N_i\}_{i=1..N_c} \tag{3}$$

where N_c is the total number of classes within the represented software system. Thus, the graph contains one node for each system class C_i. A node N_i has a collection of links, each representing a relationship that the class represented by N_i has with other classes in the system. Links, or arcs, are pairs of nodes representing respectively the starting and the ending class:

$$L_{ij} = (N_i, N_j) \tag{4}$$

A link L_{ij}, carries also a weight that we indicate as $W(L_{ij})$. Now let's introduce the symbols we use for representing methods, messages and implementors:

- $M(C_i) = \{methods\, of\, the\, class\, C_i\}$
- $S(m) = \{messages\, sent\, inside\, m\}, m \in M(C_i)$
- $P(s) = \{implementors\, of\, message\, \mathbf{s}\}, s \in S(m)$
- $dim(P(s)) = number\, of\, implementors\, of\, message$ **s**

We define the weight of a link as the sum of related dependence, W_{dep}, and inheritance, W_{inh}, contributions:

$$W(L_{ij}) = W_{dep}(L_{ij}) + W_{inh}(L_{ij}) \tag{5}$$

$$W_{dep}(L_{ij}) = \sum_{m \in M(C_i)} \sum_{s \in S(m)} \frac{1}{dim(P(s))} \cdot I_s(C_j) \tag{6}$$

$$I_s(C_j) = \{^{1\ if\ C_j \in P(s)\ (C_j\ is\, an\, implementor\, of\ s)}_{0\ otherwise} \tag{7}$$

$$W_{inh}(L_{ij}) = \{^{1\, if\, C_j\, is\, the\, direct\, superclass\, of\, C_i}_{0\, otherwise} \tag{8}$$

Now we can define both the input degree, and output degree of node N_i:

$$outputDegree(N_i) = \sum_{j=1}^{N_c} W(L_{ij}) \tag{9}$$

$$inputDegree(N_j) = \sum_{i=1}^{N_c} W(L_{ij}) \tag{10}$$

The graph built in this way reflects the relationships between the classes, using in the least biased possible way the information that can be inferred by static analysis of a Smalltalk system. The input degree of a class is directly linked to the usage of this class in the system.

We performed also another measurement on Smalltalk system, considering the distribution of classes implementing a method. For each method selector, M, we easily found in the system its implementors, $P(M)$, and their number, $dim(P(m))$.

5 Analyzing Smalltalk Classes

Once defined the graph model used to capture the structure of the software system under study, we need to build an analyzer able to generate such a graph, starting from the software system. With strong typed languages, this would be accomplished by parsing the source code, recognizing class and variable definitions, and generating the graph, as in [17]. Another approach is to use a CASE tool able to reverse-engineer the code, building the related UML class diagram [5]. The code analyzer can then take advantage of the navigation API of the CASE tool, to explore class relationships and to build the graph. Following this approach, it is possible also to directly analyze UML design models. This is the approach followed in [15].

With Smalltalk, we can take full advantage of the introspection of the language. The whole system is accessible from within itself, and no source code parser or reverse engineering is needed. Moreover, Smalltalk is endowed of powerful query methods able to return useful information, as for instance the set of all the classes of the system, the set of classes implementing a given selector, the set of messages sent within a given method, and so on. We remember that in Smalltalk everything is an object, including the classes, the methods, the message calls themselves. We analyzed two dialects of Smalltalk – Squeak (version 3.5), an open-source implementation [9], and VisualWorks (version 7.2), a commercial implementation which is freely available for non-commercial uses [8]. The system classes and methods to perform queries about the system differs between Squeak and VisualWorks, but they are substantially equivalent. We defined a simple data structure able to hold the needed information on the graph describing the system. Figure 1 shows the UML class diagram depicting this structure. A **Graph** is a collection of nodes; a **Node** has a related class and a collection of links. A **Link** has a startNode, an endNode and a weight.

To give an insight on how the dependence analysis was performed, we shortly describe its implementation in Squeak. The main classes involved in the computation are **SystemNavigation**, **Class** and **CompiledMethod**. The method

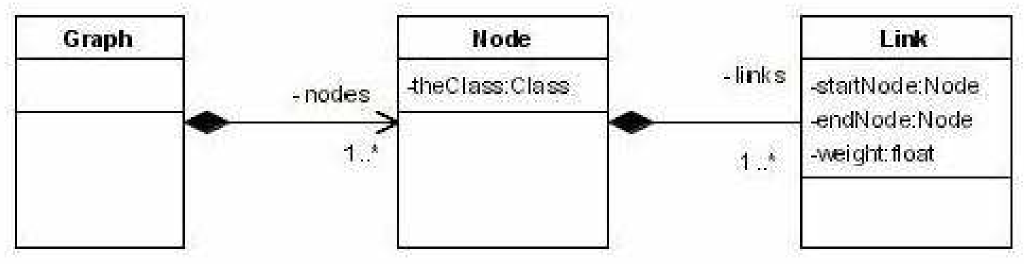

Fig. 1. *The graph structure*

allClasses of **SystemNavigation** returns the collection of all classes in the system. For a given class, the method **selectors** returns all selectors of that class. A selector is a Symbol (a kind of String) representing a method signature. The whole method, instance of class **CompiledMethod**, can be obtained by sending the message **compiledMethodAt: aSelector** to the class. The **CompiledMethod** class implements the method **messages** which returns the collection of all messages sent inside the CompiledMethod. A message is in turn a Symbol representing the selector of the corresponding invoked method. The **SystemNavigation** class implements the method **allClasses-Implementing: aMessage** which returns a collection of all classes implementing the given message. In this way, it is possible to navigate the system, building the graph described in the previous section.

A similar approach has been adopted for Visual Works even if the involved classes are not the same in the two systems. For example in VW we use the class **Refactory.Browser.BrowserEnvironment** instead of **SystemNavigation**, and so on. The graph construction has taken about four hours in Squeak (1797 classes), and three hours in VisualWorks (2227 classes) on a PC running Windows, powered by a 2.6 GHz processor.

6 Results

Our study has been structured across the following points:

- Building of the class relations graph
- Computing the survival distributions of the input degree and output degree
- Plotting the survival distributions on a Log-Log plot
- Discussing the results

Our main purpose is to verify that the distributions tails are better fitted by a power law rather than by the Poisson distribution typical of the random graphs. Moreover, we want to interpret the results in a less general context and from the more practical point of view of the software developer.

Namely, we analyzed four Smalltalk systems:

- Squeak
- Visual Works with three different parcels installed:
  - Base image
  - Base image plus Jun parcels
  - Base image plus VisualWave parcels

Jun is a large 3D graphical application. VisualWave is the VisualWorks extension to support the development of Web applications. Note that the VisualWorks parcels are analyzed together with the base image, so the results referring to these cases are clearly correlated to the results referring to the base image.

6.1 Distributions of input degree and output degree

Figure 2 and figure 3 display the distribution tails of the input degree and output degree respectively for VisualWorks and for Squeak systems. More precisely, we computed the survival distributions and plotted them in a log-log plot. The survival distribution depicts the probability that a value exists that is greater than the current one. The log-log plot performs a transformation of the power law function into a straight line function, with slope equal to the power exponent of the original curve. This allows to visually better verify the presence of a power law and to estimate the γ coefficient.

As expected, the plots show that distribution tails are actually well fitted by a power law. Table 1 shows the fitted γ exponent of the power law distributions obtained for each system graph, compared with the number of classes of the analyzed systems. The power exponent γ of the distributions is quite close to the theoretical value ($\gamma = 3$) obtained by Barabasi and it is coherent to the empirical values ($2 < \gamma < 4$) obtained for many real networks. An higher exponent denotes a tail decreasing quicker. A small exponent denotes a "fatter" tail, that is a bigger deviation from the behavior of a Gauss or Poisson distribution.

Despite this general result, which is in fact shared by various kind of different real networks, our main interest is to interpret its meaning in the specific context of software systems. The number of outgoing links of a node is a measure of how many messages are sent from within the methods of the corresponding class. A high value of the number of outgoing links denotes that the class performs many message calls from within its methods. This can be considered also as a measure of the coupling between classes. Thus, a power law distribution with fat tails is a clue that the system is characterized by a certain number of classes with an high level of coupling, while the bulk of the classes tends to be much less coupled. This interpretation is supported by other previous studies. For example, Chidamber and Kemerer [7] and Basili [4] found similar distributions for coupling metrics.

Note also that, since the output degree of a node is not divided by the number of methods of the related class, it is clearly proportional to the number and size of methods of the class. Consequently, the power law behavior seems to be related also with the distribution of class sizes, and with the fact that, when a new method is added to the system, it is more likely that it belongs to a class that already has many methods. However, both high coupling and big classes are not considered good object oriented programming style. A good object oriented system should be composed of many cohesive classes, at different detail levels, each one focusing on a single task, and endowed with a few, short methods. Our empirical analysis of the Smalltalk system shows that this is not the case, because the distribution of the number of outgoing links decays with a power law. This observation is confirmed by other studies, showing the same behavior in Java applications [15].

A broad analysis of many OO systems is outside the scope of this paper. However, these behaviors clearly point out an interesting research direction, suggesting to perform more detailed analysis on this indicator, for instance analyzing separately system classes and classes written by programmers, and analyzing the values of the outgoing links power law degree in systems with different quality rating, or before and after an extensive refactoring.

The other main indicator we examined is the input degree, which gives a measure of how a certain class is referenced by other classes in the system. The shape of the distribution is due to the fact that in Smalltalk system, there are "service" classes that are used by almost other classes. Classes such as **Object**, the **Collection** and the **Magnitude** hierarchies, come immediately to mind. There are also classes which are fairly used, both in the system and in specific parcels, while most other classes are rarely used. The usage rate is certainly not random. This behavior suggests the hypothesis that when defining a new class, it is likely to be subclass of a class already having more subclasses, and that when writing a new method the probability to send a message implemented in another class is roughly proportional to the number of sending of that message in the system. In fact, this probability is more than proportional to this number, as the coefficient values smaller than 3 suggest.

6.2 Distributions of method implementors

We also computed the distribution of the method implementors on the VisualWorks system, respectively for the base image and for the base image with

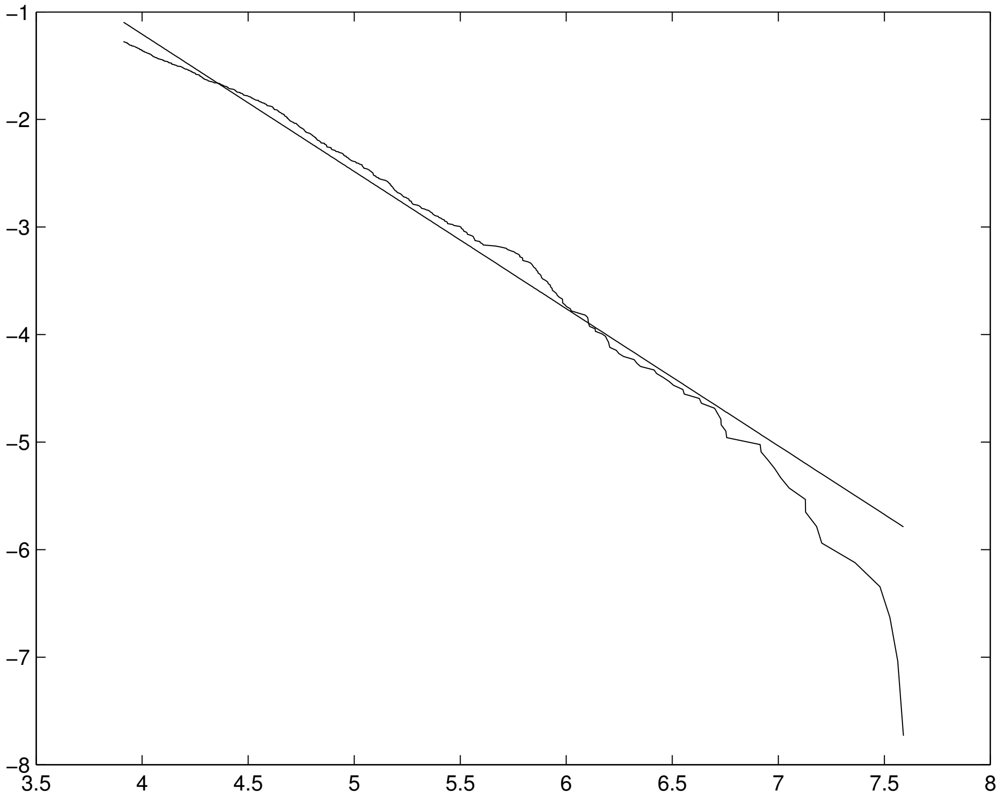

Fig. 2. *Log-Log plot of input degree survival distribution computed on VisualWorks system. Similar plots are obtained for other analyzed systems: Squeak, VisualWorks with Jun and VisualWorks with VisualWave.*

	inDegree	**outDegree**	**number of classes**
Squeak	-2.07	-2.3	1797
Visual Works	-2.28	-2.73	2227
Jun	-2.39	-2.55	3022
Visual Wave	-2.26	-2.53	2648

Table 1
Power Law exponents for input degree and output degree distributions.

Jun and VisualWave parcels loaded. Figures 4 and 5 show the log-log plots of the survival distributions for the number of implementors of each method for VisualWorks base image and Jun, together with best-fit Poisson distribution of the same data.

Table 2 shows the exponent γ of the power law distributions obtained for each implementors graph, compared with the total number of considered methods.

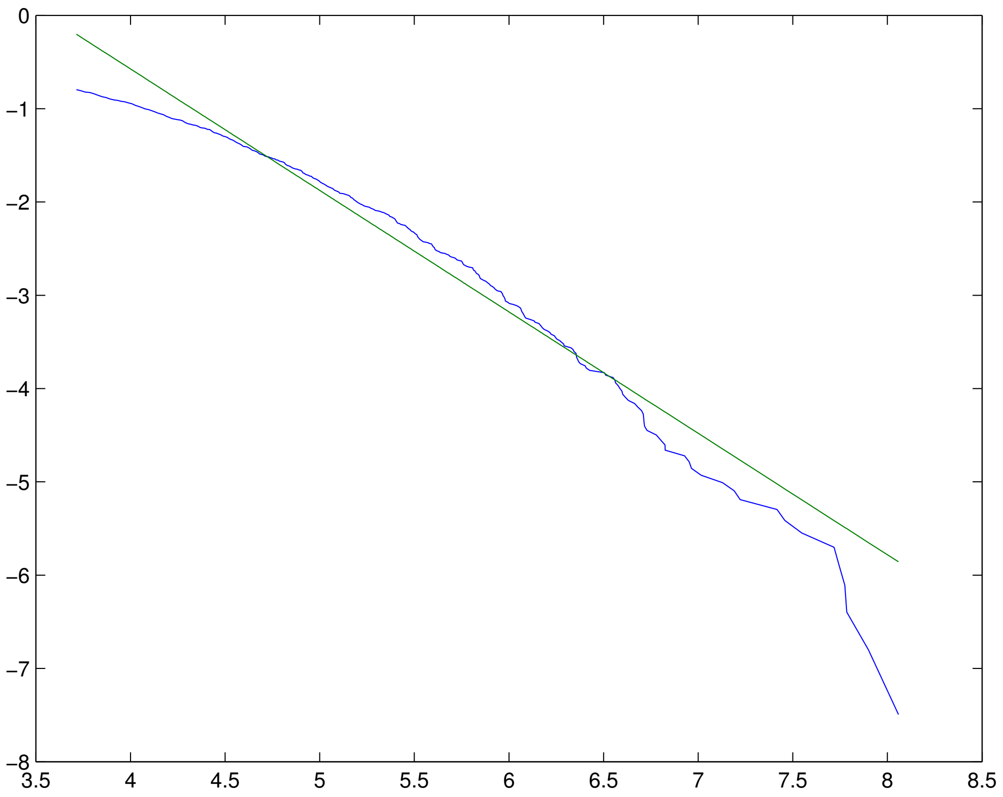

Fig. 3. *Log-Log plot of output degree survival distribution computed on Squeak system. Similar plots are obtained for other analyzed systems: VisualWorks, VisualWorks with Jun and VisualWorks with VisualWave.*

Clearly, also the tail of these distributions show a power-law behavior, with exponent lying in ranges similar to those of previous distributions.

This behavior denotes that, in the Smalltalk system, there are methods with the same name implemented in many classes, while most methods are implemented in one or few classes. This means that, when defining a new method, the probability that its name is the same of a method already present in the system is roughly proportional to the number of times that method is implemented in different classes. We know that it is good programming practice to give the same name, in different classes, to methods performing the same kind of service. This seems to be confirmed by the behavior of these distributions.

7 Conclusions and further works

In this paper we have studied distribution laws related to object-oriented class relationships of large system and application libraries of a “pure” object-

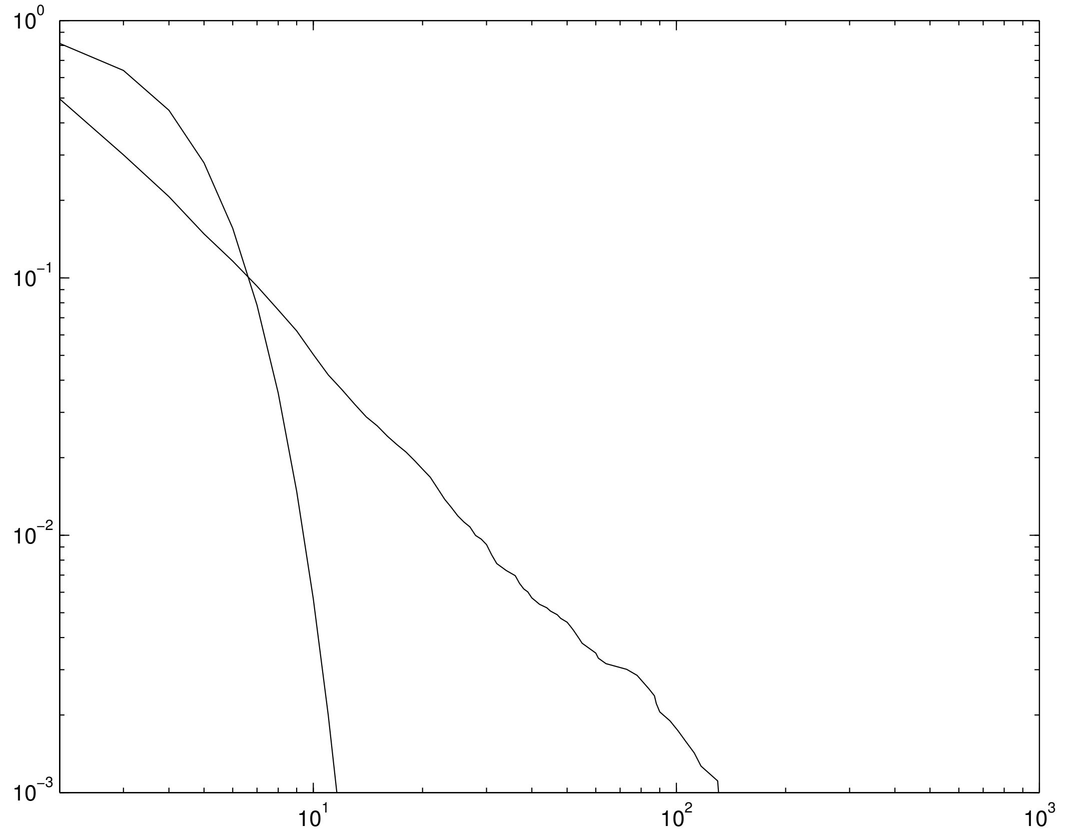

Fig. 4. *VisualWorks, log-log plot of survival distributions of the number of implementors of each method. The best-fit Poisson distribution of the same data is also shown.*

oriented system as Smalltalk. Not unlike very recent findings by others on different object-oriented designs and systems, we found that key statistical distributions of the class-relationship graph exhibit scale-free and heavy-tailed degree distributions qualitatively similar to those observed in recently studied biological and technological networks. Moreover, these distributions show strong regularities in their characteristic exponents.

Following Wheeldon and Counsell [17], we believe that these regularities are common across all non-trivial object-oriented programs. This is a strong impli-

	$-\gamma$	**Number of Methods**
Visual Works	-2.56	23883
Jun	-2.27	33489
Visual Wave	-2.54	27780

Table 2
Power Law exponent for the distributions of the number of implementors of each method

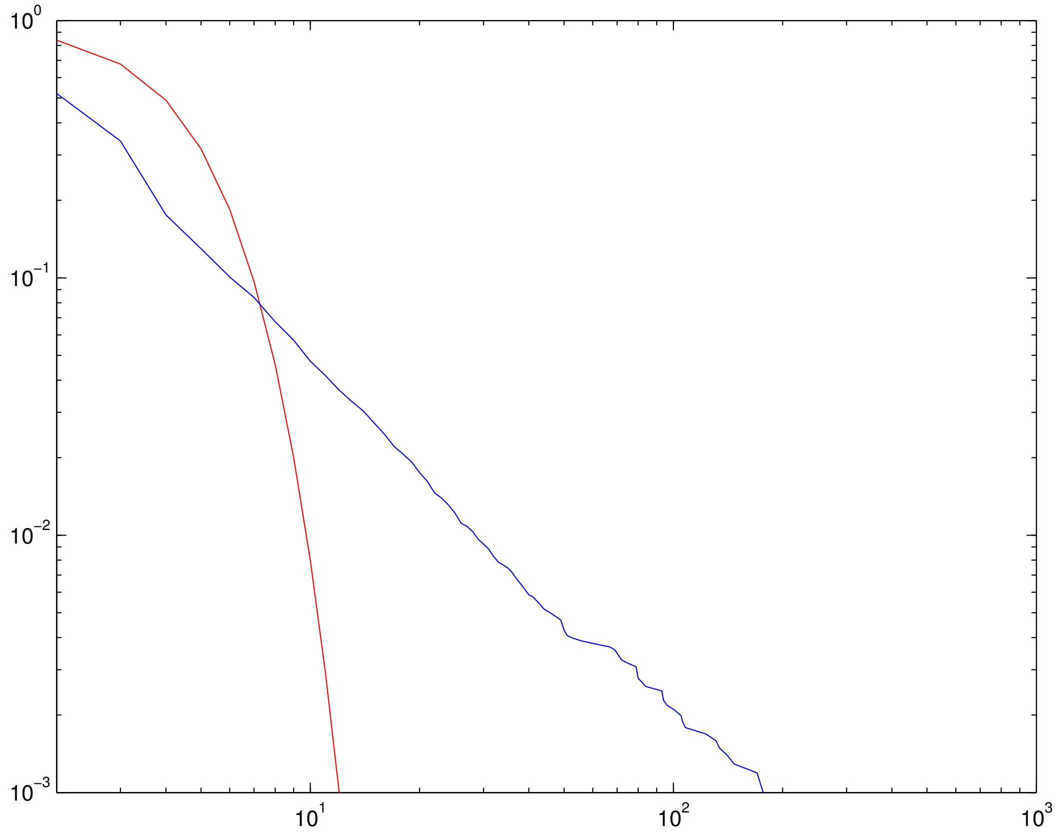

Fig. 5. *VisualWorks with Jun parcel loaded, log-log plot of survival distributions of the number of implementors of each method. The best-fit Poisson distribution of the same data is also shown.*

cation indeed, and could show the path of future research along the following directions:

- What are the statistical mechanisms involved in software development, and how it is possible to model them, accounting for the hierarchical nature of software design, and for the relationships among network structure, object interactions, and system evolution?
- During system development, when these scale-free patterns do emerge? Is it possible to develop a growth theory of software graphs, which in turn could be used to predict the dimensions of future systems and to estimate the complexity of developing and maintaining those systems?
- Is it possible to correlate statistical graph properties with software quality? What about the impact of refactoring on the software graph?
- Are there differences in statistical graph properties between open-source software, which is developed by many programmers in a somewhat destructured way, and commercial software developed following a strict process by full-time developers?
- Are there differences in statistical graph properties between software de-

veloped in the agile way, with short iterations, test-driven development, continuous integration and extensive refactoring, and software developed with waterfall-like approaches?

Answering to these questions using an innovative approach such as the augmented random graph theory could break new grounds in software engineering, and could be very valuable.

As a further conclusion, we agree with Valverde [15] that software systems present novel perspectives also to the study of complex networks. Software must be both functional and evolvable, and unlike biological systems it is to a large extent designed in advance. Traditional software does not emphasize redundancy to support fault tolerance, but it presents other degrees of freedom that play a central role in supporting evolvability, such as modularity, layering, cohesion, genericity, polymorphism, and collaboration. Are some of these degrees of freedom relevant also to the organization and evolution of biological networks? The study of software systems from the new perspective of statistical graph properties may be also useful in suggesting novel insights into collective biological function.

References

[1] Réka Albert, Hawoong Jeong, Albert-László Barabasi: Attack and Error Tolerance of Complex Networks. Nature Vol. 406, pp 378-382 (2000)

[2] Albert-László Barabasi: Linked: the new science of networks. Persus Press, New York (2002)

[3] Albert-László Barabasi, Réka Albert: Emergence of scaling in random networks. Science Vol. 286, pp. 509-512 (1999)

[4] Victor R. Basili, Walcelio L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicator, IEEE Transaction on Software Engineering, Vol.22, No. 10, (October 1996)

[5] Grady Booch, Ivar Jacobson, James Rumbaugh: The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, (1999)

[6] Frederick P. Brooks: The Mythical Man-Month. Addison-Wesley, (1995)

[7] Shyam R. Chidamber, Chris F. Kemerer, A Metric Suite for Object Oriented Design, IEEE Transaction on Software Engineering, Vol.20, No. 6, (June 1994)

[8] Cincom Corp., VisualWorks Application Developer's Guide. Cincom, (2004)

[9] Mark J. Guzdial: Squeak: Object-Oriented Design with Multimedia Applications. Prentice-Hall, (2000)

[10] Paul Erdős, Alfred Rényi: On random graphs.I, Publ. Math. Debrecen 6, pp. 290-291, (1959)

[11] Paul Erdős, Alfred Rényi: On the Evolution of Random Graphs.Publ.Math. Inst. Hungar. Acad. Sci. 5, pp. 17-61, (1960)

[12] Christopher R. Myers: Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. Phys. Rev. E 68, 046116 (2003), preprint at cond-mat/0305575

[13] Alex Potanin, James Noble, Marcus Frean, Robert Biddle: Scale-free geometry in Object Oriented programs. Victoria University of Wellington, New Zeland, Technical Report CS-TR-02/30 (2002)

[14] Sergi Valverde, Ramón F. Cancho, Richard V. Sole: Scale-Free networks from optimal design, Europhisics Letters 60, pp. 512-517 (2002)

[15] Sergi Valverde, Richard V. Sole: Hierarchical small worlds in Software Architecture. Submitted to IEEE Transactions of Software Engineering (2003)

[16] Duncan J. Watts, Steven H. Strogatz: Collective dynamics of 'small-world' networks. Nature Vol. 393, pp. 440-442 (1998)

[17] Richard Wheeldon, Steve Counsell: Power law distributions in class relationships.Proc. Third IEEE Int. Workshop on Source Code Analysis & Manipulation, Amsterdam, The Netherland, (Sept. 2003)