Hierarchical Small Worlds in Software Architecture

Sergi Valverde and Ricard V. Solé

***Abstract*— The components of a large software application do not interact in random ways. Instead, class diagrams exhibit remarkable topological similarities to other natural and artificial systems. The components of a large software application are very well connected because the mean shortest distance between them is very low in spite of having a relatively small number of connections per class. In addition, these diagrams are very heterogeneous. These measurements are of a general nature and are largely independent of the particular semantics of the application. As shown in this paper, and irrespective of the specific features of each system analyzed, the final outcome of software evolution is a small world, hierarchical class diagram with well-defined statistical properties. The consequences for software evolution are outlined.**

***Index Terms*— Software Architecture, Scale-Free Networks, Small World Networks, Software Metrics, Maintenance Costs, Software Reliability, Modular Systems.**

I. Introduction

Software development is a challenging activity that requires considerable expertise. A software application is a carefully crafted piece of engineering. The essence of a very well designed application is a critical trade-off between ease of maintenance and running performance. Such balance is very difficult to achieve. Inappropriate optimization for efficiency often produces code that is difficult to understand and extend: Premature optimization is the root of all evil (originally quoted by Tony Hoare and restated later by Donald Knuth). The prematurely optimized program freezes into a rigid state. From this state, is very difficult to plan its adaptation to new features.

Observing the difficulties faced by the young software industry [1], Dijkstra proposed that we should restrict ourselves only to programs that admit a logical organization in the framework of clear hierarchies [2]. Hierarchies isolate software parts and this separation, if effective, enables low cost replacement and/or modification. Anyway, large software tends to be a tangled structure of interrelated concepts. Unfortunately, we simply do not know yet the limits of disentanglement [3]. This overwhelming complexity requires expert developers able to put some order in it [4]. The recurrent (and unanswered) question is why is so difficult to build software systems? Besides current tinkering-based approaches, is there any safer way to develop software?

The objective of the present work is to enhance our comprehension about the nature of software design by quantitative measurements and structural analysis on existing software systems. It seems that minimization of uncertainty at low cost lies at the root of the observed software complexity. There are many sources of uncertainty that difficult software development. Specifically, trying to design predictable computations in spite of the surrounding uncertainty gives structure to our systems. In addition, it might be that reliability requirements severely constrain the space of possible software designs. One expects that a variety of mechanisms will be required for handling many different types and arbitrary sequences of inputs. No single and general computation will be able to parse efficiently such complex environment without incurring into high costs. Clearly, the system needs a certain degree of specialization. This is reflected by the many components and interconnection patterns used by large-scale software applications. We will show how to measure this heterogeneous structure in real software designs.

Next section introduces the different conceptual levels involved in software development. Section III presents software graphs, how to obtain them from existing system representations, and describes the analyzed datasets. Sections IV, V and VI are a short introduction to the theory of random graphs, small worlds and scale-free networks which is the basis for the software measurements. Section VII presents a known measure of software cost and a prediction of this cost based on real statistics. Section VIII explores the relationship between observed regularities and software evolution. Finally, section IX concludes the article with some discussion and comments about the implications on software engineering and future work.

II. Software Levels

Software evolves over time. The traditional scheme of developing a software application is a sequence of write-and-test stages towards the working release. Every new version adds or improves functionality of older versions. Development takes place simultaneously at several and clearly defined conceptual scales: implementation, component and architecture levels. This arrangement is arbitrary but traditionally helps developers to switch

Manuscript received July 3, 2003. This work is supported by the Santa Fe Institute and by grant BFM2001-2154.

S. V. is with the ICREA-Complex Systems Lab, Universitat Pompeu Fabra (GRIB), Dr. Aiguader 80, Barcelona 80003, Spain (e-mail: svalverde@imim.es).

R. V. S. is with the ICREA-Complex Systems Lab (full address above) and with Santa Fe Institute, 1399 Hyde Park Road, NM 87501, USA (phone: (34) 935422821; e-mail: ricard.sole@cexs.upf.es).

between low-level details and global views of the entire system when necessary. Each conceptual level will generate different types of defects. Anyway, the most problematic defects to fix involve all levels and require long and expensive trial-and-error developer cycles.

At every development stage (or software version), new defects are identified and the engineer plans new changes to fix them. Any useful software development scheme must address the issue of software defects. It is very difficult to predict how many resources are required to fix all defects. Most studies about software development clearly correlate cost (human and time resources) and maintenance tasks. A large fraction of these defects requires very few resources and is solvable with minimum effort. However, there is always a small fraction of defects very hard to fix. A preventive development scheme is thus preferable to the well-known expensive bug repair process. This means looking for safer ways to write software without introducing errors.

Every software application is a sequence of machine instructions. The sequence controls the behavior of the computer, which sequentially reads, decodes and interprets them in a continuous loop. Modern software comprises a very large number of instructions (from thousand to millions of basic instructions). It is surprising that the time spent by human developers for building a program of N instructions is relatively short when compared to the size of the entire space of possible programs of the same length. In order to simplify the identification of the desired final sequence, engineers structure software applications in separated components. The engineers exploit the fact that the high-level organization of the raw sequence restricts considerably the functionality of every component (that is, the sub-sequence of instructions corresponding to every component).

Here we use a broader component definition that depends on the programming paradigm used. A component maps to a function or procedure in procedural paradigm and a class for object-oriented paradigm. The component partition assumes that it is easier to solve a complex problem when we consider a subdivision into several simpler sub-problems [5]. Partitioning enables various developers to work simultaneously in different parts of the software system with little or no communication. This division of work not only reduces the cost but also has the additional benefit of limiting the consequences and the propagation of defects. Anyway, it is unclear how to make the best partition. We will return to these questions later on the paper.

The designer assigns clear responsibilities to each component and controls their degree of interaction through the interface definition. Components can be dependable in two different ways: (i) component access data stored in another (data dependence) or (ii) transfers control flow when calling a subroutine (control flow dependence). Both types of dependency represent ways of information transfer between components. Designing a computation involves the definition of an information flow traversing a chain of related components. Often, a component makes sense only

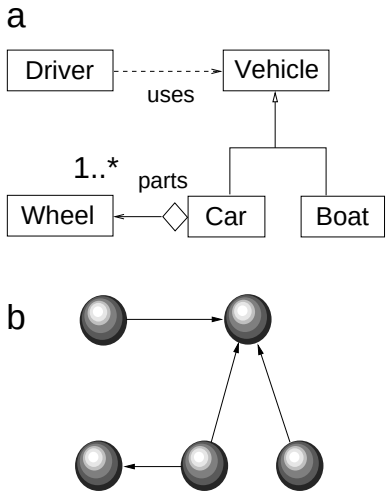

Fig. 1. (a) A simple UML class diagram. (b) And its interpretation as a plain software graph (see text). Every class maps to a single graph node. The relationships between classes are represented by links. Note that no distinction is made between use, inheritance and composition. Link directionality is discarded in some measurements (see text for details).

in the context of a computation involving several components. High software performance means efficient information transferring between components. This sometimes will require introducing internal working details of components into other ones. As we will see, this is a source of design problems.

Good engineering practices promote minimization of component coupling while maximizing the internal cohesion. Too many dependencies result in closely coupled components, which is undesirable because the large amount of communication required. Low cohesion is a symptom of poorly designed component, which address different and unrelated tasks. Unfortunately, the 'minimum coupling/maximum cohesion' principle is a conflicting design goal [6]. Because the constraints, real software components subscript to roughly three different ways of processing information: producers (high fan-in), consumers (high fan-out) and producers/consumers (mixed fan-in and fan-out). The former type of component promotes effective reuse of code while the latter can be problematic in components with a huge number of inputs and outputs. Note that is difficult to change highly reused nodes because their importance for the stability of the system (see below).

There are important differences between designing small-scale software and large-scale software [7]. Constructing large-scale software means not only to solve programming problems but also to orchestrate well the communication between different parts of the system. Software architecture is a high-level construction that addresses communication and coordination problems. Solving these tasks is relatively independent of implementation level details. Higher software levels involve collaborative efforts and complex planning tasks. Designers are aware of the importance of properly assembling the components and avoiding certain

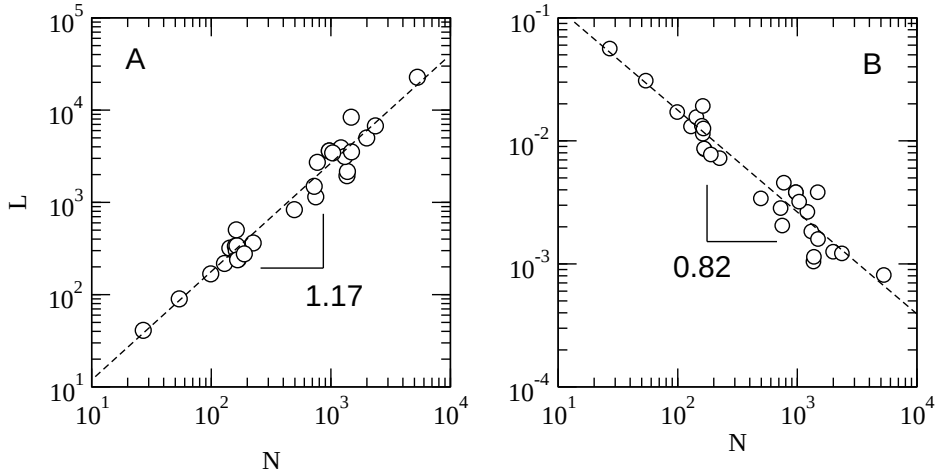

Fig..2: (A) The number of links in a software graph scales linearly with system size. This suggests that software systems grow at a constant rate and independent of system size. (B) The connectance or abundance of possible links realized in software graphs decays as system size N increases. The connectance of large systems is very small (about 0.1% of all possible links). This lacking of links makes difficult to change the global structure of large-scale software systems.

problematic connection patterns between components. In large-scale software, we find a huge variety of design patterns that involve the interaction of many components [8]. Anyway, software suffers certain degradation (entanglement) if software developer does not take special care [9]. As the design evolves, components will adopt functionalities or even replace some other existing components. Corrective actions or re-factorings regularly applied enable faster growing and allow simpler maintenance tasks.

Surprisingly, the large-scale structure of software designs exhibits striking patterns that remarkably depart from random structures and are often close to biological networks [10]. The causes of the emergence of these patterns remain unknown but new theoretical approximations are helping understanding them. The study of these patterns could offer considerable insight into the mechanisms underlying software design. Here, we look for explaining these regularities as well as their implications.

III. Software Graphs

A way to understand natural and artificial systems is by means of looking at the structure of the relationships between their constituting components. Topological measurements of these networks reported striking regularities between very different systems like the Internet, the proteome, food webs, electronic circuits or social networks posing several questions about the universality of the underlying working mechanisms [11]. Recently, we reported the first evidence for the existence of similar patterns in large software designs as well [12]. In order to detect these regularities, we must be able to define a graph G=(V, E) for the software under consideration. Let V= $\{v_i\}$, (i = 1…N) be the set of nodes and L = $\{(v_i, v_j)\}$ the set of links, that is, a directed graph Λ=(V,L). From the previous graph, we get the undirected graph G where E=$\{\{v_i, v_j\}\}$ is the set of edges. In this context, we do not make any difference between the terms graph and network. Each node is characterized by its degree k_i, which is the number of edges attached to it. Analogously, we define the in-degree and the out-degree of a node as the number of links entering the node and the number of links exiting the node, respectively (i.e: the degree is the sum of the in-degree and the out-degree). It is easy to check that the sum of the degrees of all the nodes of a graph is an even number.

For long time it has been recognized the importance of conceiving designs by diagrams [13]. Visualization is a useful tool in this context because puts the focus on relationships between components while hiding low-level details. Such diagrams are very helpful in software design. The present study explores software complexity through the analysis of design diagrams. Most modern software designs follow the object-oriented paradigm. It is a well-established practice to express object-oriented designs using class diagrams. There are several standard notations for these diagrams, UML [14] [15] being the most popular of them. Class diagrams will be our source representation for extracting a raw graph that captures the topological information and discards any other detail. From the class diagram, take every class (or component) and map it into a single node of the software graph. If two classes relate in any meaningful way (say use, inheritance or composition relationship) a link will be set in the associated graph (see Fig. 1). We will consider both directed and undirected versions of the graphs but we do not take into account detailed relationship semantics. Here, relationships have been treated all equal and have the same importance for structural analysis. We studied five UML class diagrams: ProRally 2002, Striker, JDK-A, JDK-B and Mudsi. The two first datasets are videogames from UbiSoft Entertainment. The videogame designs were released with detailed class diagrams documented by their developers. Modern videogames are one of the most complex software systems today because involve extensive user interaction and integration with many specialized hardware (3D-graphics, sound and input peripherals). This kind of software is very expensive and lengthy to produce. Monetary cost is above 1$ million and development time of about two years is not

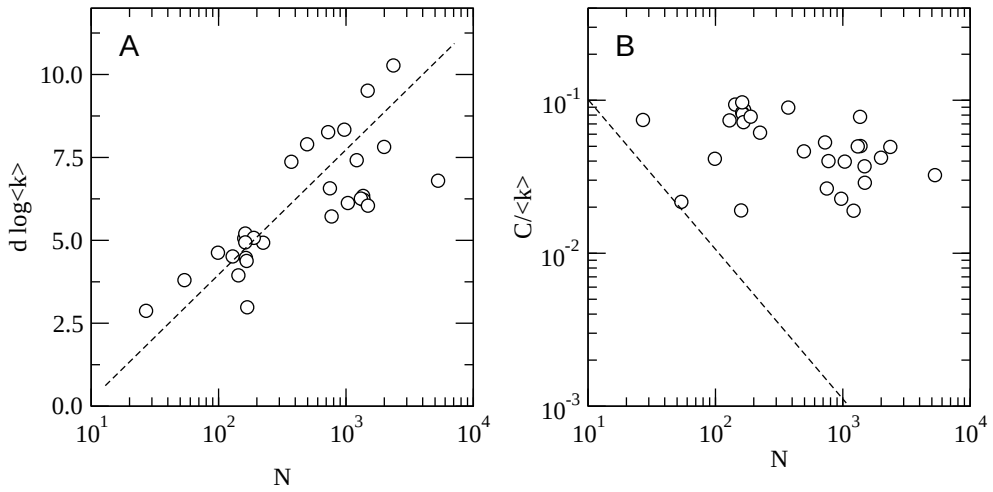

Fig. 3: (A) Average path length against system's size for the networks analysed here. The normalized distance grows with the logarithm of the number of components, as expected in small world networks (see text). (B) Normalized clustering for the systems analysed here strongly departs from the predicted scaling relation followed by random graphs (dashed line). Software graphs are much more clustered (by orders of magnitude) than their random counterparts.

rare. The JDK-A & B datasets were extracted from the public Java Development Kit 1.2. The corresponding UML diagram comes publicly with the Rational Rose98 modeling software. The JDK 1.2 is a framework (or toolkit of classes) that provides useful functionalities for Java programs. We found components for implementing graphical user interfaces, accessing files or networks and mathematical functions. Besides the different objectives addressed by games and the JDK, the main difference is that all components of a typical software application (i.e.: ProRally 2002) are connected into a single component (that is a unique large graph) while in the JDK we found several isolated components spanning a variable number of nodes. Here we only analyze the two largest JDK graphs. The smaller JDK 1.2 graphs are instances of tree-like hierarchies, as commented in [12]. Mudsi is a distributed application written in Java.

In order to improve the statistic analysis, we have also reconstructed (and analyzed) class diagrams from a variety of open source applications written in C/C++ (the reconstruction method is outlined in the appendix). The entire set of 29 class diagrams (both UML and C++) varied in functionality and size. The sample spans three orders of magnitude from N= 27 to N= 5285 nodes. For the sake of comparison, we also added the six networks studied in [16].

In analyzed software graphs, linear scaling approximately holds between the number of links L and number of nodes N (see fig. 2A):

$$L \sim N^{1.17}$$

The above empirical relation has an interesting interpretation. It means that every time the designer adds a new component to the system, it will link on average with a constant number of other existing components. This fits very well with the assumption of a linear growth model [17], independent of the current system size. We will return to this question later in the paper. Another observation is that the linear growth model restricts considerably the possible topologies of large-scale software systems. One can define the richness connectance of a graph as the fraction of used links L compared to the complete graph with N(N-1) links (self-referencing is avoided). It is easy to see that if L scales linearly with N, the richness connectance will decay approximately like 1/N (fig. 2B). This is a hard constraint for evolving large-scale software. Larger systems will require a quadratic amount of links in order to change their global structure.

IV. Random Graphs

Early work on graph theory focused in models of random graphs [18] (see the book [19] for a review). We easily obtain a random graph G_{rand} by distributing a given fraction of edges between randomly chosen pairs of nodes. Specifically, two randomly chosen nodes will be connected with a given probability *p*. This assumes no particular structure and it is the simplest way to simulate a complex network. Random graph plays the role of null model for comparison. In the following sections, we will see how software graphs remarkably depart from this random scenario.

One important thing to notice about random graphs is their regularity. Different random graph sub-regions are all indistinguishable. The homogeneous structure can be detected by measuring the degree distribution *P(k)*, that is, the fraction of nodes having k edges. For a random graph of N nodes and link probability p the degree distribution follows a binomial shape:

$$P(k) = C_{N-1}^{k} p^{k} (1-p)^{N-1-k}$$

The average degree,

$$< k >= \sum_{k=1}^{N} k p_k$$

will be for G_{rand}

$$< k >= pN$$

For large N, a Poisson distribution replaces the previous binomial distribution:

$$P(k) \approx e^{-pN} \frac{(pN)^k}{k!}$$

An important question in relation with random graphs is

TABLE I
Topological Measurements

Dataset	*N*	*L*	*d*	d_{rand}	*C*	$\frac{C}{C_{rand}}$
Mudsi	168	241	2,88	4,95	0,244	0,017
JDK-B	1364	1947	5,97	6,80	0,225	0,002
JDK-A	1376	2162	5,40	6,28	0,159	0,002
Prorally	1993	4987	4,85	4,71	0,211	0,003
Striker	2356	6748	5,90	4,46	0,282	0,002
gchempaint	27	41	2,85	3,26	0,204	0,102
4yp	54	90	3,28	3,44	0,069	0,059
Prospectus	99	168	3,80	3,77	0,14	0,034
eMule	129	218	3,87	4,16	0,237	0,025
Aime	143	319	2,66	3,34	0,413	0,031
Openvrml	159	335	3,53	3,53	0,08	0,026
gpdf	162	300	4,02	3,93	0,303	0,022
Bochs	164	339	3,15	3,60	0,335	0,025
Quanta	166	239	4,31	5,03	0,198	0,017
Fresco	189	277	4,73	4,89	0,228	0,015
Freetype	224	363	4,29	4,71	0,193	0,014
Yahoopops	373	711	5,57	4,47	0,336	0,01
Blender	495	834	6,54	5,14	0,155	0,007
M4	725	1492	5,85	4,66	0,217	0,006
GTK	748	1147	5,87	5,91	0,081	0,004
OIV	1214	3903	3,99	3,82	0,122	0,005
wxWindows	1309	3144	4,03	4,62	0,235	0,004
CS	1488	3526	3,92	4,74	0,135	0,003
Dm	162	254	4,32	4,45	0,304	0,019
Vtk	771	1362	4,52	5,26	0,141	0,005
Xmms	971	1802	6,34	5,23	0,084	0,004
Abiword	1035	1781	5,08	5,75	0,133	0,003
MySql	1480	4200	5,47	4,20	0,21	0,004
Linux	5285	11359	4,66	5,87	0,139	0,001

what are the conditions under which all elements in G_{rand} are connected. This question has obvious relevance in our context, since a connected graph will define a functional structure where either element interact directly or through some sequence of events that must take place on a path on G_{rand} (class sequence diagrams). Intuitively, if p is too small (close to zero) some nodes will be isolated, whereas for large p everyone will have at least one link. The interesting result from random graph theory [19] is that there is a critical probability p_c such that for $p < p_c$ most elements will remain disconnected whereas for $p > p_c$ most elements will belong to the same connected cluster (thus so called giant component). The transition that takes place at p_c is sharp, and implies that a connected system can be reached at a very low cost (for G_{rand}, $\langle k \rangle_c = Np_c = 1$).

Because the mean number of links per node is greater than the random percolation threshold (as seen in previous section), we expect that, with minimum effort, the majority of nodes in the (undirected) software graph will be connected in a single big cluster.

V. Small Worlds

Software graphs strongly depart from the Poissonian picture. The first evidence comes from their so-called small world structure [20], which can be characterized by using two key measures. Consider the following measurement experiment in a graph: choose a pair of nodes (v_i, v_j) and then trace a path from v_i to v_j while traversing the minimum number of edges. Count how many edges have you traversed in such path and you will obtain the length of the shortest path between the two end-nodes: $d_{min}(i, j)$. Now, repeat the procedure for all possible pairs of nodes. What you get is the mean average path length (or characteristic path length) for the graph:

$$d = \langle d_{\min}(i, j) \rangle$$

For many systems, it is useful to keep the average path length very low. Shortest paths mean faster communication. Low average path length has been observed in many real systems like the Internet or different social networks (in the latest, this phenomenon is also popularly known by 'six degrees of separation' [21]) in spite of having a large amount of nodes. We can interpret this measure as network spread or compactness. Random Poissonian graphs are very compact. Because of their regularity, it can be shown that the average path length is proportional to the logarithm of the graph size:

$$d_{rand} \approx \frac{\log N}{\log \langle k \rangle}$$

As we can see in figure 3A the spectrum of software maps analysed here fits very well with the expected small average distance predicted by the previous equation. The average path length provides a global characterization of network organization and indicates that, in spite of the fact that software maps are not random (at all) the characteristic path length is very small and thus communication among different parts is very efficient. For a complete graph characterization, we must consider local features of the network. Take for instance the network of social acquaintances. Expect that two friends of a third person will likely be mutual friends. This situation is depicted in the social network with a triangle whose nodes are persons and edges symbolize friendship. The clustering coefficient C measures the proportion of triangles in the graph:

$$C = \left\langle \frac{2}{k_i (k_i - 1)} \sum_{j=1}^{N} A_{ij} \left[\sum A_{jk} \right] \right\rangle$$

where A is the adjacency matrix for the graph with Aij = 1 if node vi and vj are connected and Aij = 0 otherwise.

Watts & Strogatz found that many real networks display both short mean distances and high clustering or non-negligible cliquishness. A network displaying these properties is a “small world” (SW). Small world networks abound on dense neighborhoods, where few 'shortcuts' (or edges connecting distant regions) have a key role for achieving low average path length. For random graphs, the clustering coefficient is inversely proportional to graph size. Thus, large random graphs will have short paths but very small clustering:

$$C_{rand} \approx \frac{< k >}{N}$$

The software graphs analysed here have clustering coefficients much larger than the expectation from the random scenario (i. e. C >> C_{rand}). In figure 3B we can clearly appreciate this result: the majority of networks have a clustering coefficient orders of magnitude larger than the random expectation. Actually, the values of C seem rather constant for different sizes. This property is known to be present in hierarchical graphs [22] thus indicating that all software graphs follow the same basic pattern of evolution. The hierarchical nature of these graphs is actually a consequence of their modular character, as has been shown within the context of genome maps [22]. Such pattern is fully appreciated in the remarkable approach followed by all systems towards the SW picture. This actually suggests that, beyond the specific properties of each system, common principles of organization result from the design process.

Real networks are sparse, but not as sparse as becoming disconnected into several pieces. There are a minimum number of edges required to connect N nodes. As we have seen before for the random graph, placing more edges will help reducing the average path length. An important thing to note here is that the designer must take into account the cost of additional wires. Most real networks are efficient because they are able to connect very well their nodes (short path distances) with few edges (small mean degree). The software graphs analyzed here are good examples of small worlds.

For instance, the ProRally 2002 game system consists of 1993 nodes and 4987 links: <k> = 5. Most classes participate in few relationships (one or two) but for any pair of classes expect a short chain of relationships in-between (d= 4.85). We provide a comparison for with G_{rand} every software graph of same size N and mean degree <k>. For instance, the clustering coefficient for ProRally 2002 is C = 0.21, which differs considerably from random explanation: C_{rand} = 0.003. A summary of similar results for the other software systems is available in Table I. Beyond the average degree <k> there is a remarkable heterogeneity of connectivities that introduces a new layer of complexity in network's characterization, to be explored in the next section.

VI. Scale-Free Networks

An additional, widespread feature of many complex networks is the high heterogeneity of their degree distributions. Specifically, we have

$$P(k) = Ak^{-\gamma} \exp(k / k_c)$$

where A is a normalization constant, k_c is a cut-off degree and the scaling exponent γ is tipically constrained to a range $\gamma \in (2,3)$. As k_c increases, the tails of the distribution become larger and the graph will display a majority of nodes having few links and a small number of nodes (the hubs) having a large number of connections [23][11]. These graphs are called "scale free"' (SF) and are found in many different contexts, from natural to technological systems.

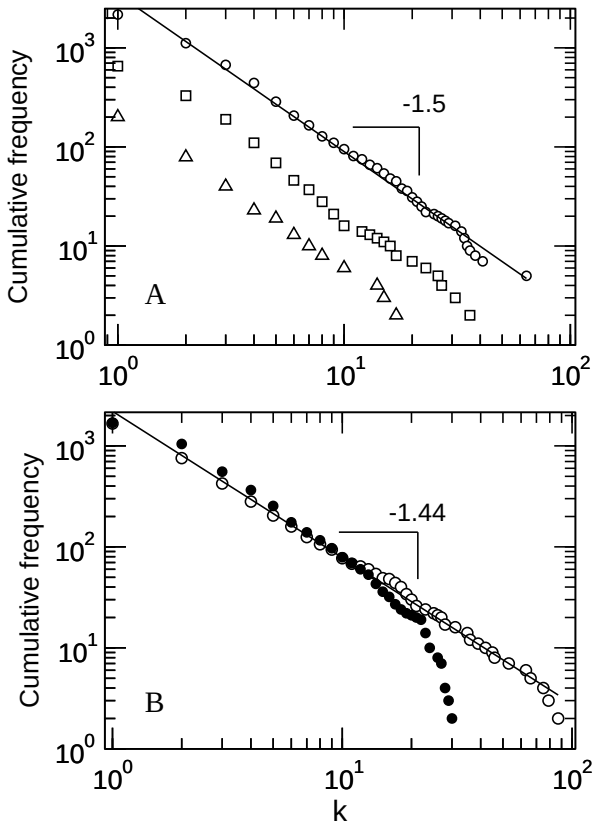

Fig. 4: (A) Cumulative degree distributions for several software graphs: eMule (N=129, triangles), Blender (N=495, squares) and CS (N=1488, circles). All distributions have an exponent about -2.5 in spite of the obvious differences in size and functionality. (B) Asymmetry of in-degree (open circles) and out-degree (filled circles) distributions for ProRally 2002. The in-degree distribution is the probability that a given component is reused by k_{in} other components. Conversely, the out-degree distribution is the probability that a component uses k_{out} other components. Note that out-degree distribution has sharp cutoff.

Their ubiquity seems to stem shared organizing principles [24].

SF networks are known to display some unexpected statistical features. In particular, looking at the moments of the degree distribution, i. e.

$$M_\mu = \int_1^\infty k^\mu P(k) dk$$

(with μ =1,2,...) and assuming that $P(k) \approx Ak^{-\gamma}$, it is easy to show that the average degree is well defined, leading to <k>=(γ-1)/(γ-2), whereas the higher moments are not, since they scale as

$$M_\mu = k^{\mu-\gamma+1}$$

and thus $M_\mu \to \infty$ for $\mu \geq 2$. Fluctuations are thus extremely important and have been shown to be the key for understanding a number of key features exhibited by SF architectures. This is the case for example of the spreading of computer viruses on the Internet [25].How do SF nets originate? There are a number of well-identified processes leading to SF structure. Most of them rely in a growing network displaying some rules of preferential attachment of new nodes [refs]. However, it has been suggested that a sparse SF network can actually result from an underlying

optimization process in which efficient communication at low cost is involved [26]. But the most interesting implications from SF architecture are related to their high robustness against random node failure, together with a high level of fragility when hubs fail [27]. In other words, information transfer keeps working in an efficient way when a randomly choosen node fails but tipically degrades when a highly connected component fails. Such observation has been shown to have immediate implications for reliable network architecture. Since system's sensitivity to component failure is a fundamental problem in any area of engineering, it is important to recognize how network topology will influence system's performance. The SF pattern, common to all studied graphs is an emergent property of software evolution: the overall architecture is not specified within the design principles and yet it seems to be the universal result of the evolution process. As a consequence of the local, distributed path of software construction, a modular, hierarchical architecture emerges. Together with it, and inevitably (unless top-down controls are introduced) the resulting system will be likely to include a high robustness against failure of most random errors but also a high fragility when some specific parts fail to behave properly.

As shown in figure 4 and in table II, all the systems are SF. The fact that all the systems analysed here display SF structure, in spite of the obvious differences in size, functionality and other features, indicates that strong constraints are at work during software evolution. In order to properly estimate the scaling exponent γ, we used the cumulative distribution $P_>(k)$, defined as follows:

$$P_>(k) = \sum_{k' > k} P(k')$$

(so if $P(k) \sim k^{-\gamma}$, then we have $P_>(k) \sim \int P(k')dk' \sim k^{-\gamma+1}$). A clear regularity is that the exponents obtained from the directed graphs differ from the undirected class diagram. Tipically, we observe $\gamma \sim 2.5$, with $\gamma_{in} < \gamma$ and $\gamma_{out} > \gamma$. In other words, if we look at the number of outgoing and incoming links, the resulting degree distributions are different. They are more heavily tailed for the in-degree and more rapidly decaying for the out degree. As noted by Chris Myers, nodes with high in-degree tipically result from broad reuse [16]. The reasons for such asymmetry are rooted precisely in the economization of development effort and related costs. We will return to this important question in the next section.

TABLE II
Exponents of Cumulative Degree Distributions

Dataset	*Degree*	*In-degree*	*Out-degree*
Mudsi	1.74 ±0.04	1.2 ±0.08	2 ±0.05
JDK-B	1.55 ±0.08	1.39 ±0.05	2.3 ±0.14
JDK-A	1.41 ±0.02	1.18 ±0.02	2.39 ±0.14
Prorally	1.72 ±0.03	1.44 ±0.02	1.88 ±0.1
Striker	1.7 ±0.04	1.54 ±0.03	1.73 ±0.06
gchempaint	1.63 ±0.31	1.11 ±0.35	1.41 ±012
4yp	1.54 ±0.09	1.3 ±0.05	1.59 ±0.18
Prospectus	1.67 ±0.09	1.13 ±0.09	1.92 ±0.27
eMule	1.58 ±0.03	1.51 ±0.07	1.42 ±0.08
Aime	1.43 ±0.05	1.3 ±0.04	1.48 ±0.07
Openvrml	1.34 ±0.06	0.94 ±0.05	1.59 ±0.23
gpdf	1.64 ±0.11	1.23 ±0.1	1.76 ±0.17
Bochs	1.37 ±0.08	1.17 ±0.09	1.64 ±0.2
Quanta	1.69 ±0.1	1.55 ±0.13	1.87 ±0.13
Fresco	1.66 ±0.09	1.14 ±0.1	1.76 ±0.19
Freetype	1.65 ±0.07	1.42 ±0.04	1.82 ±0.16
Yahoopops	1.67 ±0.05	1.46 ±0.06	1.69 ±0.05
Blender	1.64 ±0.04	1.36 ±0.05	2.04 ±0.09
M4	1.7 ±0.08	1.38 ±0.05	2.04 ±0.16
GTK	1.51 ±0.04	1.22 ±0.02	2.38 ±0.2
OIV	1.43 ±0.02	1.14 ±0.03	2.1 ±0.12
wxWindows	1.41 ±0.03	1.11 ±0.02	2.18 ±0.12
CS	1.58 ±0.02	1.22 ±0.03	1.96 ±0.09

VII. Software Costs

Software engineer designs in order to minimize the interactions between components while satisfying the hard constraints on development process [28]. It is important to choose well the partition that enables solving the software problem in the given amount of time. There are many different designs satisfying the requirements. To highlight the importance of economization consider the different choices when new functionality integrates into the system. One choice is to extend an existing software component because its relationships enable to implement the new functions. This solution could be bad in terms of clarity because the modified component may lose its intended previous meaning. The other option is to create a new component and adding some new relationships (links). As we can see, component creation could involve a cost that outweighs the benefits of having a clever diagram. Here, cost refers to the time (resources) required for building the software system. Is it possible to measure the quality of a given design?

A useful measure of cost must be able to capture the effort required to make a change. As commented previously in section II, most changes are consequence of defect repairing. Cumulative component dependency (CCD) is a topological measure presented in [29] that relates software quality with software maintenance costs. This measure recognizes the fact that not all software designs are equally adaptable to changes. Minimizing CCD for a given set of components is a design goal [29]. Defect location and test on designs arranged like acyclic tree-like hierarchies are more effective. Less time is spent on testing code because the implied dependency ordering between components (low development cost). Cycles are undesirable because they do not allow defining such an ordering. This measure of cost tells us that good designs must avoid cycles.

In order to compute the cumulative component cost we must consider the directed software graph $\Lambda=(V,E)$ (see fig. 1). For each node $v \in V$, define the reachability set S(v) as the set of nodes reachable from v (a node w is reachable from v if there is a (directed) path from v to w). This is the

set of nodes reached by standard depth-first search started at node v. It turns out that the cost of a node is the size of its reachability set (taking into account that every node is self-reachable):

$$CCD(v) = |S(v)|$$

This is a measure of the linking time (and storage) required for testing the component v in isolation [29]. In order to build a node it is required to build previously all the other components that the node relies on. Any single component modification necessarily implies re-linking of all related components. That is, high cost nodes will be more frequently affected by changes. The cost of the graph is simply the sum of all nodes' cost, which is equal to the size of the transitive closure T:

$$T = \sum_{v \in V} |S(v)|$$

Computing the size of transitive closure of a directed graph is a fundamental graph problem that has many applications in other fields like database systems and design of network routing algorithms. The transitive closure of a directed graph Λ is another directed graph where there is a link between nodes v and w if there is a path from v to w in the graph Λ. Warshall's algorithm is an efficient method of finding the transitive closure of a directed graph [30]. The size of the transitive closure is simply the number of links in the transitive closure graph. It is interesting to give the expressions for the size of the transitive closure of the complete directed graph Λ_c and the balanced binary tree Λ_t. In the complete graph, every node knows every other node and thus it is not possible to test components in isolation. The cost grows as fast as the system size squared:

$$T(\Lambda_c) = N^2$$

Hierarchical designs offer a great advantage because it is possible to do testing incrementally. A simple testing scheme will be to test first the leaf nodes (that is, nodes whose out degree is zero). Assuming that such tested nodes are safe (or reliable), we can progressively move towards higher levels until reaching the root node. The cost of testing the entire graph is proportional to the logarithm of the system size [29]:

$$T(\Lambda_t) = (N+1)(\log_2(N+1) - 1) + 1$$

In order to give an idea of the quality of the software design, it is useful to compute the normalized cumulative component dependency (NCCD), which is the ratio of CCD of graph containing N components to the CCD of a balanced binary tree of the same size.

Previous efforts have been concentrated in finding efficient algorithms for computing the size of transitive closure of a directed graph. In particular, there is a linear time algorithm for directed graphs with bounded out-degree [31]. Because their structure, such graphs have a cost which grows sub-quadratically with their size. This suggests that limiting the maximum out-degree will result in software that costs less. This might be the reason for the sharp cutoff observed in the software out-degree distribution (see fig. 4B). Moreover, economization imposes a very hard constraint in highly reused nodes (or nodes with high in-degree). From the point of view of cost, it is inconvenient to add an outgoing link to a highly reused node because the new link automatically rises up the cost of the entire system.

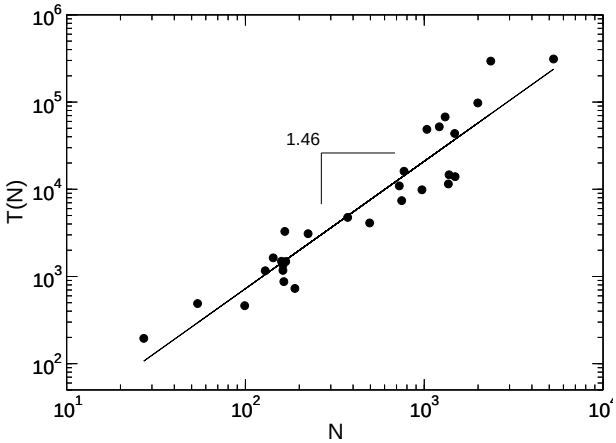

Fig. 5. Empirical relationship between software cost and size (see text for details).

For the software graphs analyzed here, we find that their cost scales with system size (see fig. 5):

$$T \sim N^{\eta}$$

where η =1.46 ± 0.09. That is, real software graphs have intermediate cost between the (ideal) balanced binary tree and the complete directed graph.

VIII. Software Evolution

It has been recognized the importance of historical contingency in explaining scale-free networks. In the model of Barabási-Albert (also known as preferential attachment or 'rich-gets-richer') a scale-free network with an exponent of minus three is progressively constructed by adding one node at every step [23]. This node connects to up *m* nodes selected probabilistically because of their importance within the network. The BA model assumes a node is important if it has a high degree. The model simplicity is its main attractiveness but unfortunately is not accurate for many real systems like software. The main disadvantage of the scale-free BA model is that it does not reproduce the high clustering already present in many systems.

We have studied in detail the evolution of the Pro Rally 2002 game system. Today it is common practice in software developer teams to use a backup system to avoid costly losses of source code. In a way, this means a 'fossil record' of the software evolution is available. From one of such databases we recovered a series of 176 intermediate class diagrams comprising the two years of development process. The first thing to notice is that ProRally 2002 grows linearly. At each time step, both number of nodes and number of links increase by a constant amount (give approximate rates). In spite of the transient changes displayed by different measures, the mean degree <k>, the mean shortest distance d and the mean clustering C tend to stabilization around some constant values.

In figure 6 three key evolving features of software architecture are shown. The first is <k>, which shows

roughly three phases (fig. 6A). The first third of the evolution takes place around an average degree $\langle k \rangle_I \sim 3.5$ - 4, followed by a linear increase until a new plateau is reached at $\langle k \rangle_{III} \sim 5$. The growth phase in degree is not matched by the two measures characterizing the small world property (fig. 6B-C). The average path length rapidly reaches a steady state with $d \cong 5$. This indicates that the global communication exhibited by the class diagram is early attained by the system. Something similar occurs with C, which experiences a rapid increase followed by a further increase at later stages. The small world pattern thus emerges rather soon and is maintained through the evolution. The number of links keeps changing towards larger values, thus indicating that the system experiences extensive rewiring without changing its basic architecture.

The fact that network diameter d is constant through most of the process has nontrivial consequences. As shown by Puniyani and Lukose, growing random networks under the constraint of constant diameter display SF architecture [32], with a scaling exponent γ between two and three. Specifically, they found that

$$P(k) \sim k^{3-\frac{\alpha}{\beta}}$$

when $\alpha \leq 1$ is the scaling exponent relating network size with the fluctuations in network connectivity:

$$N^{\alpha} = \frac{1}{< k >} \int^{k} k^2 P(k) dk$$

and β is the scaling exponent linking the cutoff k_c with size, i.e.:

$$k_c \sim N^{\beta}$$

Using our dataset, we obtain $\beta = 0.62 \pm 0.09$ and $\alpha = 0.42 \pm 0.08$, which gives a predicted scaling exponent $\gamma \cong 2.59$, to be compared with the average over all systems, $\langle \gamma \rangle = 2.57 \pm 0.07$ (computed from table II). The scaling law in the cutoff k_c (N) allows us to provide an analytic calculation of the scaling between L and N. Here we have

$$L = N \int_0^{\infty} kP(k) dk$$

$$\sim \frac{N}{k_c^{2-\gamma}}$$

$$\sim N (N^{\beta})^{\gamma-2} = N^{1+\beta(\gamma-2)} \sim N^{1.22}$$

in very good agreement with the exponent obtained in section III for the software graphs.

IX. DISCUSSION

The important point of our work is that different software architectures share the same universal topological features, that is, small-worldness and scale-free behavior. This is a very surprising result and raises several interesting questions about how we (as human designers) organize knowledge and express the inner workings of a very complex, and abstract machine. Surprisingly, natural networks share the same statistical features. Seems that there is a clear relationship between software evolvability and how components are embedded in the architecture.

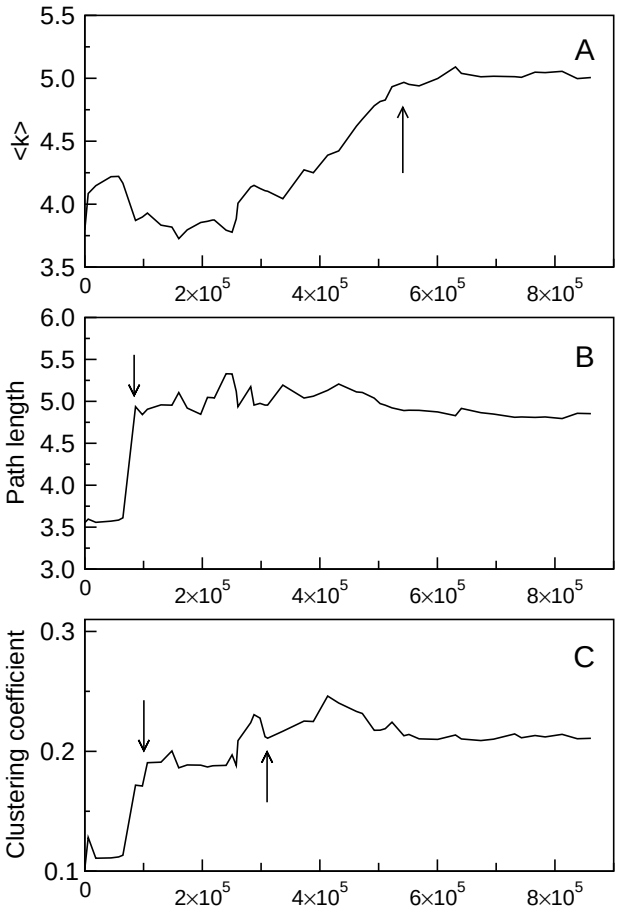

Fig. 6. Evolution of some quantities of interest (ProRally 2002) Time is displayed in minutes. (A) Evolution of mean degree. (B) Evolution of mean path length. (C) Evolution of clustering coefficient.

The observed scale-free, hierarchical structure displayed by software maps indicates that some basic principles of organization and evolution are at work. In this sense, our results might provide some solid grounds for a theory of software evolution. The analysis reported here reveals that, irrespective of the specific features of each system (and they notably differ in their functional meaning) they all exhibit the same global pattern of organization. Such pattern is not explicitly defined at the level of design rules of software development. Yet, as a consequence of a distributed process of evolution, scale-free structures emerge.

Recent studies on graph evolution through local rules reveal that scale-free patterns are very common [11] but not all evolution rules lead to hierarchical patterns or modularity. One result emerges from these studies with practical consequences in our context: highly heterogeneous graphs inevitably result from a distributed process of link addition, deletion and rewiring through the process of network growing. As a consequence, well-defined structures will be obtained involving a very narrow range of large-scale properties. The narrow range of clustering coefficients, the sparseness of the graphs and the similar exponents obtained for the link distributions tell us that, as it happens to occur with some biological structures, strong constraints operate on top of local rules of evolution [10]. Can natural systems provide some guide for understanding

software evolution and perhaps useful metaphors for a theory of software evolution?

The biological metaphor of evolution on a fitness landscape is very useful here. In its simplest form, a fitness landscape is defined as a single valued scalar function F(X) of the state (or configuration) X of a system [33] where the variable X is typically multidimensional. Usually the so called fitness function F(X) is something to be maximized. As the system explores the parameter space (by mutation or some other sort of change) a hill climbing takes place and higher fit points of the landscape are reached. The landscape can be more or less rugged and as a consequence a few or many peaks are available. In rugged landscapes, the system will eventually reach a local peak and no further change will be allowed. The presence of multiple local peaks tipically reflects the presence of conflicting constraints between different variables involved. This situation is known to occur in software evolution too: often, once a given level of complexity is reached, no further improvement is possible unless we start again from scratch.

In spite that the software engineers have a broader view of the system's state (no such a thing happens in the evolutionary context, where there is no a priori planning of the final function) it seems obvious that some common principles of biological evolution are at play here too. The most clear is the presence of tinkering in both natural and artificial systems. Francois Jacob made the original mention of tinkering within the context of biological evolution in 1977 [34]. Jacob pointed out that evolution does not operate as an engineer (which somehow foresees the product of his design) but instead as a tinkerer. On the one hand there is no preconceived plan and on the other evolution through natural selection uses everything at his disposal to produce workable structures. Additionally, each system at each level uses as ingredients some systems at the lower scale. As a consequence, new properties can arise at higher levels but new constraints also emerge. Constraints limit the further evolution and the system might get eventually trapped in some local peak of the fitness landscape.

It has been noticed several times that in order to advance the current state of software engineering a scientific theory is required [35]. Our measurement framework provides quantitative evaluation of software designs and some general invariants (like the scale-free behavior) which might be a first step towards a scientific theory of software development.

An interesting avenue for future exploration is to correlate dynamic and static measurements of software. For instance, consider a class A than uses another class B by calling the method `B::f()`. One can imagine measuring the 'strength' of the A→B link as the number of times the `B::f()` method was invoked during a typical execution. Moreover, it is already possible to assign weights not only to links but to single components too. In order to compute this weight it is possible to use the ability of most modern profilers of measuring the frequency of execution of components. Preliminary measurements show that component execution frequency also follows power-law distribution. This is consistent with the known empirical relation that programs spend 80/90 percent of their time executing about 10 percent of the code. Is there any correlation between the scale-free structure of software architecture and the frequency of execution of components?

APPENDIX

Class Diagram Reconstruction Procedure

In order to obtain the software graph from the C/C++ sources we restrict ourselves only to the information contained in the header files (i.e: source files with extension *.h and *.hpp). We have written a simple C/C++ parser for the header files that extracts the graph from the `class` and `struct` declarations (data types). Isolated procedure declarations are discarded.

Every class or struct identified is assigned to a single graph node. A data field (or attribute) belonging to a class or a struct is considered as a directed link between the owner class and the corresponding node of the referenced data type. It is important to note that simple data types like `int`, `float` or `char` (and other simple derivates) are not taken into account in this analysis. References to these data types are simply discarded for any subsequent consideration.

Class inheritance is also taken into account. If a class A inherits from another class B, then directed link A→B is added. In case of multiple inheritance, more than one link is allowed but we have found that in real systems there are few links coming from this kind of declaration.

There is also the question of considering data types referenced in class methods (functions). In the analysis presented by Myers [16] they were simply discarded. Here, we have taken into account some of the data types declared by class methods. If a method returns a complex data type, then the owner class links with the node corresponding to the returned data type. Creation involves a high degree of coupling between the generating class and its "offspring".

We simplify the reconstruction process by parsing all header files in two passes. In the first pass, all nodes are located so it is possible that a node references another unknown to this moment. These pending references will be solved in a second pass and thus yielding the (directed) software graph. The graph is post-processed and only the largest connected component (in the undirected sense) is selected for analysis.

ACKNOWLEDGMENT

We thank Pau Fernández, Jose M. Montoya, Ramón Ferrer i Cancho and Stuart Kauffman for sharing many thoughts and ideas about the nature of complex networks. We also thank helpful discussions with the software engineers Alex Rodriguez, Jose A. Andreu, Jose Paredes & Antoni Zamora from the ProRally 2002 team and the musicians at Opeth.

REFERENCES

[1] 1968 NATO Conference Report, Naur & Randell Eds, NATO Scientific Affairs Division, Brussels 39, Belgium.

[2] E. W. Dijkstra, "The Humble Programmer", *Comm. ACM*, vol. 15, no. 10 , pp. 859-866, 1972.

[3] E. W. Dijkstra, "The End of Computing Science?", *Comm. ACM*, vol. 44, no. 3, pp. 92, March 2001.

[4] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *Computer*, pp. 10-19, Apr. 1987.

[5] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", *Comm ACM*, vol. 15, no. 12, pp. 1053-1058, Dec. 1972.

[6] J. M. Bieman, B. Kang, "Measuring Design-Level Cohesion", *IEEE Trans. Soft. Eng.*, vol. 24, no. 2, pp. 111-123, Feb 1998.

[7] DeRemer, F. and Kron, H. H. "Programming-in-the-Large vs. Programming-in-the-Small", *IEEE Trans. Soft. Eng.* vol. 2, no. 2, pp. 80-86, June 1976.

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, Reading, MA, Addison-Wesley, 1994.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, A. Mockus, "Does Code Decay? Assesing the Evidence from Change Management Data", *IEEE Trans. Sofware. Eng.*, vol. 27, no. 1, pp. 1 – 12, Jan. 2001.

[10] R. V. Solé, R. Ferrer-Cancho, J. M. Montoya, S. Valverde, "Selection, Tinkering and Emergence in Complex Networks", *Complexity*, vol. 8, pp. 20-33, 2002.

[11] S.N. Dorogovtsev and J.F.F. Mendes, *Evolution of Networks: From Biological Nets to the Internet and the WWW*, Oxford, New York, 2003.

[12] S. Valverde, R. Ferrer-Cancho, R. V. Solé, "Scale-Free Networks from Optimal Design", *Europhys. Lett.* , vol. 60, no. 4, pp. 512-517, Nov. 2002.

[13] C. Alexander, *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, 1964.

[14] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.

[15] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*. New York, Addison-Wesley, 2000.

[16] C. R. Myers, "Software Systems as Complex Networks: the Emergent Structure of Software Collaboration Graphs", cond-mat/0305575, 2003.

[17] W. M. Turski, "Reference Model for Smooth Growth of Software Systems", *IEEE Trans. Software Eng.*, vol. 22, no. 8, Aug. 1996.

[18] P. Erdös and A. Rényi, "On the Evolution of Random Graphs", Pub. Math. Inst. Hungarian Acad. Sci., vol. 5, pp. 17-61, 1960.

[19] P. Bollobás, *Random Graphs.* Academic Press, London, 1985.

[20] D.J. Watts and S.H. Strogatz, "Collective Dynamics of Small-World Networks", *Nature*, vol. 393, no. 440, 1998.

[21] J. Guare, *Six Degrees of Separation: A Play*, Vintage Books, NewYork, 1990.

[22] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai and A.-L. Barabási, "Hierarchical Organization of Modularity in Metabolic Networks", *Science*, vol. 297, no. 30, Aug. 2002.

[23] R. Albert and A.-L. Barabási, "Statistical Mechanics of Complex Networks", *Reviews of Modern Physics*, vol. 74, no. 47, 2002.

[24] A.-L. Barabasi and E. Bonabeau, "Scale Free Networks", *Sci. Am.*, pp.60-69, May 2003.

[25] R. Pastor-Satorras and A. Vespignani, "Epidemic Spreading in Scale-Free Networks", *Phys. Rev. Letters*, vol. 86, 3200-3, 2001.

[26] R. Ferrer-Cancho and R. V. Solé, "Optimization in Complex Networks", *Statistical Physics in Complex Networks*, Lecture Notes in Physics, Springer, Berlin, 2003.

[27] R. Albert, H. Jeong and A.-L. Barabási, "Error and Attack Tolerance of Complex Networks", *Nature*, vol. 406, no. 27, pp. 378-381, July 2000.

[28] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1992.

[29] J. Lakos, *Large-Scale C++ Software Design*. Addison-Wesley, Reading, Massachusetts, 1996.

[30] S. Warshall, "A Theorem on Boolean Matrices", *J. Assoc. Comput. Mach.*, vol. 9, pp. 11-12, 1962.

[31] R. J. Lipton and J. F. Naughton, "Estimating the Size of Generalized Transitive Closures", *Proc. 15th Int. Conf. on Very Large Data Bases*, pp. 315-326, 1989.

[32] A. R. Puniyani and R. M. Lukose, "Growing Random Networks under Constraints", cond-mat/0107391, 2001.

[33] S. A. Kauffman, *The Origins of Order*, Oxford University Press, New York, 1993.

[34] F. Jacob, "Evolution as Tinkering", *Science*, vol. 106, pp. 1161-1166, 1976.

[35] M. M. Lehman and J. F. Ramil, "An Approach to a Theory of Software Evolution", *Proc. 4th Int. Workshop Principles of Software Evolution*, Vienna, Austria, pp. 70-74, 2001.

S. Valverde received the M.S. degree in Computer Science (1999) from the Polytechnic University of Catalonia, in Barcelona. His working experience has been mainly related to software interactive systems and computer graphics. He has worked as software engineer at R+D El Periódico de Catalunya and UbiSoft Entertainment. He is a PhD candidate under the supervision of Ricard V. Solé. The focus of his current work is to understand complex artificial networks from the Internet to complex software systems. He is member of the ACM.

R. V. Solé received the degree in Biology (1986) and in Physics (1988) from the University of Barcelona, and the Ph D degree in Physics from Polytechnic University of Catalonia. He is currently research professor at the Universitat Pompeu Fabra, in Barcelona, where he leads the Complex Systems Lab. He is also external professor at the Santa Fe Institute and senior member of the NASA-associate Astrobiology Center, in Madrid. His current research interests involve understanding the origins and evolution of complex networks in both natural and artificial systems.