Chapter 4

On the Nature of Design

Sergi Valverde and Ricard V. Solé
Complex Systems Lab
ICREA-Universitat Pompeu Fabra, Barcelona
svalverde@imim.es
ricard.sole@upf.edu

1 Introduction

At the beginning of the industrial revolution, an extraordinary event attracted the attention of scientist, philosophers and layman alike. It was so extraordinary in fact that even today we are fascinated by it and by the no less uncommon people who got involved. The subject of this story was an amazing machine, more precisely an automaton. Known as the Turk, it was a mechanical chess player, made of wood and dressed in a Turskish-like costume (see Fig. 5). It played chess with Napoleon, inspired Charles Babbage and moved the great Edgar Allan Poe to write a critical essay about the nature of the automaton [1].

Although mechanical automata were not new at the time the Turk appeard into the scene in 1770, it was certainly a far-sighted invention. From the available accounts of these times, it had to be a rather impressive rival. Kempelen's automaton was life-size, and was able to move its head and eyes and move the chessmen forward. It was also able to say a few words such as "Check".

The expectation and doubts raised by the Turk were the result of its life-like, intelligent behavior. The machinery inside the Turk did not look complicated enough to explain the virtually astronomical repertoire of movements observed. So to speak, the hardware was impressive but the software was missing. Previous automata achieved fame by displaying a given repertoire of mechanical actions that were repeated again and again with the same sequence. Vaucanson's duck, for example, imitated a real bird and was able to quack, flap its wings and even simulate digesting food. In spite of the complexity of these actions, the internal

mechanism was a clock-like system with wheels and levers. All these mechanisms and the wires connecting them with the different parts of the automaton were included inside a large pedestal. But the chess player faced a great challenge: to be able to play a game with an enormous potential combinatorics. How it could it be possible? Although Charles Babbage considered the possibility of building an intelligent machine after playing with the Turk in 1819, it was apparent to him that the automaton was probably hiding a human inside it. That was of course the case.

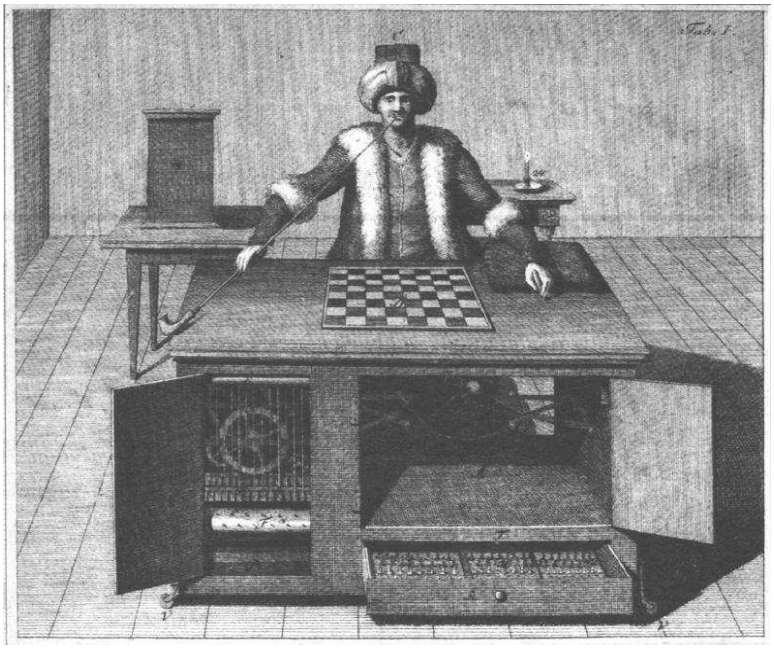

Figure 1: Von Kempelen's chess player automaton, the Turk. Here a front view is shown, with the cabinet doors open showing the internal mechanisms which were claimed to power the automaton's abilities. In reality, a man was actually hidden inside the cabinet and manipulated the automaton.

An interesting point here is how life-like (or "human") the Chess Player was. In a more general context, the Turk raises questions on the boundaries between life and the artificial. What features of living systems can be captured by man-made designs? Engineered structures seem to be closer to the physical than the biological world. But this might actually be a misleading conclusion. The key difference that distinguishes biology from physics is that biological systems perform computations. The origin of such difference stems from the role that information plays in the first, which is not shared by the second [2]: there is an evolutionary payoff placed on being able to predict the future. More complex organisms are better able to cope with environmental uncertainty because

they can compute and can also make calculations that determine the appropriate behavior using what they sense from the outside world. Such computing systems emerge through evolution as a consequence of different (non exclusive) mechanisms [3, 4, 5, 6, 7].

Perhaps the earliest exploration of a theoretical basis for life-like structures is Von Neumann's study of self-reproducing automata. While looking at the minimal, formal conditions required for a given system to replicate itself, Von Neumann found that these automata should include two key ingredients in their architecture: *hardware* and *software* [8]. As noted by several authors, there is a surprisingly good mapping between Von Neumann's finding and the actual structure of cellular organization. Although this work was formulated several years before Watson and Crick's discovery of DNA, it already presented a formal picture of *what should be expected* to be observed.

Computer science has been evolving over the last 50 years in many directions, but in some fundamental sense it has been frozen into a well-defined view of computing based on von Neumann's ideas of computers (see below). Although there was an early fascination in getting inspired by nature while thinking on how machines should compute, such initial fascination rapidly faded out. Powerful designs rapidly emerged and became real. Fast computers were built and biological metaphors became unnecessary.

When looking at the architecture of cells, three basic components can be properly identified (together with a membrane structure separating the inside from the outside):

- The genome, and the regulation pathways defined by interactions among genes;
- The proteome, defined by the set of proteins and their interactions; and
- The metabolome (or metabolic network) also under the control of proteins that operate as enzymes.

The last two components define the hardware of cells, while the first is the software. Roughly speaking, the instructions written in the DNA sequence (the genetic material) are executed provided that the appropriate hardware is present. Perhaps not surprisingly the jargon of molecular cell biology is full of terms suggesting that computations are taking place, such as *transcription*, *translation* or *genetic code* [9].

Technological design and evolution reveal a number of traits in common with natural evolution [3]. On the other hand, many patterns found in nature seem to result from a combination of optimal designs together with strong structural constraints [3][4][5]. Such similarities are made more apparent while looking at the overall pattern of interactions among components both in cells and artifacts [10].

Understanding the origins and implications of computation in biology as well as in technology requires understanding the system level behavior of both hardware and software. Although hardware has received great attention, software has

been less appreciated in spite of its fundamental relevance, complexity and plasticity. This chapter constitutes a very early attempt to explore the main features of the architecture and functional organization of software, from the microscopic to the macroscopic level. In particular, we will report different network patterns observed in software structures. By understanding how these patterns originate, we might be able to provide tentative answers to some fundamental questions, such as:

1. Are there constraints to optimality in technological designs?
2. What type of emergent patterns result from engineering?
3. Is there tinkering at some level in the organization of artifacts?
4. Are emergent patterns similar to those observed in natural structures?
5. Are there fundamental differences in the global structures observed in natural and artificial structures?
6. Can we get inspiration from biological patterns in order to obtain new types of designs?

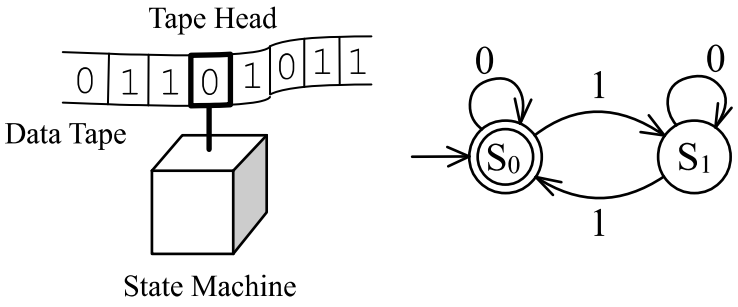

Figure 2: Schematic representation of a state machine (left). The state transition diagram for a simple finite state machine that accepts binary words with an even number of ones (right). Accepting states are depicted by double circles (see text).

2 Computing Machines

In order to start our exploration, we need to approach the problem in a way that is largely independent of specific computing machine features. In this context, computation theory [14] allows us to explore key features of complex systems by integrating architecture and function at different levels. Any computing device can be modelled with an abstract state machine. A state machine consists of a (possibly infinite) data tape of symbol cells and a computing device (see Fig. 2). The computing device handles the data tape according to a predefined set of

rules (or transition functions). The behavior of this machine can be described graphically by means of a state transition graph, where nodes depict machine states and directed links represent the possible transitions between states. At every step, a transition rule is selected depending on the current device state and the symbol read by the tape head. The rule instructs the device if a new symbol must be written on the tape, what is the next state and the direction the data tape shifts (i.e: move to the left or to the right).

A diversity of computing models can be obtained by disabling certain features of the most powerful model of computation, the Turing machine. This computer can write symbols in the data tape and also shifts the tape to the right or to the left, as desired. It can be shown that the Turing machine is powerful enough to simulate any computer [24]. On the other hand, the simplest model of computation is the finite state machine (FSM). The finite automaton has a read-only and unidirectional data tape. Another limitation of this machine is the finiteness of input words placed in its tape. This type of machine is only able to emulate a very constrained family of computers. In spite of these limitations, this class of machines really deserves some analysis because many dynamical features of cell biology, such as the cell cycle, can be described in terms of the FSM (see Fig. 3).

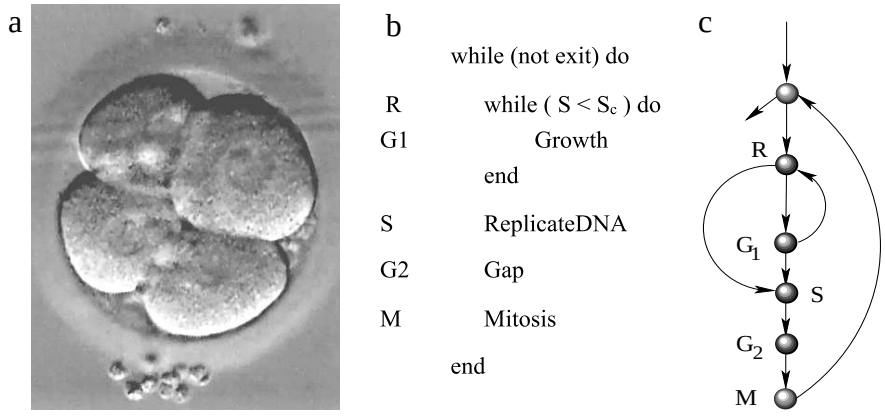

Figure 3: Many processes taking place inside living cells can be understood in terms of a computation. Some particular situations can be easily mapped into a discrete, finite state machine (see text). An example is the cell cycle. In (a) a micrograph of a dividing mammalian cell is shown. In (b) the corresponding basic algorithm for the cell cycle is given and the associated finite automaton is displayed in (c).

We can illustrate the inner workings of the FSM with an example. Fig. 2 shows the state transition diagram for a simple finite automaton. This simple computing device is intended to detect if the input word has an even number of ones. To perform this function, only two states are required. Every link is labelled with a symbol used to match the next state transition. Initially, the machine is configured in the starting state, which is denoted by a pointing arrow

(i.e: the state S_0 in the above figure). The tape is loaded with an input word (say "011") and the head is pointed to the first symbol (that is, "0"). The computing device performs the transition $\langle S_0, 0\rangle \rightarrow S_0$ and the tape head is shifted one position to the next symbol. In the following transition, the machine reads "1" and the new state becomes S_1. Finally, the symbol "1" is read and the state changed again to S_0. There are no more symbols to read so the machine checks if the current state is an ending state. In this case the word is accepted but not every word presented to the machine leaves the machine at an ending state (i.e: try "111").

In order to recognize more complex words it is necessary to extend the finite state machine capabilities. For instance, no finite state machine is able to accept words having an arbitrary number n of ones and zeros like 0^n1^n. The machine should be able to process a potentially infinite number of states. Counting requires the device to remember the number of symbols past read by using a memory storage, which is precisely the ability possessed by the Turing machine. The physical realization of the Turing machine is the Von Neumann architecture.

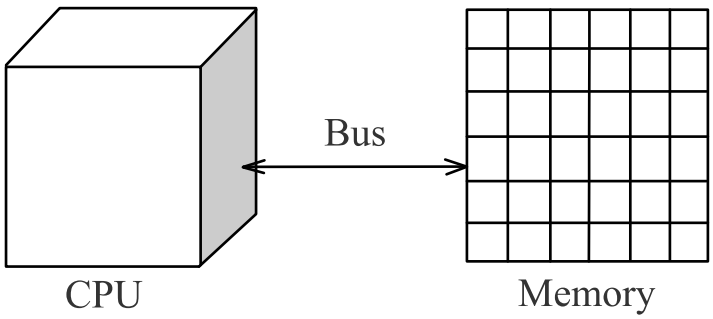

Figure 4: The Von Neumann architecture has three different parts: the CPU, the memory and the bus connecting both (see text for detailed description). The program is stored in the memory along with data, parameters and temporary calculations. The CPU traverses the memory recognizing program instructions and performing their associated actions. A little memory (registers) is included in the CPU and used during normal operation like for locating the current instruction.

The Von Neumann computer consists of three differentiated components: a central processing unit or CPU, a data store (or memory) and a wires connecting both components (or bus) (see Fig. 4). The CPU is a state machine which is able to recognize a number of special words (also called "program instructions"). The memory is a finite grid of cells analogous to the data tape used by the Turing machine. But unlike the Turing machine, the CPU is capable of direct access to any particular memory cell simply by referencing its position in the data store (or "memory address").

3 The Memory Stored Program

The signature of the universal computer is the memory stored program [24]. Any complex system requires this component in order to properly react and adapt to a changing environment. The program is a sequence of instructions that describe the computer's behavior. The CPU scans this sequence and activates different actions by accepted instructions. It can be shown that a very small instruction repertoire is enough to implement any program: arithmetic operations between two memory locations, store and/or retrieval of memory cells, and branching instructions. Complex behavior is achieved by the interpretation of the stored program, which yields a particular interleaving of calculations and memory accesses.

Assignment is the key instruction of the Von Neumann computer. The assignment stores a word or a number (often the value returned by evaluating a numeric expression) in a given memory cell. This places a very important restriction because the computer is only able to handle a single word-at-a-time [15]. Moreover, this computation model requires large amounts of data traffic exchanges through the bus in order to do useful work. The situation is worsened because a large fraction of traffic is wasted for sending memory addresses, that is, information for locating the required data. Because of the large volume of words exchanged and the word-at-a-time limitation, the bus rapidly becomes the bottleneck of the computation. Programming the Von Neumann computer means planning and specifying the enormous traffic of words through the Von Neumann bottleneck [15]. In order to minimize traffic exchanges and partly reduce performance problems, memory access patterns should be planned with care. In this context, the ordering of instructions is a key factor.

The sequence of instruction activation can be arbitrarily specified by inserting some special branching instructions that indicate to the CPU the address of the next instruction to be executed. The CPU stores the address of the current program instruction in a special cell called "program counter" register (PC). When the current operation finishes, the PC is automatically incremented in order to point to the following instruction. Branching instructions can change the state of the program counter in several ways and are very important in defining program behavior. For instance, there are unconditional branching instructions that allow the CPU to move to a distant location after (or before) the current instruction. Sometimes the jump is executed depending on the success of some arithmetic test or condition. Loops (executing the same instructions several times) can be implemented by combining unconditional and conditional instructions.

3.1 Flow Graphs

Program dynamics is defined at the interplay between memory contents and the instruction branching process. In this context, graphs are a useful tool for expressing interaction between different program parts. As static structures, they provide the skeleton on top of which function takes place. Still, this flow

graph is an incomplete characterization of program complexity because (besides other reasons) the interaction between instructions and memory cells is not depicted. However, as will be shown below, such static structures can be highly constrained in terms of the possible range of graphs that can be found for a given purpose. The flow graph describes the instruction processing order, encoding all possible branchings between program instructions[13]. Every node $v \in V$ in the flow graph (also named "basic block") represents a continuous sequence of instructions. The last instruction of a node is always a branching instruction or decision point. Directed links $(v, u) \in E$ signal the transferring of control from the last instruction of the source node v to the first instruction at the destination node u. Execution flows unidirectionally from the entry to the exit node, which are two special nodes present in any flow graph.

We call *in-degree* the number of links entering a node and *out-degree* the number of links exiting a node. Nodes with out-degree two denote conditional branching. With conditional branching, the transfer of flow depends on the evaluation of a conditional clause (link taken/not taken). On the other hand, unconditional branching is represented by nodes with out-degree one. Additional performance information may be attached to both nodes and links. For example, useful performance profiles are frequency of visits to a node or the frequency of traversing a link.

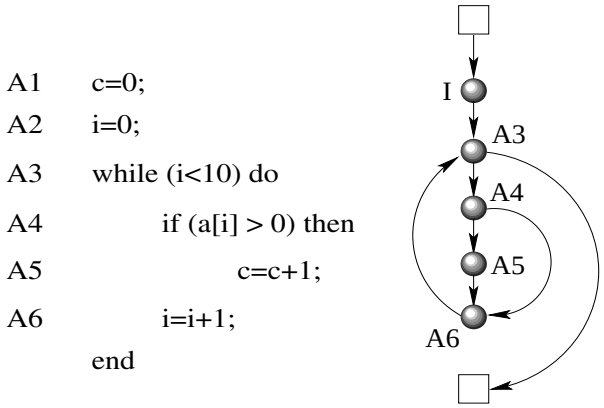

Figure 5: A simple program and its flow graph. Entry and exit points are denoted by empty boxes. Several instructions (A1,A2) are grouped within the initialization node (I). The main feature of the control flow graph is a loop, which is expressed with the back link (A6,A3). The looping condition is tested at node A3. When $i \geq 10$ the entire loop is skipped. At this moment, variable c equals the number of non-zero entries in vector a_j. Note also the branching point at A4 where the link (A4, A5) or the link (A4, A6) is chosen depending on the value of a memory cell. A multiplicity of different programs can be mapped onto the same control flow graph.

3.2 Program Predictability

Designers must pay attention to the evaluation order of instructions, that is, to the structure of the flow graph. An innocent permutation of a random pair of program instructions may yield very different program semantics. For example, let us swap instruction A6 and A1 in Fig. 5. We obtain an unresponsive program that never finishes and gets caught in an infinite loop. Unfortunately, it was shown by Turing that no automatic procedure is able to detect if a program stops for any given input configuration [24]. This so-called halting problem is deeply related to the inability to predict the future behavior of a computer program. In spite of the apparent simplicity of some programs, it turns out that we *cannot* predict what microscopic states result from the composition of simple instructions.

Any useful artificial object must be predictable. Software engineering should not be an exception to this rule. Take planetary missions for example. This is the kind of system that requires autonomy and strong tolerance under highly stressing environments. In the current state of art, reasonably predictable software is obtained by doing a lot of tests in many different scenarios. Unfortunately, the engineers can not plan in advance every situation faced by the software controlling the robot deployed on an unknown planet.

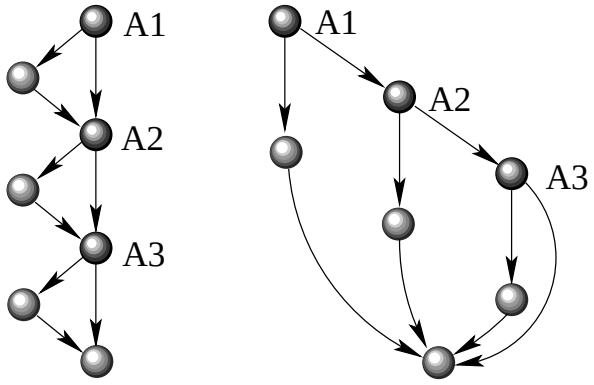

Figure 6: Two control flow graphs with the same number of predicates and branches, but very different number of (acyclic) paths. The control flow graph on the left defines 2^3 possible paths to be compared with the 4 possible paths of the right control flow graph. In this sense, the left directed graph is less predictable than the right one. Basically, this is reflected in the in-degree of some nodes on the left. Those nodes will be the crossroad of more than one path and thus increasing the uncertainty of the expected program behavior.

Predictability is also deeply linked to performance. For example, modern computers exploit regularities found in programs to yield better performance. A good example is provided by data and instruction caching. In order to avoid slow memory accesses, frequently referenced memory portions are copied into

a fast memory (or cache). This scheme results in speed-up only if program execution displays a certain predictability, that is, if the processor accesses the same memory region more than once.

Dynamic software behavior can be understood in terms of a walking through the flow graph. However, even the simplest flow graphs display an enormous number of potential execution paths. Walk length is theoretically unbounded because it is possible to visit some links many times (such as the link (A6,A3) in Fig. 5). We can also restrict the discussion to acyclic walks, also called paths. A path is a finite sequence of links $(u_1, u_2), (u_2, u_3), (u_3, u_4), ..., (u_{n-1}, u_n)$ in the flow graph where the destination of each link equals the source of the following link. No single link is reported more than once. The number of links in the path is the path length.

Unfortunately, flow graphs of large software systems still have an enormous number of potential paths. For instance, the popular application Microsoft(R) Word contains more than 2^{64} potential paths [23]. Again, the serious limitations deduced from the halting problem prevents us from differentiating between potential paths and actually visited paths from the static program description. In fact, this is equivalent to the problem of determining if a related program halts or not, which we know to be undecidable in general [24].

Surprisingly, the statistical analysis of real flow graph indicates that some programs are more predictable than others (see Fig. 6). Empirical studies have revealed that programs as a whole traverse a tiny fraction of the millions of potential program paths. Software performance profiles typically reveal that 90% of execution time is spent in a very small number of so-called "hot paths". This subset of traversed paths is largely independent of parameter variability, suggesting that software dynamics is not a purely driven process. In addition, real programs display a non-negligible amount of local correlation. Both global and local empirical regularities have been exploited in modern computer hardware in order to predict the next instruction to be executed by storing past branching outcomes (a mechanism analogous to data caching). In addition, these regularities enable the programmer to focus on a few program paths. This offers advantages when detecting performance bottlenecks and/or computation errors.

4 Programming and Separation of Concerns

Branching allows the reuse of instructions without the need for code duplication. The branching instruction allow us to partition the program into disjoint code pieces that alternate execution flow. Such a simple technique minimizes the number of program instructions and saves scarce memory resources, an important constraint in old computers. Beyond performance requirements, there is also a more important reason for structuring the program. The partition promotes the view that different functionalities must be provided by different program components [11]. Ideally, the mapping between functionality and components should be one-to-one but it was realized early on that, for complex programs, it is really difficult to make a clear division of labor.

Now imagine that our program achieves clear separation of responsibilities. In this case, the program state can be partitioned into disjoint pieces, each with a well-defined function. Then the whole state transition function can be expressed as the concatenation of simpler state transition functions. This program structure extends to memory organization because each piece computes a well-defined part of the global program state, without overlaps. In this ordered system, the whole coincides with the sum of its parts. Unfortunately, real software practices show us that the above clear separation is not an easy objective to reach. We find it very difficult to decompose the global program state in a linear combination of simple pieces. Instead, global program behavior is often defined by the interaction between more than one component (i.e, one memory cell accessed by two distant program instructions). In this case, interaction involves complex temporal correlations. Because of the constraints, our systems tend to exhibit complex dynamics that are more than the sum of the parts.

4.1 Object-Oriented Programming

Another important factor influencing programming is the language used for expressing the program. Using the same reduced set of machine instructions turns out to be very inconvenient for human designers. Different artificial languages have been considered for this task. The requirements imposed by a programming language are numerous and, to a certain extent, contradictory. Fortunately, the listener (the computer) is so constrained that ambiguity must be removed from the artificial language. This greatly simplifies the syntax of programming languages.

A broad characterization splits the world of programming languages into two big groups: declarative and procedural. When using the former type of language, the program constitutes a formal specification of what is wanted to be computed. Declarative languages (such as Prolog) only tell the computer what is desired and not how to achieve it. Conversely, a program written with a procedural language (like C, C++ or Basic) is a step-by-step detailed recipe of how to perform the computation. All previously presented code samples are instances of programs written in a procedural language. They are detailed plans that, when interpreted by the computer, yield the desired behavior. Declarative languages are very desirable from the user point of view but suffer from severe performance problems.

Modern programming practices (i.e: object-oriented languages like C++ or Java) are procedural and extensible. These languages are so powerful that they enable the programmer to create new software entities that represent real-world concepts. At this level, the program is understood in terms of high-level processes that manipulate abstract entities. The programmer looks at the domain of software application trying to localize the relevant entities and the relationships between them. For example, if a business application deals with customers and selling orders, the programmer should explicitly define the 'customer' and 'order' entities as part of the program description. In this example, 'customer'

and 'order' are related to each other because a particular order is placed by a given customer. In addition, attributes and processes are attached to entities. For example, with every customer the program stores her name and a process that enables the customer to issue a new order.

The complexity associated to software requires an evolutionary approach. The number of requirements is so large that we cannot develop the full application in a single step. In addition, some requirements are just unknown when we start to develop the system. This iterative process requires the programmer to switch between local and the global views of the software system. The programmer tends to focus her efforts on a single part of the complete design, which later is integrated with the whole system. Moreover, several developers can work on different parts simultaneously. Unfortunately, experience tells us that one cannot simply add more and more human developers and expect that the development process will be greatly shortened. As the number of programmers increases, so do the chances of unwanted interaction. There will be conflicting design decisions among programmers, which may result in project delays and overruns. Soon, the cost of communication outweighs the benefits of having many programmers working in parallel. In this context, an adequate global architecture of the software system (i.e., the set of components and their relationships) can be of great help.

4.2 Class Structure

Every advantage offered by object-oriented programming is based on the concept of class. The class recognizes that software development must be incrementally performed. When a new functionality must be added to the system, the engineer addresses a little computing device encoded by a class. The class machinery is nothing more than a grouping of variables and the code operating on them. The class variables are also known as attributes and each block of instructions is called a method. In order to avoid redundancy, the class might be allowed to cross its boundaries by accessing attributes and methods from other classes. The entire system is viewed like a network of interacting and simpler computing devices.

Inside the class, method interaction can be direct and indirect. The former type of interaction takes place when a method transfers the execution flow to another method, like directed edges in the flow graph. An indirect (but somewhat stronger) method interaction is through shared variables (like the methods f and g in Fig. 7). The only way that a method can alter the behavior of another method is by exchanging a variable. The execution path in a method depends on the values of accessed variables.

The notion of class cohesion is one of the most important object-oriented features. Good class design displays strong cohesion, that is, "a class should not be a collection of unrelated members, but all the members of a class should work together to provide some behaviors of the corresponding objects" [27]. Designs with strong class cohesion are believed to be more maintainable and reusable.

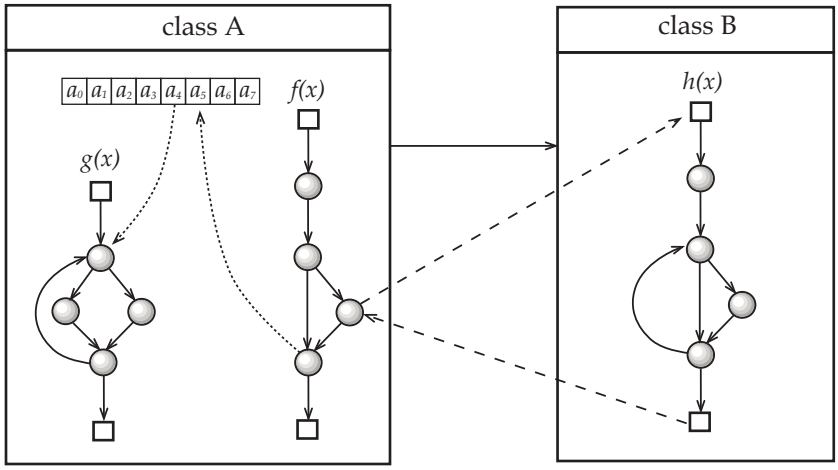

Figure 7: Mixed graphical representation for a simple software system consisting of two classes (also components) A and B. The diagram shows the following elements. Class A defines a single attribute (a vector of eight elements $a_0, a_1, a_2, ..a_7$) and two methods f and g. Every method is described by its control flow graph. Methods f and g indirectly interact by means of the data vector. The class B encapsulates the method h. Eventually, f transfers the control to h. The dashed line indicates that control flow is crossing class boundaries.

When the class methods are loosely interconnected, this is a sign of poor design and the class must split into several classes. Conversely, a class with strong cohesion will be difficult to split into isolated parts [28].

In this case, methods are closely related to each other by shared attributes. A bipartite graph representation is well-suited for measuring class cohesion. The bipartite graph $G = (F, V, E)$ consists of two disjoint sets F and V of nodes representing methods and attributes, respectively. Only interaction between two nodes of unlike sets is displayed. An edge belongs to the graph $\{f, v\} \in E$ if the method $f \in F$ references the variable $v \in V$. Two methods f and g will be indirectly related only if they share the same variable, that is, only if the $\{f, v\} \in E$ and $\{g, v\} \in E$. This bipartite graph is called an attribute-method reference graph. For instance, the attribute-method reference graph for class A in Fig. 7 consists of only three nodes: one for attribute a_i and another two for the methods f and g. The graph has only two edges connecting the attribute with the two methods. The attribute-method reference graph for class B in the same figure will have only a single node for the method h.

Once the reference graph is defined, the cohesion is measured as a function of the fraction of methods that should be removed in order to disconnect it [28]. Fig. 8 illustrates the notion of cohesion captured by this measurement.

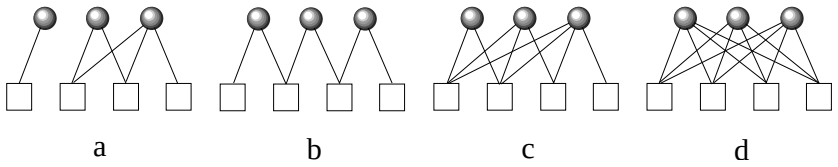

Figure 8: From left to right, attribute-method reference graphs for several classes displaying increasing levels of cohesion (see text). Attributes are displayed with circles and methods with boxes. Note that every box encloses a control flow graph. (a) shows an already disconnected class with weak cohesion. This is a symptom of poor design suggesting that two unrelated functionalities were enclosed within a single class. A better design will split the class in two different classes. Complete bipartite graph (d) always displays very strong cohesion. All methods must be removed in order to disconnect the graph. (Adapted from [28]).

4.3 Class Diagrams

Complex software systems rarely consist of a single class. The typical software system performs several functionalities distributed (more or less) evenly among a collection of interrelated classes. Software engineers are aware of this and they explicitly depict this large-scale organization through class diagrams. For this purpose, graphical languages like UML have been devised to communicate software designs in a standard way[36]. A simple UML class diagram is displayed in Fig. 9.

Class diagrams represent a number of interesting features about software designs. A quick look at this diagram gives a global idea of the internal software structure. The class diagram is an abstraction of the domain of software actuation, the entities and the nature of their relationships (i.e: in the previous sample commercial application, its class diagram should reflect the interaction between customers and orders). In order to introduce new software functionalities or to fix unwanted software behavior (also known as bug fixing), programmers navigate the information space defined by the class diagram. It is believed that some class diagrams enable fast identification of the relevant software pieces that must be changed or modified, thus reducing the total amount of effort spent by the programmer to accomplish her task. Can we identify and measure common structural patterns of class graphs? How is software quality reflected in its structure?

An early answer to the previous questions was given in their seminal book by Gamma, Helm, Johnson and Vlissides [29]. In their book, the authors proposed to assess the quality of an object-oriented system by looking at the patterns of collaborations between classes. It turns out that for some particular design problems there is some preferred solution which is more frequently selected among other candidate solutions. [29] presents a full catalogue of common solutions (or design patterns) observed in object-oriented programming. Every single design pattern has a name and is described by its intent, motivation and structure.

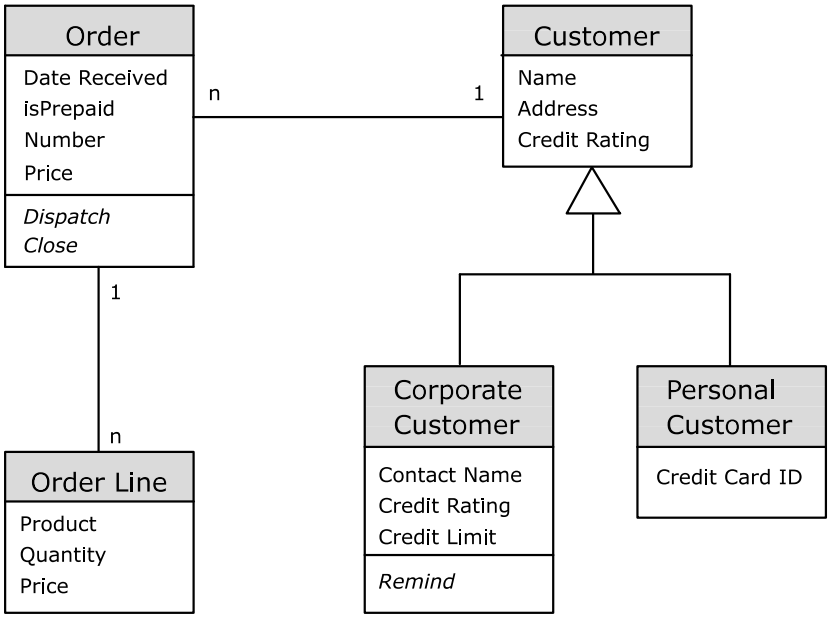

Figure 9: A simple class diagram for a commercial software application, in UML notation. The diagram shows five classes: Customer, Corporate Customer, Personal Customer, Order and Order Line. Every class is divided into three sections: name (shaded), attributes and methods (in cursive). Classes might relate to each other in three ways: composition, inheritance and use. These relationships are denoted by decorated links connecting two classes. For instance, the fact that the one customer can place more than one order is represented by a single relationship between 'customer' and 'order'. The numbers at the end-points of the link are the multiplicity of the relationship, telling how many objects will participate in the relationship. In the example, the customer is related to 'n' orders, but every order is only related to a single customer. Another typical relationship in UML diagrams is inheritance. This applies when two classes are similar but have different features. In the figure, both corporate customer and personal customer are related to the customer class by an inheritance relationship, indicating that they are able to place orders but in different ways.

In addition, the book classifies patterns in several families depending on their purpose (what a pattern does) and their scope (specifies whether the pattern applies primarily to classes or to objects). Are these patterns the signature of universal laws followed by high-quality software structures?

5 The Small World of Software Architecture

It is clear that the so-called "design patterns" approach is a qualitative catalogue of software knowledge. Here, we propose a new approach to document software

knowledge, which is based on the quantitative study of structural patterns in object-oriented systems. The first requirement of this new approach is to represent software structure with a network. The software graph is defined by a pair $\Omega_s = (W_s, E_s)$, where $W_s = \{s_i\}, (i = 1, ..., N)$ is the set of $N = |\Omega|$ classes and $E_s = \{\{s_i, s_j\}\}$ is the set of edges/connections between classes. The *adjacency matrix* ξ_{ij} indicates that a static interaction exists between classes $s_i, s_j \in \Omega_s$ ($\xi_{ij} = 1$) or that the interaction is absent ($\xi_{ij} = 0$). Nodes in the software graph are black boxes hiding internal class complexity. Similarly, links in the class graph hide the specific meaning of an underlying static collaboration between classes. There are two ways to recover the software graph: (1) from the class diagram described in UML or (2) from the source code itself.

When there is no explicit UML class diagram available, the analysis of source code is the best method for recovering the software graph. In [18] we have described a simple algorithm that recovers the software graph from C++ or Java source code. Automatic documentation tools implement similar algorithms [22]. The reconstruction process is implemented by a finite state machine, which looks for class definitions by finding all instances of the keyword "class" in the source code. The analysis of the class declaration body provides the links connecting classes. In this case, we look for the so-called "inheritance" and "uses" relationships found within the class. Every time the parsing process detects a class attribute, an edge is set from the owner class to the referenced class. Our definition of the software graph does not make any distinction between different types of static collaborations, which are always represented with a plain link. The reason for not attaching semantic information to nodes and links is that here we are only interested in modelling and characterizing structural patterns. Fig. 10 shows a software graph recovered from a real software application.

We define the average path length l as $l = \langle l_{min}(i, j) \rangle$ over all pairs $s_i, s_j \in \Omega_s$, where $l_{min}(i, j)$ indicates the length of the shortest path between two nodes. The clustering coefficient is defined as the probability that two classes that are neighbors of a given class are neighbors of each other. Poissonian graphs with an average degree $\bar{k}$ are such that $C \approx \bar{k}/N$ and the path length follows:

$$l \approx \frac{\log N}{\log(\bar{k})} \tag{1}$$

C is easily defined from the adjacency matrix, and is given by:

$$C = \left\langle \frac{2}{k_i(k_i - 1)} \sum_{j=1}^{N} \xi_{ij} \left[\sum_{k \in \Gamma_i} \xi_{jk} \right] \right\rangle_{\Omega_s} \tag{2}$$

This provides a measure of the average fraction of pairs of neighbors of a node that are also neighbors of each other. In a remarkable paper [17], Watts and Strogatz observed that many social, biological and technological networks, while very different in purpose and nature, share a number of common traits. All these systems are instances of what is known as small-world. Small-world

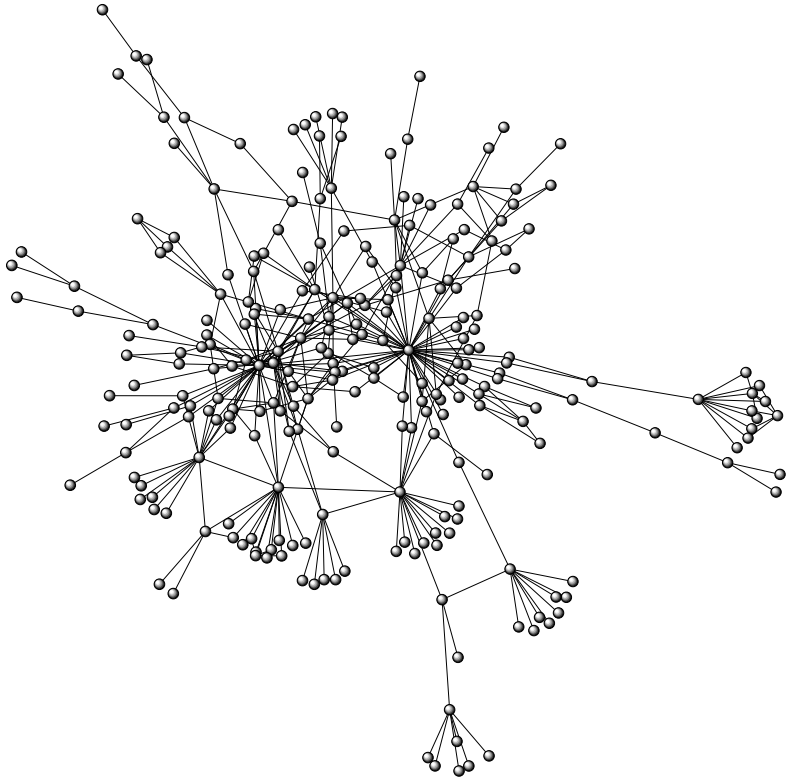

Figure 10: The largest connected component of the software graph Ω_s reconstructed from the source code of the 3D tool Aztec (http://aztec.sourceforge.net).

networks displays high clustering C, that is, nodes are connected in local neighborhoods. The surprising thing about small-world networks is that, in spite of the limited scope of nodes, the average path length l is very low. That is, any node is reachable within a small number of hops. This is achieved by means of a small number of key links or "shortcuts" that act like bridges connecting distant network regions.

We have analyzed the class diagrams for 29 different software systems (see [18] for a detailed analysis). These diagrams are examples of highly optimized structures, where design principles call for diagram comprehensibility, grouping components into modules, flexibility and reusability (i.e. avoiding the same task to be performed by different components). Although the entire plan is controlled by software engineers, no design principle explicitly introduces small-worldness. The resulting software graphs, however, turn out to be small worlds (see Fig. 11).

The small-world structure has important effects on the resulting network dynamics. A small-world communication network like the Internet propagates

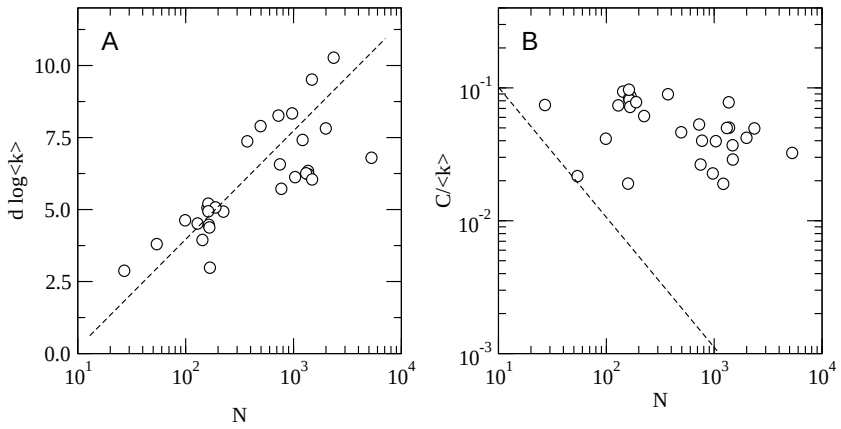

Figure 11: Average path length against network size for 29 different software systems. (a) Normalized distance grows with the logarithm of the number of classes, as expected in small world networks (see text). (b) Normalized clustering strongly departs from the predicted relation followed by random graphs (dashed line).

messages very quickly because of the low average path length. Moreover, its clustered nature makes the network very resilient to the loss of single elements. That is, there is enough redundancy in the number of paths connecting two nodes because their neighborhood is densely connected. Studies of synchronization in networks has also shown how small-world properties can be exploited in order to reach a globally synchronized state in a decentralized and robust manner [19]. This phenomenon is deeply related to computation. Indeed, we have determined that a large number of software applications are small-worlds. It might be that our software graphs arrange in small-world settings for similar reasons. But, how does the system reach the small-world architecture? How is the small-world exploited by computation processes?

5.1 A Simple Explanation for the Small World

The microscopic software structure is captured by the relationship between classes and methods (see Fig. 8), which accepts a bipartite graph $G = (W_s, W_m, D)$ representation (also known as class-method reference graph). The set W_s of classes and the set W_m or methods are disjoint, that is, $D = \{\{s_i, m_j\}\}$ where $s_i \in W_s$ and $m_j \in W_m$. The adjacency matrix ψ_{ij} for the bipartite graph encodes these connections:

$$\psi_{ij} = \begin{cases} 1 & \{s_i, m_j\} \in D \\ 0 & otherwise \end{cases} \tag{3}$$

This is an $N \times M$ binary matrix where $N = |W_s|$ (the number of classes) and $M = |W_m|$ (the number of methods). Relationships between nodes of the same kind can be recovered by means of one-mode projection of the bipartite graph

G (see Fig. 12). The projections yield two one-mode graphs $G_s = (W_s, D_s)$ and $G_m = (W_m, D_m)$. The adjacency matrix ψ^s for the graph G_s is related to the adjacency matrix ψ by

$$\psi^s_{ij} = \sum_k \psi_{ik}\psi_{jk} \tag{4}$$

and a similar relation holds between the adjacency matrix ψ^m for the graph G_m and ψ,

$$\psi^m_{ij} = \sum_k \psi_{ki}\psi_{kj} \tag{5}$$

Interestingly, it can be shown that projections are not random graphs even if the links between classes and methods are chosen at random. There are two important constraints affecting the projected graphs.

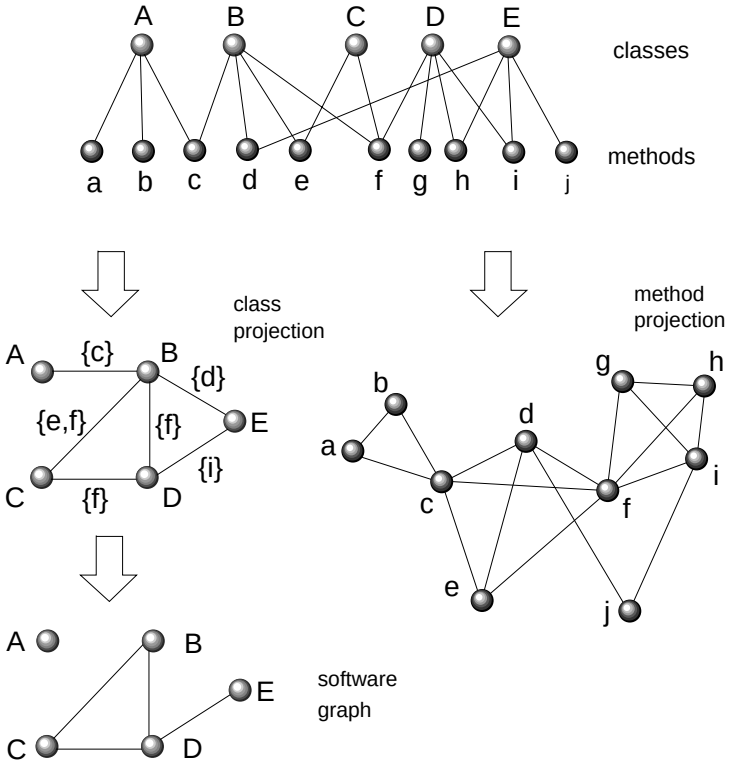

Figure 12: The reference graph relating classes and methods (top graph) can be projected in two one-mode graphs (middle graphs). The software graph is a subgraph of the class projection. Random bipartite graphs yield highly clustered one-mode graphs for free, that is, the software graph is constrained to follow certain non-random topological properties (see text).

The bipartite structure induces high clustering and low average path length. Let us consider the method projection G_m. In this projection, two methods m_i and m_j will be related if they both access the same class s_k. Projection predicts that all methods owned by a class will be related to each other, thus yielding highly clustered G_m graph (even if the bipartite graph is sparse) (figure 12 right). In terms of software engineering practices, the above means that classes tend to display strong cohesion (see figure 12 (c) and (d)). Actually, this hypothesis is in agreement with empirical studies of object-oriented software[28]. It can be shown that the projection of a random bipartite graph is always a small-world. Newman et al. [16] derived the equations for the clustering C and averaged path distance l when the underlying bipartite graph has Poisson distributed connections. The clustering coefficient $C(G_s)$ for the class projection of a random bipartite graph is:

$$C(G_s) = \frac{1}{\mu + 1} \tag{6}$$

where μ is the average number of methods referencing a class. There is a similar equation for the clustering coefficient $C(G_m)$ of the method projection:

$$C(G_m) = \frac{1}{\nu + 1} \tag{7}$$

where ν is the average number of classes referenced by a method. Note that $M\nu = N\mu$. The average distance l for the one-mode class projection of a Poisson bipartite graph is also very small:

$$l(G_s) = \frac{logN}{logz} \tag{8}$$

and

$$l(G_m) = \frac{logM}{logz} \tag{9}$$

where $z = \mu\nu$ is the expected average degree for the one-mode projection.

The bipartite network approach assumes that the class projection will explain most of the topological properties of the software graph. The software graph $\Omega_s = (W_s, E_s)$ should be a subset of the class projection $G_s = (W_s, D_s)$:

$$|E_s| = p\,|D_s|$$

where p is the fraction of edges lacking in the software graph with respect to the class projection. The differences between the Ω_s and G_s could be due to errors in the approximate reconstruction process of Ω_s described in the previous section or simply because the bipartite approach is not correct. We have performed a comparison between the topological properties of the software graph Ω_s and the same measures taken from the class projection G_s (see table 1). There is considerable agreement between the average degree, the clustering coefficient and average path length, thus suggesting that macroscopic software graph properties derive from the microscopic bipartite network (see Fig. 13).

Net	$< k >$	C	l
Ω_s	4.29	0.16	5.52
G_s	4.24	0.19	5.21
Random	13.04	0.08	2.71

Table 1: Comparison between software graph Ω_s and the projected graph G_s from the software bipartite graph G. Both Ω_s and G networks were obtained from a large software system analyzed in [18]. A Poisson bipartite graph is provided for comparison. Parameters: $N = 1071$ classes, $M = 9218$ methods, $\mu = 10.59$ and $\nu = 1.23$.

Still, we can appreciate how the real software graph deviates from the random bipartite network (having the same parameters as the real software system). The clustering coefficient is about one order of magnitude larger than the random counterpart (see table 1). Moreover, the average path length is two times larger than random. The random bipartite explanation tells us that the projection will be naturally correlated (clustered) and will be a small-world even if the underlying graph is uncorrelated (which is not the case), that is, the small-world property is achieved for free. The software graph will display a small-world architecture because of the encapsulation mechanism associated with object-oriented languages. The consequences of this observation still remain to be completely uncovered but we can safely conclude that the small-world behavior of software structures is not an additional requirement selected by human designers during software development.

5.2 Scale-Free Networks

Dijkstra was the first to claim that software engineers must be not only concerned with function but also with software structure [11]. He recognized the importance of having a well-organized software system that enables easy changes and modifications. He guessed that such an ideal software structure will be represented by a hierarchical tree. In these systems, function is provided by assembling the behavior of simple components which have clearly defined responsibilities. The coarse layout of these systems will resemble a planar graph, where nodes are software modules and edges depict existing collaborations. In this context, the aesthetics of the graph are associated with clever design. The planarity of the graph is interpreted as a signature of the clarity and clear separation of concerns achieved by its human designer.

The ordered diagram is also very homogeneous. It is easy to detect a repeating pattern in the way nodes connect to each other. In a tree, every node has only one predecessor and a slowly varying number of successors. This regular pattern can be detected by looking at the degree distribution $P(k)$ or the probability of a node having k connections. For a homogenous network like the tree or the random graph, this distribution follows the exponential distribution (equation 1.2). The main feature of this distribution is the small variance around

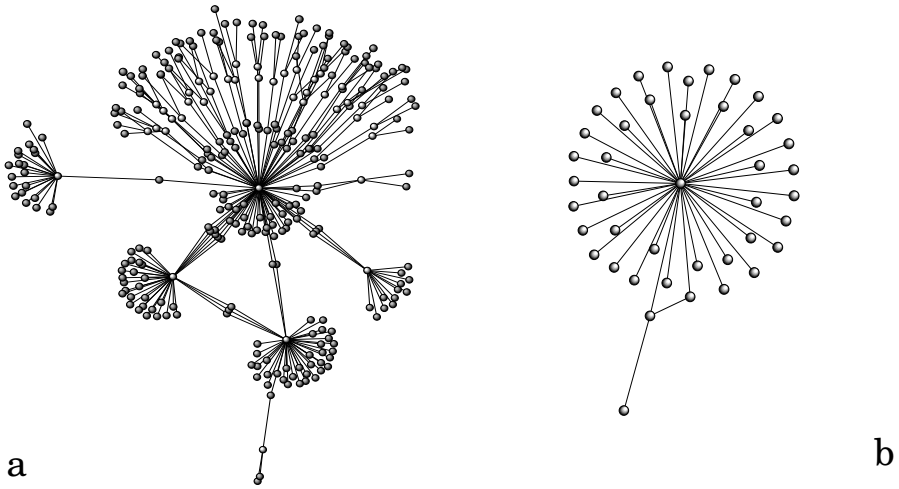

Figure 13: Looking at different granularities of the same software system. (a) A large subgraph of the class/method bipartite graph for the software application poEdit v1.2.5 (http://poedit.sourceforge.net/). Class nodes and method nodes are displayed with empty balls and gray balls, respectively. A noticeable feature is the asymmetry between average class degree ($\mu = 8.65$) and average method degree ($\nu = 2.38$). (b) The class projection for the previous bipartite graph coincides with the software graph obtained from the source code. The one-mode graph is star-shaped with a single triangle. The class at the center is the so-called hub.

the average degree. The graph has a well-defined scale.

An additional, widespread feature of many complex networks is the scale-free behavior of their degree distributions. Specifically, we have

$$P(k) = Ak^{-\gamma} \exp(k/k_c) \tag{10}$$

where A is a normalization constant, k_c is a cut-off degree and the scaling exponent γ is typically 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 [31][33]. These graphs are called 'scale free' (SF) and are found in many different contexts, from natural to technological systems [37]. Their ubiquity seems to stem from shared organizing principles [30]. SF networks are known to display some unexpected statistical features. In particular, looking at the moments of the degree distribution, i. e.

$$M_\mu = \int_1^\infty k^\mu P(k) dk \tag{11}$$

(with $\mu = 1, 2, ...$) and assuming that $P(k) \approx Ak^{-\gamma}$, it is easy to show that the average degree is well defined, leading to $< k >= (\gamma - 1)/(\gamma - 2)$, whereas

the higher moments are not, since they scale as

$$M_\mu = k^{\mu-\gamma+1} \tag{12}$$

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 [32].

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 [31]. 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 [34]. But the most interesting implications from SF architectures are related to their high robustness against random node failure, together with a high level of fragility when hubs fail [35]. In other words, information transfer keeps working in an efficient way when a randomly chosen node fails but typically degrades when a highly connected node fails. These observations have been shown to have immediate implications for reliable network architecture. Since a system's sensitivity to component failure is a fundamental problem in any area of engineering, it is important to recognize how network topology influences system performance.

Interestingly, all the software networks studied here are scale-free [18], that is, the degree distribution in software graphs scales with degree, $P(k) \sim k^{-\gamma}$. In order to properly estimate the scaling exponent γ, we have used the cumulative distribution $P_>(k)$, defined as follows:

$$P_>(k) = \sum_{k'>k} P(k') \tag{13}$$

so if $P(k) \sim k^{-\gamma}$, then we have

$$P_>(k) \sim \int P(k')dk' \sim k^{-\gamma+1} \tag{14}$$

A clear regularity is that the exponents obtained from the directed software network differ from the undirected one. Typically, 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 (see figure Fig. 14). They are more heavy tailed for the in-degree and more rapidly decaying for the out-degree distribution. Classes with high in-degree typically result from broad reuse [21]. The reasons for such an asymmetry might be rooted in the economization of development effort and related costs [18]. In principle, maximum in-degree is unbounded because linking to a class imposes no cost on the reused class. On the other hand, the complexity of the class increases with the number of used classes. The benefit of reusing some externally provided functionality is overwhelmed by the additional machinery required, which limits the maximum out-degree.

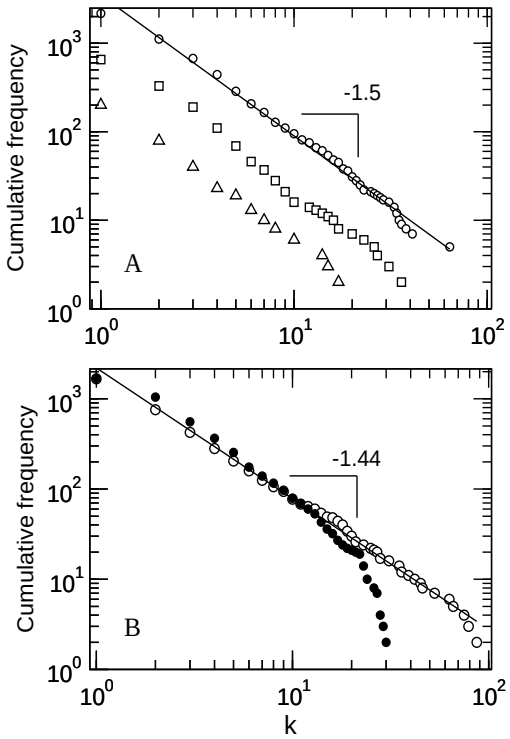

Figure 14: (a) Cumulative degree distributions for different software graphs varying in size: N=129 (triangles), N=495 (squares) and N=1488. All distributions have an exponent about -2.5 in spite of the obvious differences in size and functionality. (b) Asymmetry of in-degree (open circles) and out-degree (black circles) distributions for ProRally 2002 system. 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.

Our recent studies suggest that the scale-free pattern is the fingerprint of some universal pattern of software development. The SF pattern 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 software development. The fact that all the systems analyzed 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.

5.3 Software Evolution yields Scale-Free Networks

The small-world behavior of software appears to be an unavoidable consequence of encapsulation, that is, simpler functions are grouped into components. However, the random bipartite model does not explain the slightly higher-than-

random average path length. The unaccounted differences force us to look for alternative explanations about their causes. One source of inspiration is the evolution of software itself. Considerable effort has been devoted to studies about the generation of scale-free networks by evolving processes. It seems that contingency plays a very important role in shaping the network. Barabsi and Albert (BA) proposed the "rich-gets-richer" mechanism as a universal process yielding a scale-free network. In the BA model, the degree of a node is a proxy of its importance. The network is grown by adding a new node at a time whose m links are preferentially connected to the most important (that is, the more connected) existing nodes [31]. Such an evolving process yields a scale-free network with an exponent of -3. Unfortunately, the BA model cannot realistically reproduce other features of networks (i.e., the BA network has very low clustering).

A related theoretical result by Puniyani and Lukose indicates that growing networks with constant average path length leads to a scale-free network [38]. They have shown that a randomly growing network under the constraint of constant average path length always yields a scale-free network, with an exponent between -2 and -3 [38]. The degree distribution for the evolved SF network is:

$$P(k) \approx k^{3-\frac{\alpha}{\beta}} \tag{15}$$

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

$$N^{\alpha} = \frac{1}{\langle k \rangle} \int^{k} k^2 P(k) dk \tag{16}$$

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

$$k_c \approx N^{\beta} \tag{17}$$

For the systems analysed in [18], we get $\beta = 0.62 \pm 0.09$ and $\alpha = 0.42 \pm 0.08$, which gives a predicted scaling exponent $\gamma = 2.59$. This is in very good agreement with the averaged exponent for the studied systems $< \gamma >= 2.57 \pm 0.07$.

In order to check if the SF software network is related to a pattern of constrained growth, we have analyzed the evolution of the computer game Prorally 2002, which is a large video game (about 2000 classes in its final release) developed by Ubisoft during two years. We have collected a large sample of class graph snapshots taken at different moments of Prorallys evolution. From this data set we have computed the time series of its average path length and we have observed that, after an initial and sudden jump, the evolution of average path length appears to be constant during Prorallys evolution. Fig. 15 displays the evolution of the average path length for the ProRally 2002 system [20][18].

Interestingly, the sudden jump in average path length does not correspond to a sharp increase in development activity due to some external deadlines or other external pressures. The evolution of number of nodes and links is almost

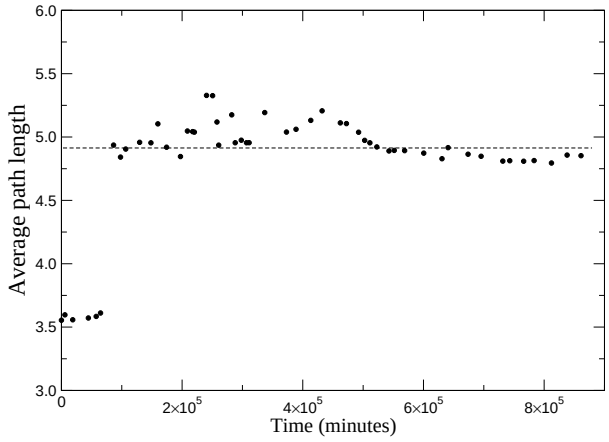

Figure 15: The evolution of average path length for the ProRally 2002 system. In spite that system size grows in a linear way, the average path length is kept constant during the project lifetime (see text).

linear (not shown) and thus, the average amount of work spent at each moment is more or less constant. Other projects we have analyzed show a similar growth pattern. We have shown that this pattern of constrained growth predicts the observed exponent of the degree distribution, thus confirming the result by Puniyani and Lukose. However, the origin of this constrained growth pattern remains to be explained. This is an apparently difficult question because building a model of software evolution appears to be a very complicated task. It is widely acknowledged that software is probably one of the most intricate human inventions. In principle, any useful model of software structure should take into account the many different mechanisms involved in computer programming. Unfortunately, some of the principles underlying computer programming are relatively unknown. For instance, it seems important to consider cognitive skills of computer programmers involved in this task. In this context, we are developing stochastic models of network growth that imitate common programming practices (like code duplication). The comparison of synthetic networks generated with these models with real software networks can shred some light into the mechanisms responsible of software growth and ultimately, the models will enable us to understand software development in a quantitative and unambiguous manner.

6 Conclusions

Understanding the origins of natural and artificial complexity requires the consideration of both their function and architecture. In order to perform a given function in an efficient way, not all topological patterns of interactions among units are allowed. Cost and proper communication are two essential require-

ments for most complex systems. Moreover, constraints of different nature exist: some are historical and others arise from both structural and dynamical limitations. Although it seems reasonable to think that the engineer overcomes the barriers which might be imposed to natural evolution, some lessons can be learnt from the analysis of both types of networks.

In this chapter we have reviewed a number of key features of software architecture and function in relation with its evolution and constraints. We have seen that, in spite the designed, human-driven evolution of software graphs, there are strong constraints limiting the effective repertoire of designs that are ultimately reachable. The scale-free, small world structure found in all, large-scale computer programs is a consequence of basic limitations imposed by an appropriate communication among different parts at low cost. It is certainly interesting to see that natural and artificial networks seem to share several key regularities at the network level. Eventually, models of software structure should provide insights into how internal and external forces constrains processes of artificial design.

Beyond the exact evolutionary rules shaping both types of structures, common principles might be at work. In this context, it is interesting to see that both software maps and electronic circuits [39] share some common features (such as their small world structure and their heterogeneity) with cellular networks, in spite of their differences in robustness. Such observations suggest that the plasticity implicit in cellular webs, as far as related to network topology, might inspire future developments of reliable technological systems by exploiting the common architectural patterns.

Bibliography

[1] T. Standage, *The Mechanical Turk*. Penguin Press, London (2002)

[2] J. J. Hopfield "Physics, computation, and why biology looks so different", J. Theor. Biol. 171, 53-60 (1994).

[3] S. J. Gould, *The Structure of Evolutionary Theory*, Belknap Press, Cambridge MA, (2002).

[4] S. A. Kauffman, *The Origins of Order: Self-Organization and Selection in Evolution*, Oxford University Press, 1993.

[5] F. Jacob, "Evolution as Tinkering", Science, vol. 196, 1161-1166, (1976).

[6] R. V. Solé and B. Goodwin, *Signs of Life: How Complexity Pervades Biology*

[7] B. Goodwin, *How the Leopard Changed Its Spots: Evolution of Complexity*, Charles Scribner's Sons, New York, (1994).

[8] J. von Neumann, "Theory of Self-Reproducing Automata", University of Illinois Press (edited and completed by A.W. Burke (1966).

[9] D. R. Hofstadter, *Gödel, Escher, Bach: an eternal golden braid*. Basic Books, New York (1979).

[10] R. V. Solé, R. Ferrer-Cancho, J. M. Montoya and S. Valverde, "Selection, Tinkering and Emergence in Complex Networks", Complexity, vol. 8(1), 20-33, (2002).

[11] E. W. Dijkstra, "The Structure of the 'T.H.E.' multiprogramming system", Comm. of the ACM, vol. 11, no. 5, pp. 453-457, (1968).

[12] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", Comm. of the ACM, vol. 15, 1053-1058, (1972).

[13] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass (1988)

[14] J. E. Hopcroft, R. Motwani and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston, 2nd Ed. (2000)

[15] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs", Comm. of the ACM, vol. 21, 8, 613-641, (1978)

[16] M.E.J Newman, S.H. Strogatz and D. J. Watts, "Random Graphs with arbitrary degree distributions and their applications", Santa Fe Institute working paper, SFI/00-07-042, (2000).

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

[18] S. Valverde and R. V. Solé, "Hierarchical Small-Worlds in Software Architecture", Santa Fe Institute, working paper SFI/03-07-044, (2003).

[19] S. Strogatz, *Sync: The Emerging Science of Spontaneous Order*, Hyperion Books, (2003).

[20] S. Valverde, R. Ferrer-Cancho and R. V. Solé, "Scale-Free Networks from Optimal Design", Europhysics Letters, 60, pp. 512-517, (2002).

[21] C. R. Myers, "Software Systems as Complex Networks: the Emergent Structure of Software Collaboration Graphs", Phys. Rev. E, vol. 68, 046116, (2003).

[22] A similar lexical reconstruction method is also performed by the automatic documentation system Doxygen by Dimitri van Heesch (http://www.doxygen.org).

[23] T. Ball and J. R. Larus, "Programs follow paths", Microsoft Research, Redmond, WA, Tech. Rep. MSR-TR-99-01, January, (1999).

[24] A. M. Turing, "On Computable Numbers", Proc. of the London Math. Soc., 2-42, pp. 230-265, (1936).

[25] J. von Neumann, *The Computer and the Brain*, New Haven, Yale University Press (1958).

[26] T. McCabe, "A Complexity Measure", IEEE Trans. on Soft. Eng, SE-2, 4, pp. 308-320, (1976)

[27] J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice-Hall, (1991).

[28] H-S. Chae, Y-S. Kwon and D-H. Bae, "A Cohesion measure for object-oriented classes", Soft. Pract. and Exp., vol. 30, 12, pp. 1405-1431, (2000).

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

[30] A.-L. Barabási and E. Bonabeau, "Scale Free Networks", Sci. Am., pp. 60-69, May (2003).

[31] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks", Science, vol. 286, pp. 509-512, (1999)

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

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

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

[35] R. Albert, H. Jeong and A.-L. Barabási, "Error and Attack Tolerance of Complex Networks", Nature, 406, pp. 378-382, (2000)

[36] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley, New York (1999).

[37] D. Braha and Y. Bar-Yam, "Topology of Large-Scale Engineering Problem-Solving Networks", Phys. Rev. E, vol. 69, 016113, (2004).

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

[39] R. Ferrer-Cancho, C. Janssen and R. V. Solé, Topology of technology graphs: small world patterns in electronic circuits. Phys. Rev. E. 63, 32767 (2001).