Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Power laws in software systems.

Roberto Tonelli

Advisor: Michele Marchesi
Curriculum: ING-INF/05 Informatica

Cycle XXIV
February 2012

Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Power laws in software systems.

Roberto Tonelli

Advisor: Michele Marchesi
Curriculum: ING-INF/05 Informatica

Cycle XXIV
February 2012

Dedicated to my wife.

Contents

List of Figures

*

List of Tables

*

Chapter 1

Introduction

The need for measuring software has become more and more impelling about two decades ago, with many documents devoted to software measures, both in the software industry and in the scientific literature [53], [19], [80], [32]. Without measurements software management can be very uneffective, since software products are extremely complex, and planning, estimations, and control become unaccurate. Software metrics were created in order to improve the process of software development, with the goal of measuring and controlling its essential parameters. Even if the meaning is (or was) used in a broad sense, software metrics generally refer to: product, process, resource, or project measurements. We restric our attention on the first meaning, dealing with product metrics alone, which may be distinguished in size metrics, complexity metrics, and quality metrics, generally related to each other. A milestone in the definition of a useful set of software metrics was the work of Kidamber and Kemerer [19], the first trial in addressing the problem of implementing a new suite of metrics for Object Oriented (OO) design.

This thesis tackles the problem of measuring software quality in Object Oriented (OO) systems by using such novel approaches and techniques gathered from all these different disciplines. The paradigm adopted considers modern OO software systems as complex software networks [78], according to its modern view derived from Barabasi-Albert [11] work's for the WWW network. This paradigm offers the opportunity of introducing software metrics which were out of the more traditional schemes of software engineering for measuring software, and of assessing software quality using new tools. The main reference points of this thesis for the estimation of software quality will be software defects, which will be considered as pivotal points, with all the other software metrics rounding around. Barabasi and Albert [10], were the first to investigate the concept of complex network, giving rise to the modern complex network theory. In a complex network the probability P(k) that a vertex in the network

interacts with k other vertices decays as a power law, following $P(k) \simeq k^{-\alpha}$, where α is the power law exponent, providing a scale-free topology. This has been found valid also for the internet [30].

The concept of complex network has been extended to software systems [76], [56], and to OO software [60], were the network's nodes were identified with software components, and the network's edges with the interactions among them. As a consequence, power law distributions, scale free and small world properties and even fractals properties of complex networks [70] are necessarily present in software systems. Despite such amount of recent researches in this field, the applications to software enginnering, and in particular to the improvement of software quality, is largely lacking.

Modern software systems are made by thousands of classes linked by thousands of dependency relationships and by millions of lines of code. Thus the investigation of their properties by mean of a statistical approach is not only opportune but is rather a must. Until the recent past, the statistics of software systems were usually resumed by using metrics averages and standard deviations, or quantiles, in order to characterize the measured software properties. The complex network perspective shows that such statistics may be meaningless, since the empirical distributions of software properties are very well approximated by power-laws in the tail. An approach which uses the modeling of statistical ditributions holding such property is thus preferable. Through this thesis statistical analysis is sistematically executed and statistical distributions eavy tailed, like power laws, will play a major role for the investigation of software properties and for assessing software quality.

Software engineering aims at engineerizing the way software systems are conceived, organized, assembled and written. With the advent of the Object Oriented paradigm, software is developed according to schemes where the interconnections among classes play a major role. This picture depicts objects exchanging messages each other, where each single object plays its own role (functionalities), and interacts with other software actors. Its natural association with a complex network has been depicted by Myers' work [56], in which classes are interpreted as nodes and class dependencies as links. Such schematic view allows immediately to understand how defects can affect software in different proportions: a code defect introduced into a class linked to a very few classes will probably be less effective than a defect affecting a largely linked class. The implications of such a simple concept for the software industry can be very large and can range from software maintainability costs, to defect detection strategies, to resources allocation and so on.

One of the problems of using the information known about the localization of code defects for improving software quality and for reducing development and maintenance costs is that defects appear after software is developed, thus

their measure means the software is already defective. On the contrary the ideal or hypothetical aim of software engineering is to produce non defective software. Software engineering thus must look for metrics which can be measured during software development and that can provide side information about the probability of finding defects after coding.

This thesis explores novel metrics that can be of help in assessing where code defects likely lie, and that can be kept in control during software development, in order to reduce the probability of introducing defective code. The thesis affords these problems first performing an empirical analysis of benchmark software systems, in order to recover the empirical distributions of software defects and of various software metrics. Then exploiting and building analytical models for explaining the empirical data. Finally proposing novel software metrics related to software quality through their relationships with code defects, all associated to the complex network theory. Last but not least, the previous concepts have been used to study the role of refactoring for improving software quality.

1.1 Thesis overview.

The thesis is organized according to this scheme:

- *Chapter 2* provides an overview of the concept of complex software network, and a state of the art of related works. In fact, typically, software systems are built out of many interacting modules and subsystems, at many levels (functions, classes, interfaces, libraries, source files, packages, etc.). This modular structure, where software entities interact reciprocally, suggests a graph based representation, where software entities can be represented as nodes, and all different relationships, like object interactions and routine calls, can be represented as connections between them.

- *Chapter 3* presents the metrics we used through this thesis for understanding the structure and properties of software networks. In particular it describes the Social Network Analysis metrics (SNA metrics) and the Fractal Dmension for complex networks.

- *Chapter 4* discusses the statistical distribution chosen for exploring the structure of complex networks and the statistics of empirical data measured in lareg siftware systems. It also presents the generative model associated to the more important statistical distributions, and describe how these generative models can be practically applied to large software systems.

- in *Chapter 5* we discuss in detail the Yule process, and describe an application for modeling the behavior of various software systems written in Java. It also shows how the evolution of such systems through differnt releases can be fit very well by this model.

- in *Chapter 6* we describe the statistical properties of bugs and present a generative model for explaining the empirical distributions. In fact, the distribution of bugs in software systems has been shown to satisfy the Pareto principle, and typically shows a power-law tail when analyzed as a rank-frequency plot. We further discuss the subject from a statistical perspective, using as case studies five versions of Eclipse, to show how log-normal, Double Pareto and Yule-Simon distributions may fit the bug distribution at least as well as the Weibull distribution.

- in *Chapters 7, 8* we present an analisys of software networks by using the SNA metrics. We investigate if the new proposed SNA metrics possess the same statistical properties found for bugs and have similar empirical distributions. Moreover, study the possible correlations with Bugs and/or with other metrics and properties. We also investigate if these analytical distribution functions which may be used to forecast future properties of the software systems. Next we present the analysis of the software graphs of 96 systems of the Java Qualitas Corpus, obtained parsing the source code and identifying the dependencies among classes. For two systems, Eclipse and Netbeans, we computed also the number of bugs, identi- fying the bugs affecting each class, and finding that some SNA metrics are highly correlated with bugs, while others are strongly anticorrelated.

- in *Chapter 9* we propose for characterizing software quality the Fractal Dimension of software networks. In this chapter we present an algorithm for computing the fractal dimension of a software network, and compare its performances with two other algorithms.

- in *Chapter 10* we analyzed various releases of the publically available Eclipse software systems, calculating the fractal dimension for twenty sub-projects, randomly chosen, for every release, as well as for each release as a whole. Our results display an overall consistency among the sub-projects and among all the analyzed releases. We found a very good correlation between the fractal dimension and the number of bugs for Eclipse and for twenty sub-projects.

Chapter 2

Software Networks

2.1 Literature Overview

Recently, some researchers have started to study the field of software, in the perspective of finding and studying the associated complex graphs and their statistical properties. In fact, many software systems have reached such a huge dimension that it looks sensible to treat them using the stochastic random graph approach [34]. Typically, software systems are built out of many interacting modules and subsystems, at many levels (functions, classes, interfaces, libraries, source files, packages, etc.). This modular structure, where software entities interact reciprocally, suggests a graph based representation, where software entities can be represented as nodes, and all different relationships, like object interactions and routine calls, can be represented as connections between them. A recent example of a comprehensive graph-based approach to model object-oriented designs is the “Big-Bang" Graph Representation proposed by Li [46]. Such graph representation has also the advantage of being useful for analizing scale-free behavior or power-law distribution of some software properties.

Modern software systems are made of many elementary units (software modules) interconnected in order to cooperate to perform specific tasks. In particular, in OO systems the units are the classes, which are in turn interconnected with each other by relationships like inheritance and dependency. Recently, it has been shown how these software systems may be analyzed using complex network theory [76] [23] [60]. In software networks, the classes are the nodes and the relationships among classes are the edges. This property opens the perspective to analyze software networks using metrics taken from other disciplines, like Social Network Analysis (SNA) [65]. The SNA metrics can be used together with more traditional product metrics, like class LOCS, number of Bugs, or the CK suite, to gain a deeper insight into the properties of software

systems. Recent studies showed the importance of SNA metrics in measuring the interactions among software modules [87], and in particular how centrality measures are useful to identify software hubs, which show higher defect-proneness.

Considering software systems as graphs is not a new approach, and different authors have already investigated some of their properties, like the distribution of Fan-in or Fan-out of network nodes [48], [76], finding features characteristic of complex networks, like for instance the presence of power-laws in the tail of the distributions of these metrics [23] [13].

An oriented graph can be associated to an OO software system, whose nodes are classes and interfaces, and whose edges are the relationships between classes, namely inheritance, composition and dependence. This approach has already been used in literature. In [78] complex software networks were analyzed with nodes representing software entities at any level, and links representing syntactical relationships between modules, subprograms, instructions. In [48] software is seen as a network of interconnected and cooperating components, choosing modules of varying size and functionalities, where the links connecting the modules are given by their dependencies. In [87] nodes are binaries, and edges are dependencies among binaries pieces of code. In [81] inter-class relationships where examined in three Java systems, and in [13] the same analysis was replicated on the source code of 56 Java applications. Object graphs were analyzed in [60] in order to reveal scale-free geometry on object oriented programs, were the objects were the nodes and the the link among objects were the network edges.

Many software systems have reached such a huge dimension that it looks sensible to treat them as complex networks [34], [56]. The study of real complex networks has revealed that many of them share some fundamental common properties. One of these properties is related to the degree distribution for these networks, that often follows a power law [10, 11]. Networks that exhibit this kind of distribution are known as scale-free networks, indicating the presence of few highly connected nodes (usually called hubs) and a large number of nodes with small degree. Another important property is the small-world feature, also known as the generalization of the famous 'six degree of separation'[52]. In small-world networks, a very small number of steps is required to reach a given node starting from any other node. A recent paper [70] has found that the structure of these networks is often also self-similar, and it is possible to calculate their fractal dimension. These properties have been found also for software networks [56], [77]. This is the reason which justifies the assumption that software networks are complex networks, and the motivations to study them using the same approach.

Object Oriented programming prescribes to assign different roles and re-

sponsibilities to different classes of a software system, where they interact with each other through their dependencies. The overall system "behavior" results from the specificity of each object and from the interactions among such "individuals". At source code level, these interactions are identified by the dependencies among classes, and related to the software architecture. It is thus interesting to analyze these software systems as a network of nodes, or individuals, connected by edges due to relationships among them, representing classes and their dependencies.

2.2 Object-Oriented Systems as Networks

The basic building block of OO programming is the class, composed of a data structure and of procedures able to access and process these data. The data structure is made up of fields (instance or class variables) that represent the state of an object. A class has also a behavior expressed in terms of methods that represent the procedures able to access and process the data structure. Classes may be defined at various levels of complexity, and are related across different kinds of binary relationships, such as inheri- tance, composition and dependence, which are well- known properties of OO design. For the software systems considered in this work, but also in general, there is not unique way for building a software network. One first difference can be introduced at the vertex level, discriminating among simple classes, abstract classes, or interfaces. Another difference can exist at the link level, for example distinguishing among directed or un-directed links. It is also possible to consider only particular kinds of links, like dependencies alone, and so on. In this thesis we will consider not only the classes, but when it is convenient, also abstract classes or interfaces, and even Compilation Units (CU) as networks nodes, while dependencies and inheritance as binary relationships, namely network links. Analyzing the source code of an OO system, it is possible to build its class graph -a graph whose nodes are the classes, and the graph edges represent directed relationships between classes (2.1).

In this graph, the in-degree of a class is the number of edges directed toward the class, and is related to the usage level of this class in the system, while the out-degree of a class is the number of edges leaving the class, and represents the level of usage the class makes of other classes in the system. It has already been shown that OO software networks exhibit the scale-free and small-world properties, and thus can be considered complex networks. The in- degree distributions are power laws with exponent $\gamma \simeq 2.5$ [48], [22], while the out-degree distributions are more controversial, and are mainly log-normal or Double-Pareto distributions [22], [54]. The dispute among log-normal or double Pareto (or

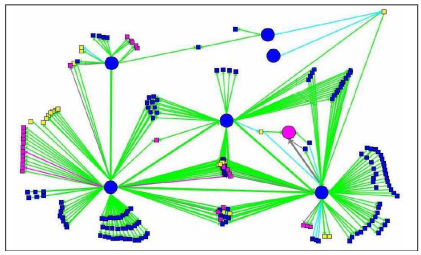


Figure 2.1: Fig. 2.1. Example of a portion of class graph for Eclipse. Classes, abstract classes and interfaces, as well as different relationships, have different colors.

even simple Pareto) is not purely academical. In fact it involves the validity and the suitability of different models of software process production, and presents potentially important practical implications on software quality and costs. The possible paths on the software graph, and thus the distances among nodes, may be different when considering the out- links or the in-links network. This may influence the final value of the fractal dimension. Thus, when convenient, we decided to consider only un-directed links.

2.3 Software Networks

According to the object oriented paradigm, a software system is the abstraction of a real domain, where objects interact reciprocally sending messages to each other. The common interface of a family of similar objects is embodied in the class construct. Different classes can be related through different type of relationships, such as inheritance, composition, aggregation, association and dependence. Not only classes and objects, but also many other elements, such as attributes, methods and interfaces, are defined in an object oriented system, representing entities at a different level of granularity and abstraction. Thus, for any software system built according to the object oriented approach, it is possible to easily define different kind of networks made of nodes representing specific software entities and connections representing relationships between them. For example, it is possible to define the network made of the run time objects, where two class instances are connected if one sends a message to the other; a static class network where a node is the representation of a class and a connection is the representation of a relationship between them, such like inheritance or dependence; a more detailed network, where different types of node represent different types of entities, such classes, attributes and methods, and different types of connection are defined to represent spe-

cific relationships among them, such as instantiation between classes and attributes, calling between methods, and so on. This thesis will generally focus on the representation of classes and their relationships, but sometimes we will extend the concept of software network to Compilation Units (CU – java files containing one or more classes) and their relationships. Specifically, we will consider inheritance and dependence relationships. Once defined the graph model used to capture the structure of the software system under study, a code analyzer must provided to generate such a graph, starting from the software system code. With strong typed languages, this would be accomplished by parsing the source code, recognizing class and variable definitions, and generating the graph –a graph whose nodes are the classes, and the graph edges represent directed relationships between classes. The in-degree distributions are power law like others software properties [75]. The out-degree distribution are more controversial. Some outhors found power law behaviour [56] whereas others found a lognormal behavior [23].

Once the network is generated, it is possible to compute on it all the metrics that can be associated to a network, including the fractal dimension. Note that the box-counting algorithm defined in [70] works only for non-directed graphs, because the distance between two nodes taken as the shortest path between the nodes would not satisfy the expected commutative property in the case of directed graphs. For this reason we did not consider link orientation in the computation of the fractal dimension of the graphs.

Chapter 3

Metrics for Software Networks

Measuring software to get information about its properties and quality is one of the main issues in modern software engineering. Limiting ourselves to object-oriented (OO) software, one of the first works dealing with this problem is the one by Chidamber and Kemerer (CK), who introduced the popular CK metrics suite for OO software systems [19]. Other OO metrics have been also proposed, like MOOD [14] and the Lorenz and Kidd metric suite [47], but the CK suite remains by far the most widely used. In fact, different empirical studies showed significant correlations between some of CK metrics and bug-proneness [20] [12] [71] [37].

3.1 SNA Metrics.

Once the software graph is defined, we can compute on this graph the metrics used in Social Network Analysis. We restricted ourselves to the subset of SNA metrics that were found most correlated to software quality [87] [24]. Some of these metrics are the so-called "EGO metrics". For every node in the graph, there exists a subgraph composed by the node itself, called "EGO" (from the Latin word "ego", meaning "I"), and its immediate neighbors. Such subgraph is called the EGO Network associated to the node. The analysis of the EGO-networks gives information about the role of the 'EGO' inside the entire network. In particular, EGO-network metrics provide insights on the extent each CU is connected to the entire system, and on the flow of information. In the definition of the EGO network, we considered the graph links as un-directed links.

Other SNA metrics we considered, not directly related to the EGO network, are some centrality metrics, determining how important a given node/edge is relative to other nodes/edges in the network. Overall, we consider the following

SNA metrics:

- Size. Size of the EGO-network related to the considered node (i.e. Compilation Unit); it is the number of the nodes of the EGO-network.
- Ties. Number of edges of the EGO-network related to the node.
- Brokerage: the number of pairs not directly connected in the EGO network, excluding the EGO node.
- Eff-size. Effective size of the EGO network; the number of nodes in the EGO network minus one, minus the average number of ties that each node has to other nodes of the EGO network.
- Nweak-comp. Normalized Number of Weak Components; the number of disjoint sets of nodes in the EGO network without EGO node and the edges connected to it, divided by Size.
- Reach-Efficiency; the percentage of nodes within two-step distance from a node, divided by Size.
- Closeness; the sum of the lengths of the shortest paths from the node to all other nodes.
- Information Centrality: the harmonic mean of the length of paths starting from all nodes of the network and ending at the node.
- DwReach; the sum of all nodes of the network that can be reached from the node, each weighted by the inverse of its geodesic distance. The weights are thus 1/1, 1/2, 1/3, and so on.

All previous metrics are computed on the class or CU graphs, and are among those studied in [87]. It is useful to shortly describe how these SNA metrics may be relevant to software systems, namely what they try to measure. The first five are strictly EGO metrics, and describe the software neighborhood of a class. Size measures the class directly connected to a given class while Ties, measured on such neighborhood, measures how dense are the software connections in this local network. Brokerage measures for how many couples of classes the given node acts like a broker, bridging the information flow among couples. Eff-size measures the redundancy of the connections in the EGO network, reducing the class Size by an amount proportional to the local average Ties. If the average Ties is high, the local network has in fact redundant channels available for the information flow. The role of the EGO class in the information exchange is then reduced. It must be noted that the average Ties refers

only to the local network, and not to the global network, where, as we will see in the following, the distribution of Ties among all the nodes presents a fat tail. Nweak-comp measures how much the class is needed to keep connected the other software units. The remaining are not EGO-metrics, and are all centrality metrics. They measure if, in the global software network, the class plays a peripheral rather than a central role.

3.2 SNA related works

Only recently, SNA has been applied to the study of software systems. Zimmermann and Nagappan used SNA metrics to investigate a network of binary dependencies [87]. With regard to the study of OO software systems, only Tosun et al., to the authors' knowledge, applied SNA metrics to OO source code to assess defect prediction performance of these metrics [74]. In particular, there are no studies investigating the relationships and the correlations among SNA metrics, traditional metrics, and Bugs metrics, and the corresponding statistical distributions. It must be noted that, when the measures are distributed according to power-laws, or other leptokurtotic distributions, traditional quantities like average or standard deviation may lose their meaning, and may be not characterizing measures anymore [57]. Knowledge of the overall statistical distribution is needed for characterizing the system properties. In particular, they are needed in order to obtain estimates of the metrics values for the future software releases. In past years, product metrics, extracted by analyzing static code of software, have been used to build models that relate these metrics to failure-proneness [51] [71] [20] [12] [37]. Among these, the CK [16] suite is historically the most adopted and validated to analyze bug-proneness of software systems [71] [20] [12] [37]. CK suite was adopted by practitioners [20] and is also incorporated into several industrial software development tools. Based on the study of eight medium-sized systems developed by students, Basili et al. [12] were among the first to find that OO metrics are correlated to defect density. Considering industry data from software developed in C++ and Java, Subramanyam and Krishnan [71] showed that CK metrics are significantly associated with defects. Among others, Gimothy et al. [37], studying a Open Source system, validated the usefulness of these metrics for fault-proneness prediction.
CK metrics are intended to measure the degree of coupling and cohesion of classes in OO software contexts. However,

The studies done using CK metrics do not consider the amount of "information" passing through a given module of the software network. Social Network Analysis (SNA) fills this gap, providing a set of metrics able to extract a new, different kind of information from software projects. Recently, this ability of SNA

metrics was successfully employed to study software systems. Zimmermann and Nagappan [87] showed that network measures derived from dependency graphs are able to identify critical binaries of a complex system that are missed by complexity metrics. However, their results are obtained considering only one industrial product (Windows Server 2003). Tosun et al. [74] reproduced the previous work [87] extending the network analysis in order to validate and/or refute its results. They show that network metrics are important indicators of defective modules in large and complex systems. On the other hand, they argue that these metrics do not have significant effects on small scale projects. Both previous studies [87] [74] did not consider mutual relationships among SNA metrics and complexity metrics, therefore they did not show if SNA metrics carry new information with respect to CK suite. In this thesis, instead, we compute the correlation matrix among SNA Metrics and CK metrics, considering also mutual correlations with respect to Issue, Bug, LOC, Fan-out and Fan-in.

3.3 Software Networks Fractal Dimension.

A recent study [70] found that the structure of complex networks is often also self-similar, and it is possible to calculate their fractal dimension using the box-counting method. As we saw, this method consists in covering the entire set, the network in this case, with the minimum number of boxes NB of linear size (diameter) lB. For a given network G and box size lB, a box is a set of nodes where all distances l_{ij} between any two nodes i and j in the box are smaller than l_B. If the number of boxes scales with the linear size l_B following a power law (see eq. 3.1), then d_B is the fractal dimension, or box dimension, of the graph [5]: The computation of the fractal coefficient of a network is thus a two-step one. First, an assessment of the self-similarity of the network has to be done, com- puting the minimum number of boxes covering the network, varying lB from one to a given number, usually 10 or 20. This is the most computational intensive step. Then, one has to check whether $N_B(l_B)$ is linear in a log-log plot, showing a power-law behavior. This check is somewhat subjective, though it is possible to compute confidence intervals to this purpose. Eventually, an estimate of dB is made fitting the plot with an LMS algorithm. It has been already reported that OO software networks related to classes of large Smalltalk and Java systems show a patent self-similar behavior, with frac- tal dimension between 3.7 and 5.1 [22]. So, for OO software networks it is important to have efficient and reliable algorithms able to compute their fractal dimension. A general method has been devised by Song et al. [70].

The method consists in covering the network with N_B box of linear size l_B. The number of boxes N_B needed to tile the entire network is computed for different values of l_B. If the number of boxes scales with the linear size l_B following a power law (see eq. 3.1), then d_B is the fractal dimension, or box dimension, of the graph [31], [70]:

$$N_B(l_B) \sim l_B^{-d_B} \tag{3.1}$$

The computation of the fractal dimension of a network is thus a two-step one. First, we need to compute the minimum number of boxes covering the network, varying l_B from one to l_{Bmax}, between 10 and 20. Then, the last step is somewhat subjective because one has to check whether $N_B(l_B)$ is linear in a log-log plot.

The slope of the straight line fitting the log-log plot is the fractal dimension d_B. The fractal dimension has been already reported also for OO software networks [22]. In fig. 3.1 we report the box counting analysis of the software network related to Eclipse 3.2. Similar plots are observed also for the other versions of Eclipse analyzed, and for the Eclipse sub-projects the power law behavior is patent.

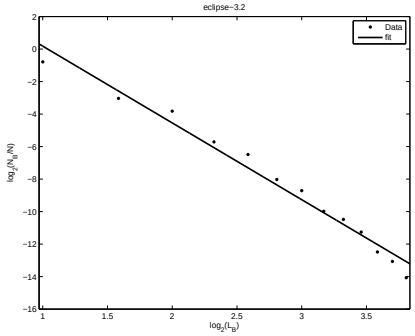


Figure 3.1: *Log-log plot of N_B vs. l_B for Eclipse 3.2.*

The coverage algorithm used to tile the entire network with boxes of size l_B was devised after a comparison between three algorithms, both in terms of performance and precision [24]. The best compromise between the algorithms analyzed is the 'Greedy coloring'algorithm described by Song et al. [69]. They showed how the box counting problem can be mapped to a graph node coloring problem, with no edges connecting two nodes of the same color, which is a NP-hard problem. We evaluate a two-dimensional matrix c_{il} of size $N \times l_{B_{max}}$, whose values represent the color of node i for a given box of size $l = l_B$. The steps are as follows:

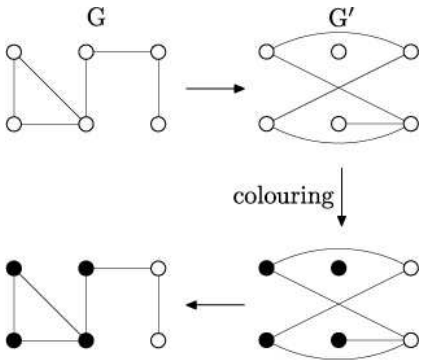


Figure 3.2: *Scheme of the construction of the dual network G', with given box size $l_B = 3$. The greedy algorithm is used for vertex coloring on G'. This figure is taken from [69].*

a) Assign a unique id from 1 to N to all network nodes, without assigning any colors yet.

b) For all l_B values, assign a color value 0 to the node with id = 1, i.e. $c_{il} = 0$.

c) Set the id value i = 2 and repeat the next steps until i = N.

d) Calculate the distance l_{ij} from i to all the nodes in the network with id j less than i.

e) Set $l_B = 1$.

f) Select one of the unused colors $c_{jl_{ij}}$ from all nodes j < i for which $l_{ij} < l_B$. This is the color c_{jl_b} of node i for the given l_B value.

g) Increase l_B by one and repeat (c) until $l_B = l_{B_{max}}$

h) Increase i by 1.

i) end.

In Fig 3.2 we illustrate an example with $l_B = 3$, where we build the dual graph G' from the graph G connecting two nodes with distance larger than or equal to l_B. With the greedy algorithm we color the vertex in G' and we go back to the box-covered G network.

Chapter 4

Power Laws in the Tail and Modeling Software Properties.

There are in nature, as well as in man-made systems, signatures of properties distributed according to a power-law distribution. Examples of such signatures occur in proteins and genes families, city sizes, people and firm wealth, the Internet and the Web, word frequencies, the realm of scientific citations and others. Sometimes these properties may be organized, in a natural way or for convenience of representation, according to a complex network structure. While some of these networks, such as the WWW and the Internet, display a global power-law shape within the whole degree distribution, many real networks show the scale-free trend effects only in the tail of the distribution. A power-law distribution, also called Pareto distribution or Zipf's law, implies that small occurrences are extremely common, whereas large instances are very rare, but can occur with not negligible probability. When data are distributed according to a power law, it is possible to find samples whose values are as large as the sum of the values of many (or most) other samples. For example, the wealth of the richest man in the U.S. is equal to the sum of the wealth of several tens of million other persons.

There are many possible different mechanisms able to generate power-law distributions. Among them, the most convincing and widely applicable mechanisms is perhaps the Yule process [55] [57]. Let us consider a system made of generic entities, each with a measurable property. Examples of such systems are the collection of research papers, the set of actors playing in movies, and the Web pages. Their properties may be, respectively, the number of citations received, the number of other actors cast in the same movies, and the number of hyperlinks to that page from other Web pages. In a Yule process, entities get random increments of a given property in proportion to their present value of that property. Thus, the probability of a paper obtaining a new citation is pro-

portional to the number of citations it already has; the probability that a new actor is cast with a well-established actor is higher than the probability of her being cast with a less known actor. Likewise, in a Web page it is more likely that a link is made to a well-known page which is already well connected, and so on. This type of rich-get-richer process has been called “the Gibrat principle" [67],“cumulative advantage" [29], or preferential attachment [10].

Further examples of these properties are the lines of code of a class, a function or a method; the number of times a function or a method is called in the system; the number of time a given name is given to a method or a variable, and so on. Later on we will present a study where the Yule Simon model is used to clearly represent the empirical data.

4.1 Related Works

Some authors already found significant power-laws in software systems. Kai and Yin [42] found that the degree distribution of software execution processes may follow a power-law or display small-world effects. Potanin et al. [60] showed that the graphs formed by run-time objects, and by the references between them in object-oriented applications are characterized by a power-law tail in the distribution of node degrees. Valverde et al. [76][77] found similar properties studying the graph formed by the classes and their relationships in large object-oriented projects. They found that software systems are highly heterogeneous small world networks with scale-free distributions of the connection degree. Myers [56] found analogue results on large C and C++ open source systems, considering the collaborative diagrams of the modules within procedural projects and of the classes within the OO projects. He also computed the correlation between some metrics concerning software size and graph topological measures, revealing that nodes with large output degree tend to evolve more rapidly than nodes with large input degree. Other authors found power-laws studying C/C++ source code files, where graph edges are the files, while the include relationships between them are the links [39], [26]. Tamai and Nakatani [72], proposed a statistical model to analyze and explain the distributions found for the number of methods per class, and for the lines of code per method, in a large object-oriented system.

While most of these studies are based on static languages, such like C++ and Java, Marchesi et al.[49] provide evidence that a similar behavior is displayed also by dynamic languages such as Smalltalk. Concas et al. found power-law and log-normal distributions in some properties of Smalltalk and Java software systems – the number of times a name is given to a variable or a method, the number of calls to methods with the same name, the number of imme-

diate subclasses of a given class of five large object-oriented software system [22], [23]. The Pareto principle is used to describe how faults in large software systems are distributed over modules [33], [58], [59], [8], [85]. Baxter et al. [18] found power-law and lognormal distributions in the class relationship in Java programs. They proposed a simple generative model that reproduces the features observed in real software graph degree distributions. Ichii et al. [41] investigated software component graphs composed of Java classes finding that in-degree distribution follows the power law distribution and the out-degree distribution does not follow the power-law. Louridas et al. [48], in a recent work, show that incoming and outgoing links distributions have in common long, fat tails at different levels of abstraction, in diverse systems and languages (C, Java, Perl and Ruby,). They report the impact of their finding on several aspects of software engineering: reuse, quality assurance and optimization. Also Wheeldon and Counsell [81], as well as other researchers, found power-laws for the distribution of many software properties, like the number of fields, methods and constructors of classes, the number of interfaces implemented by classes, the number of subclasses of each class, as well as the number of classes referenced as field variables and the number of classes which contain references to classes as field variables.

Thus, there is much evidence that power-laws are a general feature of software systems. Concas et al. [22], explained the underlying mechanism through a model based on a single Yule process in place during the software creation and evolution.

4.2 A More Convenient Representation.

It is common practice to report empirical data statistics by mean of histograms, which are a rough approximation of a Probability Density Function (PDF). Such representation however carries many drowbacks, in particular when data are power-law distributed in the tail. The problems with representing the empirical PDF are that it is sensitive to the binning of the histogram used to calculate the frequencies of occurrence, and that bins with very few elements are very sensitive to statistical noise. This causes a noisy spread of the points in the tail of the distribution, where the most interesting data lie. Furthermore, because of the binning, the information relative to each single data is lost. All these aspects make difficult to verify the power-law behavior in the tail. To overcome these problems we systematically report the experimental CCDF (Complementary Cumulative Distribution Function) in log-log scale, as well as the best-fitting curves in many cases. This is convenient because, if the PDF (probability distribution function) has a power-law in the tail, the log-log plot displays a straight

line for the raw data. This is a necessary but by no means a sufficient condition for power-law behavior. Thus we used log-log plots only for convenience of graphical representation, but all our calculations (CDF, CCDF, best fit procedures and the same analytical distribution functions we use) are always in normal scale. With this representation, there is no dependence on the binning, nor artificial statistical noise added to the tail of the data. If the PDF exhibits a power-law, so does the CCDF, with an exponent increased by one. Fitting the tail of the CCDF, or even the entire distribution, results in a major improvement in the quality of fit. An exhaustive discussion of all these issues may be found in [57]. This approach has already been proposed in literature to explain the power-law in the tail of various software properties [23] [48].

The CCDF is defined as $1 - CDF$, where the CDF (Cumulative Distribution Function) is the integral of the PDF. Denoting by $p(x)$ the probability distribution function, by $P(x)$ the CDF, and by $G(x)$ the CCDF, we have:

$$G(x) = 1 - P(x) \tag{4.1}$$

$$P(x) = p(X \leq x) = \int_{-\infty}^{x} p(x')dx' \tag{4.2}$$

$$G(x) = p(X \geq x) = \int_{x}^{\infty} p(x')dx' \tag{4.3}$$

4.3 Distribution Functions and Generative Models

In this thesis, we used different distribution functions for explaining different properties of software systems which exhibit power-laws in the tail. In general, associated to a statistical ditribution, there is a generative model, which explains the reason why the empirical data display such particular statistical distribution, or the evolutionary process generating the empirical distribution. In this section, we present four distribution functions which are suited to model sample data presenting leptokurtic behavior, that is with a "fat tail", which is the case for many software properties, including "bugs hitting modules" propery [48]. These distributions were already used for software data. For each distribution, we briefly discuss how it might be linked to a generative process, when it is possible. Note that the straight power-law distributions, can be treated as special cases of the "Double Pareto" distribution, as explained below.

The first distribution is the well-known log-normal distribution. If we model a stochastic process in which new elements are introduced into the system units in amounts proportional to the actual number of the elements they contain, then the resulting element distribution is log-normal. All the units should have the same constant chance for being selected for the introduction of new

elements [57]. This general scheme suits large software systems where, during software development, new classes are introduced into the system, and new dependencies –links– among them are created. Or for modeling the process of bug introduction into software modules. It might be reasonable to assume that, during development, each module has roughly the same random probability of being updated, and that bugs (or dependencies leading to the fact that the file needs to be changed due to the correction of another module) are introduced proportionally to the amount of bugs already existing in that module. In fact, a module containing bug-prone code, or simply containing more code, has a higher chance to develop new bugs when updated with respect to a module containing better code, or a lesser amount of code. This process, however, lacks the ability to correctly model the introduction of a bug into a module having no bugs yet. The log-normal has also been used to analyze the distribution of Lines of Code [86]. The log-normal distribution has been also proposed in literature to explain different software properties ([13], [22], [48]). Mathematically it is expressed by:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma}x} e^{-\left(\frac{ln(x)-\mu}{2\sigma}\right)^2} \tag{4.4}$$

It exhibits a quasi-power-law behavior for a range of values, and provides high quality fits for data with power-law distribution with a final cut-off. Since in real data largest values are always limited and cannot actually tend to infinity, the log-normal is a very good candidate for fitting power-laws distributed data with a finite-size effect. Furthermore, it does not diverge for small values of the variable, and thus may also fit well the bulk of the distribution in the small values range.

The power-law is mathematically formulated as:

$$p(x) \simeq x^{-\alpha} \tag{4.5}$$

where α is the power-law exponent, the only parameter which characterizes the distribution, besides a normalization factor. Since for $\alpha \geq 1$ the function diverges in the origin, it cannot represent real data for its entire range of values. A lower cut-off, generally indicated x_0, has to be introduced, and the power-law holds above x_0. Thus, when fitting real data, this cut-off acts as a second parameter to be adjusted for best fitting purposes. Consequently, the data distribution is said to have a power-law in the tail, namely above x_0.

The third distribution is the Double Pareto distribution. It is an extension of the standard power-law, in which two different power-law regimes exist. There are different forms of literature for the Double Pareto distribution [54] [63] [64] [36]. We use the form described in [36] for the CCDF, because it provides a smoother transition across the two different power-law regimes:

$$P(x) = 1 - \left[\frac{1 + (m/t)^{-\alpha}}{1 + (x/t)^{-\alpha}} \right]^{\beta/\alpha} \tag{4.6}$$

The Double Pareto distribution is able to fit a power-law tail in the distribution, with the advantage of being more flexible than a single power-law in fitting also the head of the distribution, where it is very similar to a log-normal. This distribution was successfully used to model the process of file creation in a software system, providing the correct distribution of files size [54] [62].

The Double Pareto distribution can be generated by a set of stochastic processes, each one of the form described above for the log-normal distribution, but which are performed in parallel and can be randomly stopped, as in the Recursive Forest File model by Mitzenmacher [54]. A description of the model and the implications on bugs insertion into software modules will be given in the one of the following chapters. Here we briefly discuss how such a process might be applied to bug generation, considering that a large software system can be divided into many subsystems that are worked on in parallel, and that from time to time some of these subsystems can be declared stable, and thus "frozen", with no further activity performed on them. This addition model is clearly more realistic than a single monolithic system whose modules are all subject to be updated in the same way, as in the generative process associated with the log-normal distribution.

The fourth distribution we consider is the Weibull distribution, used to explain the rate of faults in a technical system, due to the failure of single components. Its CCDF is given by eq. (4.7):

$$P(x) = e^{-(\frac{x}{\gamma})^{\beta}} \tag{4.7}$$

Weibull distribution is commonly used in reliability engineering, and with shape parameter $\beta \simeq 2$ it was applied for estimation of bug detection rates during software development and after deployment [82]. A direct application to bug occurrences in modules in a snapshot of the source code of a system has not yet been proposed. However, the Weibull distribution is very flexible, and was found to fit the Alberg diagram of bugs [85] very well, with shape parameter $\beta < 1$.

Finally, we consider the Yule-Simon distribution, introduced to explain the power-law in the tail of the distribution of genera and species in nature [84], and of the frequency of words in books [67]. The mechanism behind the generation of a power-law distribution is called "preferential attachment".

Let us consider a system made of entities, labeled by i, with a property characterized by a value n_i^t at time step t. As the system evolves new entities are added with an initial property value h_0, that may be null, as in our case, and the

properties of randomly chosen existing entities are incremented. The preferential attachment mechanism states that the entity selected for this increment is chosen with probability proportional to the current value of its property.

Using a master equation it can be demonstrated that the preferential attachment generates a population with a probability distribution function that can be expressed through the Legendre Beta function – in turn depending on the Euler Gamma function – depending on two parameters, α and c [57]:

$$p(x) = p_0 \frac{B(x+c,\alpha)}{B(x,\alpha)} \tag{4.8}$$

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \tag{4.9}$$

where parameters c and α are derived from the Yule model of the growth of Genera and Species in nature [84] [57]. It produces a distribution with a power-law in the tail with exponent α. In the next chapter we'll develop further the applications and the details of the Yule model.

Chapter 5

The Yule Process for Modeling Software

We present a model based on the Yule process, able to explain the evolution of some properties of large object-oriented software systems. We study four system properties related to code production of four large object-oriented software systems - Eclipse, Netbeans, JDK and Ant. The properties analysed, namely the naming of variables and methods, the call to methods and the inheritance hierarchies, show a power-law distribution as reported in previous papers for different systems. We use the simulation approach to verify the goodness of our model, finding a very good correspondence between empirical data of subsequent software versions, and the prediction of the model presented.

In order to show the suitability of the Yule process for modelling software systems statistical properties we study the same properties analysed in Concas et al. [22] [23], which are: i) the names of instance variables ; ii) the names of methods; iii) the calls of methods from inside other methods; iv) the number of subclasses of a given class. We choose to investigate these properties not only because they show a patent power-law behaviour, but also because the former two are related to design and coding guidelines, while the last one is Chidamber and Kemerer (CK) NOC metrics [19]. Property iii), on the other hand, can give hints about coupling between methods, and thus between classes. All these properties might thus provide information about software quality and adherence to coding standards.

The aim of this chapter is to provide evidence that the Yule process is actually able to stochastically model in the large some software development activities, provided that it is suitably extended. We concentrate our analysis on some representative properties which unequivocally exhibit power-law behavior, and analyze not only four different large object-oriented software systems, but also their entire time evolution. While our study is limited to four impor-

tant aspects of software development, namely variable and method naming, method calls and subclasses, there is evidence that many other properties of software designs that exhibit a power-law behavior can be modeled with the same approach. This will be the object of the next chapter.

Since for each system and for each version, considered at the time of release, a single Yule process may account for the power-law distributions of the analyzed properties but the Yule parameters may be different among the versions, we introduce a modified Yule Process which is able to stochastically model the overall evolution of the properties analyzed among the different releases.

We also analyse the differential statistical distribution, obtained by subtracting, one by one, the number of occurrences of every property among two main versions (to be made precise later), showing that it is still a power law, translated downwards with respect to the original distribution. All the data on the four systems have been collected using public repositories on the net and from CVS (Control Version System) through a parser.

5.1 The Yule process

The Yule process was first developed by George Udny Yule (1871 -1951) who used it to study evolutionary models. He was looking for a distribution that would model the number of evolving species in a genus over time. His models have since then been widely used to build phylogenetic trees. This Process deals with a population of *entities*, each characterized by a *property* whose value is an integer. The development of the population happens in time steps, $t_s = 0, 1, 2, ..., t$. At each time step:

1. A new entity is generated with probability a and its property is set to the integer value k_0.

2. With probability $1 - a$, a selected entity grows one unit in its property value. The selection of the entity that grows is done with probability proportional to the property value it already has, plus a constant c [1].

At the beginning ($t_s = 0$), there is just one entity, with property value set at k_0. At time t, let the total number of entities be N, each having a property with integer value, x_i, $i = 1, \ldots, N$.

[1] The parameter c is introduced in order to preserve the preferential attachment mechanism when a new entity is created empty ($k_0 = 0$). In this case, the probability of acquiring new properties for the entity would be zero, being proportional to the properties it already has, and thus it may be convenient to set the initial property to a non null value c.

Let us call m the average number of property value increments in between the creation of a new entity and the next. m is easily related to a, accordingly to equation:

$$m = \frac{1-a}{a}. \tag{5.1}$$

In fact, on average, every $m+1$ times, the increments of a property value occur m times, and the creation of a new entity occurs one time. Thus the respective probabilities are $1-a = m/(m+1)$ and $a = 1/(m+1)$, which is the same as eq. (5.1).

For instance, if on average three entities are chosen for incrementing by one their property values in between the addition of two new entities, $m = 3$, and $a = 0.25$. Being an average value, m is usually a fraction.

The probability distribution in the Yule process, for an entity to have the value of its property equal to k is given by:

$$p_k = \frac{B(k+c,\alpha)}{B(k_0+c,\alpha)} p_{k_0} \tag{5.2}$$

where $B(a,b)$ is the Beta function which is defined by:

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \tag{5.3}$$

and Γ is the Euler function.

p_{k_0}, the probability that an entity retains its starting value k_0, is given by the equation:

$$p_{k_0} = \frac{k_0+c+m}{(m+1)(k_0+c)+m} \tag{5.4}$$

The analysis is reported in detail in [57], where α is related to the three parameters of the process according to the formula:

$$\alpha = 2 + \frac{k_0+c}{m} \tag{5.5}$$

If the number of entities is high enough, the Beta function can be approximated, $B(a,b) \simeq a^{-b}$, and the Yule process generates a power-law distribution with equation:

$$p_k = \frac{k^{-\alpha}}{((k_0+c+\alpha-1)B(k_0+c,\alpha))} \tag{5.6}$$

neglecting the term c, which is small with respect to the values of k in the tail of the distribution. The Yule process depends only upon the values of these three parameters: k_0, m and c. Note that we used the variable k to represent

the general distribution (5.6), while we use x when referring to properties of entities.

The main limitation of the proposed Yule process is that it does not take into account possible reductions of the value of the properties of some entities, and possible deletions of entities. For instance, regarding method names, when a method whose name is X is deleted, the value of the property of the entity X decreases by one unit. But if there was only one instance of method X in the system, then also the entity X itself should be deleted.

Luckily, it has been recently demonstrated that it is in any case possible to extend the Yule process to the case of property value reduction, and entity deletion, without losing its characteristics [83]. We need just to consider an "equivalent growth rate", given by the difference of the increase and deletion rates. Consequently, in this work we will not explicitly model deletion of entities or reduction of properties, but only their addition, considering the addition rate as an "equivalent growth rate".

5.2 Estimating Yule process parameters

In order to fit empirical data with a Yule process, we need to estimate the parameters k_0, m and c. The value of k_0 is determined by the model itself. Usually, k_0 is set to zero or to one, as we will discuss later for the system properties analyzed. Also m is determined by the model itself, or by empirical measurements on the distribution of the values of the entities's properties. In the latter case, remember that m is the average number of increments made to existing entities between the additions of two new entities. So, the value of m can be estimated on n existing entities by computing the sum of the values x_i of the properties of all entities, subtracting from this sum the amount due to the initial value, k_0, introduced at creation of each entity, that is nk_0, and then averaging among all the existing n entities:

$$m \simeq \frac{\sum_{i=1}^{n} x_i - nk_0}{n} \tag{5.7}$$

Having estimated m, we are able to estimate c using another important quantity of Yule process, the probability p_{k_0} that an entity has a property value just equal to k_0, which is given by eq. 5.4. p_{k_0} can be easily estimated by measuring the fraction of entities having property value equal to k_0 with respect to the total number of entities. Then, applying eq. 5.4, one can compute the estimated value of c:

$$c = \frac{m(1-p_{k_0})}{(m+1)\,p_{k_0} - 1} - k_0 \tag{5.8}$$

As anticipated, we try to model, using the Yule process, the evolution of some properties of object-oriented software systems. For this reason, the values of the Yule parameters have been estimated for each version of the projects studied. The estimates computed using eqs. 5.7 and 5.8 are cumulated values, i.e. calculated considering the properties and entities from the beginning of the project until the version under study. We found different values of the Yule parameters for each version of the studied projects (see Tables 5.3, 5.6, 5.9, 5.12). These results highlight how the versions are not snapshots of the same Yule process with constant parameters, but seem to be the result of different Yule processes (with different parameters), one for each version. So, the power-law distributions found in Concas et al. [22], [23] on the last version of the systems analysed seem to be the result of a Yule Process where the value of m and c change at every step, a step corresponding to the time interval between two versions of the system, or, equivalently, a Yule process varying in time. In fact, if the same Yule process (with same parameters) were in place from the beginning, each software release would be characterized always by the same parameters, irrespective to the system evolution. Our analysis of the software system development through all the main releases shows instead that a single Yule process cannot explain the power-laws developed in time by the software properties analyzed. Nonetheless, all these properties exhibit, release after release, a patent power-law behavior, that is compatible for each release by a Yule process. Thus, we deem that a more general Yule process, whose parameters change during system development through the different main versions, can be able to model the overall system behavior and its evolution in time.

To further study this subject, we decided to perform a differential analysis, examining the changes of the properties of the system from one main release to the next, and as a consequence the changes of the Yule parameters, in the hypothesis that these parameters stay approximately constant between two system versions. So, we estimated the values of these parameters between two consecutive main versions. We define $m_{\mathbf{i}}$ and $c_{\mathbf{i}}$ as the estimates of m and c respectively, between versions $\mathbf{i}-1$ and $\mathbf{i}$, and $n_{\mathbf{i}}$ as the number of entities of version $\mathbf{i}$. The estimate of $m_{\mathbf{i}}$ is given by:

$$m_{\mathbf{i}} \simeq \frac{(X_{\mathbf{i}} - X_{\mathbf{i\text{-}1}}) - (n_{\mathbf{i}} - n_{\mathbf{i\text{-}1}})k_0}{n_{\mathbf{i}} - n_{\mathbf{i\text{-}1}}} \tag{5.9}$$

which is a variation of eq. 5.7 to account for the fact that now we are estimating m just referring to the increment of the system between version $\mathbf{i}$ and the previous version $\mathbf{i\text{-}1}$. Here $X_{\mathbf{i}} = (\sum_1^n x_i)_{\mathbf{i}}$ is the sum of all the properties of the entities for the version $\mathbf{i}$. The estimate of $c_{\mathbf{i}}$ is still given by eq. 5.8, which for the differential notation becomes:

$$c_{\mathbf{i}} \simeq \frac{m_{\mathbf{i}}(1-p_{k_0\mathbf{i}})}{(m_{\mathbf{i}}+1)p_{k_0\mathbf{i}}-1} - k_0 \tag{5.10}$$

Finally we define:

$$p_{k_i} = \frac{k_i^{-\alpha_i}}{((k_0+c_i+\alpha_i-1)B(k_0+c_i,\alpha_i))} \tag{5.11}$$

as the differential probability distribution among two version.

Note that the formula of eq. 5.4 giving p_{k_0} is valid in the limit $n \to \infty$, so also eqs. 5.8 and 5.10 are approximately valid only for high values of n, and $n_{\mathbf{i}}$ - $n_{\mathbf{i+1}}$, respectively. Consequently, the computation of the differential parameters $p_{k_0\mathbf{i}}$ can be made only for high values of $n_{\mathbf{i}}$ - $n_{\mathbf{i+1}}$, that is in the case of a new version with a number of new entities substantially higher than the previous version. These parameters are relative to just one version, without taking into account the previous ones. From now onwards, we call m and c the cumulated parameters, and $m_{\mathbf{i}}$ and $c_{\mathbf{i}}$ the differential ones.

5.3 Fitting the power-law distributions

Once computed the values of k_0, m and c, we proceeded plotting in log-log scale the survival distribution of the Yule distribution obtained using eq. 5.2 and the empirical data, to evaluate the goodness of the Yule process to model the generation process of such data.

Another possible verification of the goodness of the model is when the tail index α given by eq. 5.5 corresponds to the empirical value obtained by optimal fitting of the cumulative distribution of the property values using the maximum likelihood estimator (MLE), which, for continuous data, is equivalent to the well-known "Hill's estimator" [40] given by the following equation:

$$\hat{\alpha} = 1 + \frac{n}{\sum_{i=1}^{n} ln\frac{x_i}{x_{min}}} \tag{5.12}$$

In this formula, the quantities $x_i, i = 1 \ldots n$ are measured values of the property x, and x_{min} corresponds to the smallest value for which the power-law behavior holds. The standard error on $\hat{\alpha}$ is given by:

$$\sigma = \frac{\hat{\alpha}-1}{\sqrt{n}} + O(1/n) \tag{5.13}$$

where the higher order correction is positive. The MLE for the case where x is a discrete integer variable (our case) is less straightforward. Seal [66] and,

more recently, Goldstein et al. [38] studied the case of $x_{min} = 1$, showing that the solution is given by solving the trascendental equation:

$$\frac{\zeta'(\hat{\alpha})}{\zeta(\hat{\alpha})} = -\frac{1}{n}\sum_{i=1}^{n} lnx_i \tag{5.14}$$

where $\zeta(a)$ is the Zeta function. When $x_{min} \geq 1$, the estimator for α is given by a similar equation with generalized zetas [17], [21]:

$$\frac{\zeta'(\hat{\alpha}, x_{min})}{\zeta(\hat{\alpha}, x_{min})} = -\frac{1}{n}\sum_{i=min}^{n} lnx_i \tag{5.15}$$

In practice, this equation is solved numerically using binary search, or alternatively by numerical maximization of the likelihood function itself, or of its logarithm.

A simpler equation to estimate $\hat{\alpha}$ in the discrete case has been calculated by Clauset et al. [21]:

$$\hat{\alpha} = 1 + \frac{n}{\sum_{i=1}^{n} ln\frac{x_i}{x_{min}-\frac{1}{2}}} \tag{5.16}$$

which is identical to MLE for the continuous case, except for the factor $-\frac{1}{2}$ in the denominator. This equation is known to be a good approximation only for $x_{min} \geq 6$.

To calculate the lower bound x_{min} we used the method proposed by Clauset et al. [21]. We calculated the maximum distance between the CDFs of the data and the fitted model using the following equation:

$$D = \max_{x \geq x_{min}} |S(x) - P(x)| \tag{5.17}$$

where $S(x)$ is the CDF of the data, and $P(x)$ is the CDF for the power-law model that best fits the data in the region $x \geq x_{min}$. The estimate of $\hat{x}_{min}$ is the value of x_{min} that minimizes[2] D.

5.4 Results

In this section, we present in detail the measurements made on the various software systems, present their statistical distributions, and try to statistically model their generation using the modified Yule process described above. We analysed several versions of the following OO software systems: Eclipse [3] (12

[2]The values of $\hat{\alpha}$ and x_{min} reported in this work are obtained using the matlab scripts made available on line by Clauset et al. at: www.santafe.edu/ ~aaronc/powerlaws/.

versions), JDK [5] (11 versions), Netbeans [4] (17 versions), Ant [1] (10 versions). We distinguish between major and minor versions looking at the total number of entities, the values of their properties and at the maximum value of the property (x_{max}), as reported in Table 5.1 .

Table 5.1: *Empirical data computed on each releases for property "Names of methods".*

Releases	Nr. of Entities	Nr. of Properties	x_{max}
Eclipse2.0	27022	68041	536
Eclipse2.1	36863	89411	860
Eclipse2.1.1	36932	89549	860
Eclipse2.1.2	36870	89393	860
Eclipse2.1.3	37082	89833	860
Eclipse3.0	50369	124893	1435
Eclipse3.0.1	50386	124919	1435
Eclipse3.0.2	50530	125174	1435
Eclipse3.1	60611	153010	1714
Eclipse3.1.1	60779	153274	1714
Eclipse3.1.2	60725	153185	1714
Eclipse3.2	71557	181887	1854

We consider five different groups of releases for Eclipse and JDK, and six for Netbeans and Ant, where the number of total entities and properties, as well as the other statistical properties, remain practically unchanged. We choose the first version of each group as the representative of the group.

The evolution of these software systems may be easily studied because programmers use version control systems for the source code, for instance CVS [2] or Subversion [6]. The source code has been analysed with a parser in order to extract, for each version, the properties that we found to follow a power-law distribution. These are: i) the frequency of instance variables names; ii) the frequency of methods names; iii) the number of calls to methods with the same name; iv) the number of immediate subclasses of a given class.

5.4.1 Names of instance variables

The first property we studied is the distribution of instance variables names in each class of the system (including static instance variables and excluding inherited variables). In Fig. 5.1 we plot data for the representative versions of each group.

All distributions show quite a straight behavior, being JDK the less regular. A solid line, corresponding to the Yule distribution obtained using eq. 5.2 for the last version analysed, is shown in the four panes of Fig. 5.1, We reported for the same version also a dashed line with slope corresponding to the value of the α exponent estimated using the MLE for discrete case (see section 5.3).

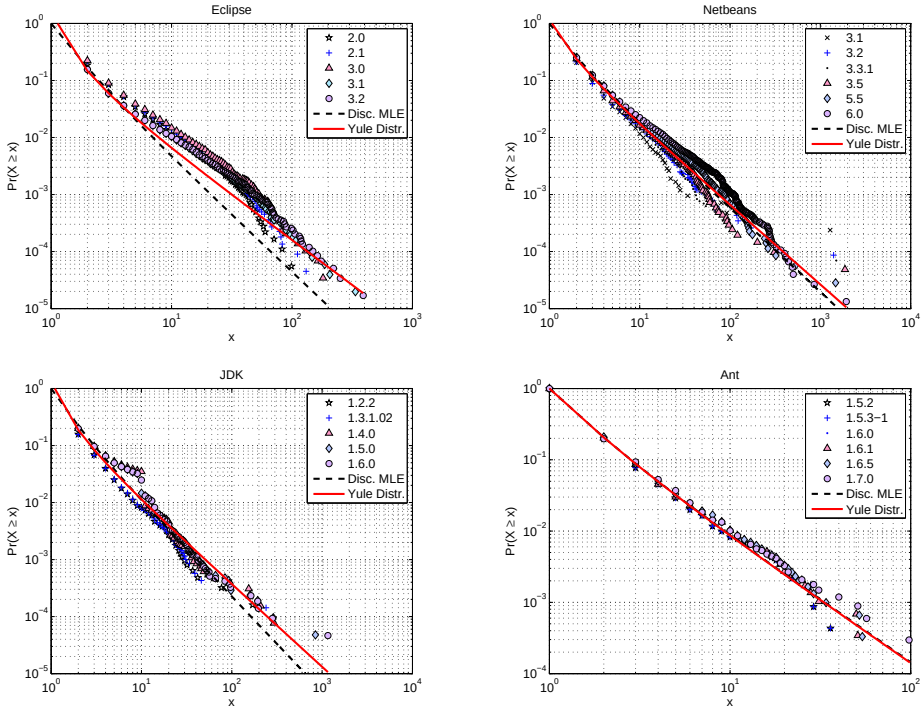


Figure 5.1: *Survival distributions of the names of instance variables for each version of the analysed systems: Eclipse, Netbeans, JDK, Ant. Solid and dashed line represent best fits to the data using the methods described in section 5.3.*

In the modeled Yule process, the entities are the names themselves, while their properties are the number of instance variables in the system having that name. In this case, the preferential attachment growth means, for example, that when a new instance variable is inserted into a class, it is most likely that the name be the same of an already existing instance variable .In fact, it is good object-oriented programming style to assign the same name to variables containing the same kind of data. On the other hand, it is also possible that a new name is used, generating a new entity with its property set to one.

In Table 5.2 we show the number of different names for instance variables (Entities), and the total number of instance variables (Nr. Properties) for each main version representative of the systems studied. As you can see, all these systems include thousands, or tens of thousand, of names and variables, and can be consequently studied under a statistical perspective.

We estimated the cumulated and differential Yule parameters ($m, c, m_{\mathbf{i}}$, and $c_{\mathbf{i}}$), for each version and for each property examined using the equations reported in section 5.2. The values of k_0, as mentioned above, is determined by the model itself. In this case $k_0 = 1$ because when a new name is added, it refers

Table 5.2: *Empirical data computed on each project for property "Names of instance variables".*

Project	Nr. Entities	Nr. Properties	Project	Nr. Entities	Nr. Properties
Eclipse2.0	18158	28404	JDK1.2.2	6249	9027
Eclipse2.1	22188	36410	JDK1.3.1-02	6988	10205
Eclipse3.0	29142	49706	JDK1.4.0	12985	22980
Eclipse3.1	50522	75598	JDK1.5.0	20933	37143
Eclipse3.2	59072	89122	JDK1.6.0	21512	38612
Netbeans3.1	4205	7876	Ant1.5.2	2306	3426
Netbeans3.2	11585	21089	Ant1.5.3-1	3433	2310
Netbeans3.3.1	14409	25761	Ant1.6.0	2887	4402
Netbeans3.5	20608	39393	Ant1.6.1	2917	4467
Netbeans5.5	35323	72502	Ant1.6.5	3026	4709
Netbeans6.0	75466	164900	Ant1.7.0	3374	5302

to just one instance variable. In Table 5.3 we report the Yule parameters calculated for the main versions of each project studied.

Table 5.3: *Yule process parameters computed on each project for property "Names of instance variables".*

Project	m	$m_{\mathbf{i}}$	c	$c_{\mathbf{i}}$	Project	m	$m_{\mathbf{i}}$	c	$c_{\mathbf{i}}$
Eclipse2.0	0.564	0.564	-0.617	-0.617	JDK1.2.2	0.445	0.445	-0.676	-0.676
Eclipse2.1	0.641	0.987	0.500	0.458	JDK1.3.1-02	0.460	0.594	-0.692	-1.632
Eclipse3.0	0.706	0.912	-0.514	-0.528	JDK1.4.0	0.770	1.130	-0.626	-3.787
Eclipse3.1	0.496	0.211	-0.702	-0.893	JDK1.5.0	0.774	0.782	-0.630	-2.188
Eclipse3.2	0.509	0.582	-0.709	-0.745	JDK1.6.0	0.795	1.537	-0.644	-2.598
Netbeans3.1	0.873	0.873	-0.635	-0.635	Ant1.5.2	0.488	0.486	-0.505	-0.505
Netbeans3.2	0.820	0.790	-0.598	-0.574	Ant1.5.3-1	0.486	0.750	-0.508	-1.000
Netbeans3.3.1	0.788	0.654	-0.586	-0.517	Ant1.6.0	0.525	0.679	-0.497	-0.435
Netbeans3.5	0.911	1.199	-0.513	-0.308	Ant1.6.1	0.531	1.167	-0.509	-0.877
Netbeans5.5	1.052	1.250	-0.494	-0.459	Ant1.6.5	0.556	1.220	-0.494	0.192
Netbeans6.0	1.185	1.302	-0.531	-0.557	Ant1.7.0	0.571	0.704	-0.568	- 0.882

These results highlight how the Yule parameters change for different versions, meaning that it is not possible to model the evolution of the project properties using a Yule process with constant parameters. At the same time, the value of α calculated under the hypothesis of a Yule process at work is consistent with the power-law exponent extracted from fitting the data. As already discussed, we introduce a model able to explain the evolution of these systems based on a Yule process, where the value of m and c change at every step. For instance, the Eclipse project could be modeled with a Yule process having five steps (one for each main version examined); for each step, the corresponding parameters $m_{\mathbf{i}}$ and $c_{\mathbf{i}}$ are reported in Table 5.3. In order to better investigate this assumption we used a Yule process simulator with the ability to vary m and c during the process (see section 5.5).

Table 5.4 reports the α exponent computed using eq. 5.5 (α Yule) and using

eq. 5.15 (α MLE). From the software engineering perspective, these variables need some interpretation. First, note that the smaller is α, the more spread is the statistical distribution. Typical values for α are in the range 2-3. This means that the larger is m the spreader are the distributions. This is reflected in the software system by a higher reuse of existing names or sub-classes with respect to the introduction of new variables names, new methods names and new sub-classes. When comparing the Yule and MLE exponent, there is a very good agreement for Ant and Netbeans projects. In Eclipse and JDK the agreement is lower, but looking at Fig. 5.1, we may observe that the fit with real data looks better using the estimated Yule distribution than the fitting obtained using the exponent computed with the MLE. Note that our fitting uses equation 5.2, which is not a perfect straight line in the log-log plots.

Table 5.4: *The scaling parameter α computed using the Yule process and the MLE method on the last version of each system for property "Name of instance variables".*

	Eclipse	JDK	Netbeans	Ant
α (Yule)	2.57	2.45	2.39	2.75
α (MLE)	2.99	2.66	2.47	2.75
x_{min}	1	1	1	1

5.4.2 Names of methods

The second property we studied is the distribution of the names of methods of the system (excluding Java constructors). Note that it is good OOP style to give the same name to methods performing the same operation, in different classes.

In Fig. 5.2 we show the survival distributions for the main versions of the four system examined. All distributions show a better straight behavior and fitting to the Yule distribution than the previous ones, being JDK the less regular, with a more curved slope in the tail. We reported the Yule distribution fitting (using eq. 5.2) only for the last version, that corresponds to the end of the estimated Yule process whose parameters change at every step. The dashed line represents the best data fit using the MLE estimate for the discrete case.

Table 5.5 shows the number of unique methods names (Nr. of Entities) and the total number of methods (Nr. of Properties) for each version of the four systems considered.

Applying to methods names a Yule process similar to that devised for instance variables names, we have again $k_0 = 1$. Using the fitting method described in section 5.1, we estimated the values of m and c. In Table 5.6 we report these results and also the values of $m_{\mathbf{i}}$ and $c_{\mathbf{i}}$, estimated using eqs. 5.9 and 5.10, for each system studied.

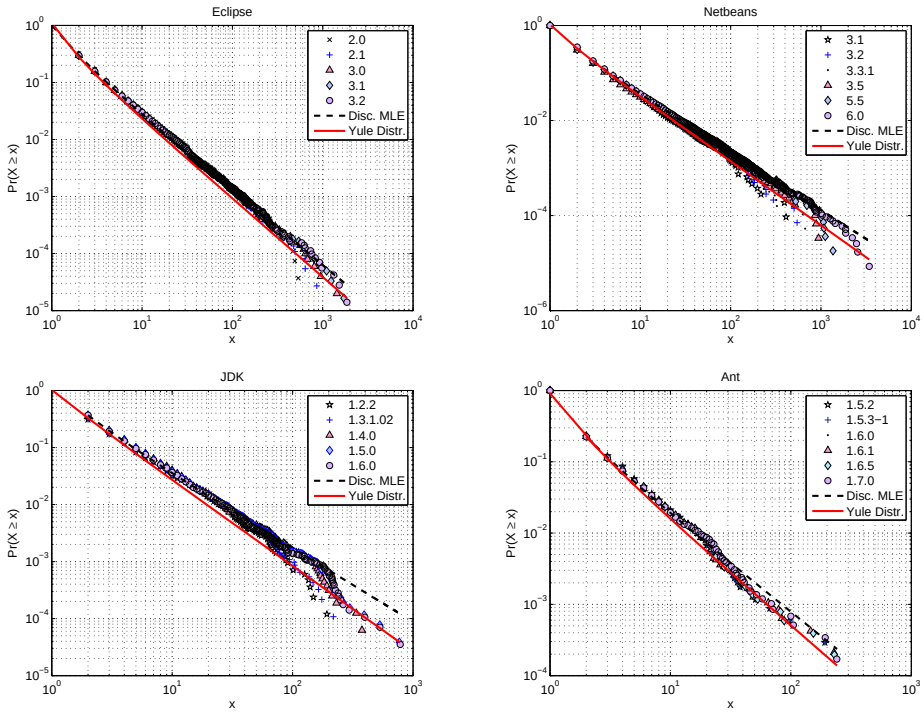


Figure 5.2: *Survival distributions of the frequency of a method name for each analysed system: Eclipse, Netbeans, JDK and Ant. Solid and dashed line represent best fits to the data using the methods described in section 5.3.*

Table 5.5: *Yule process parameters computed on each project for property "Names of methods".*

Project	Nr. Entities	Nr. Properties	Project	Nr. Entities	Nr. Properties
Eclipse2.0	27022	68041	JDK1.2.2	8404	20591
Eclipse2.1	36863	89411	JDK1.3.1-02	9247	23068
Eclipse3.0	50369	124893	JDK1.4.0	16008	43806
Eclipse3.1	60611	153010	JDK1.5.0	25953	72643
Eclipse3.2	71557	181887	JDK1.6.0	28417	78054
Netbeans3.1	10734	27271	Ant1.5.2	3421	6516
Netbeans3.2	14131	36712	Ant1.5.3-1	3426	6524
Netbeans3.3.1	18868	49159	Ant1.6.0	4691	8876
Netbeans3.5	30019	78970	Ant1.6.1	4736	8980
Netbeans5.5	55416	160788	Ant1.6.5	5085	9665
Netbeans6.0	117369	363043	Ant1.7.0	5870	11216

Also in this case the Yule process parameters m and c change with the version, but their values are much closer to each other. For this reason, the distributions look to be almost overlapping, their slope being related to the parameters m and c by the equation 5.5.

Table 5.6: *Yule process parameters computed on each project for property "Names of methods".*

Project	m	m_i	c	c_i	Project	m	m_i	c	c_i
Eclipse2.0	1.518	1.518	-0.314	-0.314	JDK1.2.2	1.450	1.450	-0.340	-0.340
Eclipse2.1	1.425	1.171	-0.397	-0.587	JDK1.3.1-02	1.495	1.938	-0.312	0.030
Eclipse3.0	1.480	1.627	-0.405	-0.423	JDK1.4.0	1.736	2.067	-0.215	-0.059
Eclipse3.1	1.524	1.745	-0.412	-0.434	JDK1.5.0	1.799	1.870	-0.115	0.072
Eclipse3.2	1.542	1.638	-0.419	-0.456	JDK1.6.0	1.747	1.196	-0.121	-0.131
Netbeans3.1	1.541	1.541	-0.308	-0.308	Ant1.5.2	0.905	0.905	-0.575	-0.575
Netbeans3.2	1.598	1.779	-0.401	-0.614	Ant1.5.3-1	0.904	0.600	-0.575	-0.571
Netbeans3.3.1	1.605	1.628	-0.400	-0.396	Ant1.6.0	0.892	0.859	-0.560	-0.513
Netbeans3.5	1.631	1.673	-0.400	-0.400	Ant1.6.1	0.896	1.311	-0.564	-0.826
Netbeans5.5	1.901	2.222	-0.407	-0.407	Ant1.6.5	0.901	0.963	-0.556	-0.443
Netbeans6.0	2.093	2.265	-0.275	-0.136	Ant1.7.0	0.911	0.976	-0.575	-0.678

In all systems there is good agreement between the empirically found values of α and the values computed using the Yule model, as shown in Table 5.7, for the last version of the four systems considered.

Table 5.7: *The scaling parameter α computed using the Yule process and the MLE method on the last version of each system for property "Name of methods".*

	Eclipse	JDK	Netbeans	Ant
α (Yule)	2.38	2.50	2.35	2.47
α (MLE)	2.31	2.29	2.21	2.36
x_{min}	1	2	3	2

5.4.3 Number of calls to methods

Another property we studied is the distribution of how many times a method name is called (excluding the invocation of Java constructors). This is an indicator of how much a set of methods with the same name is used in the system. The survival distributions for the main versions of the four systems examined are shown in Fig. 5.3. The distributions show again a quite straight behavior and a quite good fitting with the Yule distribution obtained using eq. 5.2 with the value parameters of the last version (solid line). The fitting using the MLE exponent (dashed line) looks better only for the Ant project.

Table 5.8 shows the total number of names called (Nr. of Entities), the total number of calls (Nr. of Properties) for each version of the system analysed.

Trying to apply the Yule process to the addition of new method calls, we first observe that the entities are again the names themselves. In this case, they do not refer to the call of a specific method, but to the call of the whole family of methods with the same name. The considered property is the number of

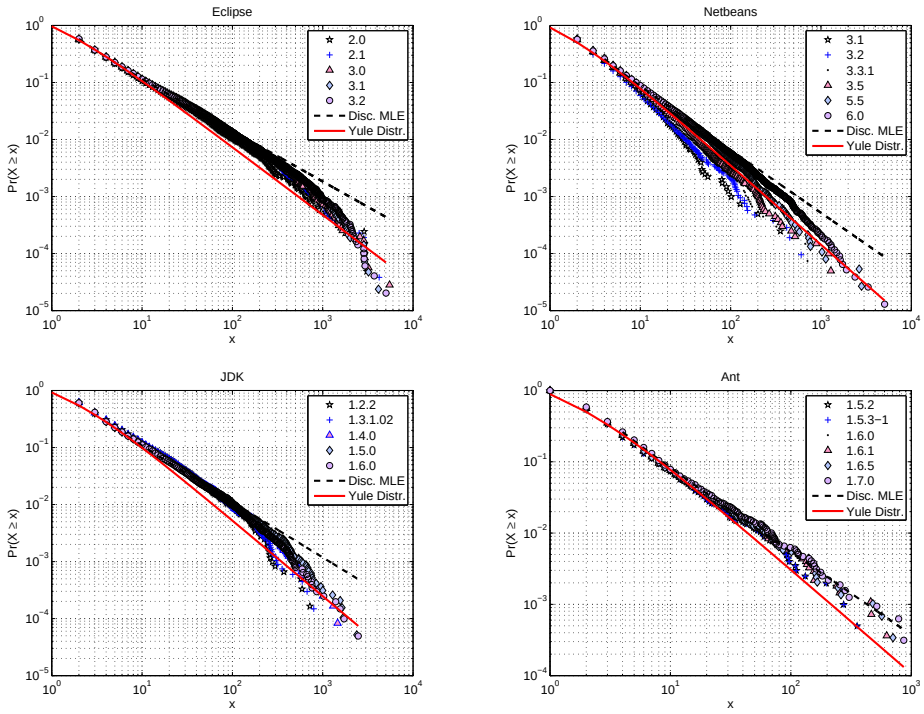


Figure 5.3: *Survival distributions of the number of method calls for each analysed system: Eclipse, Netbeans, JDK and Ant. Solid and dashed line represent best fits to the data using the methods described in section 5.3.*

Table 5.8: *Empirical data computed on each project for property "Number of call to methods".*

Project	nr. Entity	nr. Property	Project	nr. Entity	nr. Property
Eclipse2.0	20595	168072	JDK1.2.2	6072	41479
Eclipse2.1	26208	241607	JDK1.3.1-02	6728	47079
Eclipse3.0	35514	327145	JDK1.4.0	12043	90145
Eclipse3.1	42151	403713	JDK1.5.0	19284	148490
Eclipse3.2	49365	480202	JDK1.6.0	20237	146730
Netbeans3.1	4052	14691	Ant1.5.2	2014	8959
Netbeans3.2	10516	37968	Ant1.5.3-1	2018	8970
Netbeans3.3.1	13511	52286	Ant1.6.0	2735	13442
Netbeans3.5	20126	88481	Ant1.6.1	2759	13671
Netbeans5.5	37482	194580	Ant1.6.5	2932	14835
Netbeans6.0	77350	440741	Ant1.7.0	3193	17556

times they are called in the system. Setting $k_0 = 1$ for the call of a name that was never called before and applying the fitting method reported in section 5.1, we estimated the values of m and c. Table 5.9 shows these values, together with the values of $m_\mathbf{i}$ and $c_\mathbf{i}$.

We report in the first rows of Table 5.10 the estimated scaling parameters α

Table 5.9: *Yule process parameters computed on each project for property "Number of call to methods".*

Project	m	$m_\mathbf{i}$	c	$c_\mathbf{i}$	Project	m	$m_\mathbf{i}$	c	$c_\mathbf{i}$
Eclipse2.0	7.161	7.161	0.647	0.647	JDK1.2.2	5.831	5.831	1.096	1.096
Eclipse2.1	8.219	12.101	0.580	0.427	JDK1.3.1-02	5.997	7.537	1.139	1.640
Eclipse3.0	8.212	8.192	0.604	0.673	JDK1.4.0	6.485	7.103	1.119	1.104
Eclipse3.1	8.578	10.54	0.615	0.691	JDK1.5.0	6.700	7.058	0.979	0.779
Eclipse3.2	8.728	9.603	0.653	0.898	JDK1.6.0	6.251	-2.847	1.059	0.197
Netbeans3.1	2.626	2.626	1.900	1.900	Ant1.5.2	3.448	3.448	0.804	0.804
Netbeans3.2	2.610	2.601	1.084	0.736	Ant1.5.3-1	3.445	1.750	0.793	-1.000
Netbeans3.3.1	2.870	3.781	0.978	0.791	Ant1.6.0	3.915	5.237	0.871	1.205
Netbeans3.5	3.396	4.472	1.058	1.361	Ant1.6.1	3.955	8.542	0.885	5.845
Netbeans5.5	4.191	5.113	1.097	1.222	Ant1.6.5	4.060	5.728	0.858	0.594
Netbeans6.0	4.698	5.174	0.843	0.662	Ant1.7.0	4.498	9.425	1.073	33.862

using the Yule process and the discrete MLE method for the last version. The agreement between the values computed using the Yule model and the empirically found values of α is lower with respect to the other properties analyzed. We may observe that also in this case the fitting with real data looks better using the Yule distribution, as we can see in Fig. 5.3, except for the Ant project.

Table 5.10: *The scaling parameter α computed using the Yule process and the MLE method on the last version of each system for property "Number of call to methods".*

	Eclipse	JDK	Netbeans	Ant
α (Yule)	2.19	2.33	2.39	2.46
α (MLE)	1.89	1.97	2.10	2.16
x_{min}	2	4	4	4

5.4.4 Number of subclasses

The last property we studied is the number of immediate subclasses of each class of the system.

The survival distributions for the main versions of the four systems examined are reported in Fig. 5.4. All distributions show again a straight behavior, with irregularities in the extreme tail and a good fitting with the Yule distribution for the last version.

We report in Table 5.11 the number of classes (Nr. of Entities) and subclasses (Nr. of properties) in each system.

Now, let us discuss the Yule process as a possible stochastic mechanism able to generate these power-laws. Clearly, the key mechanism is related to immediate subclasses generation, because new classes are usually added to the system

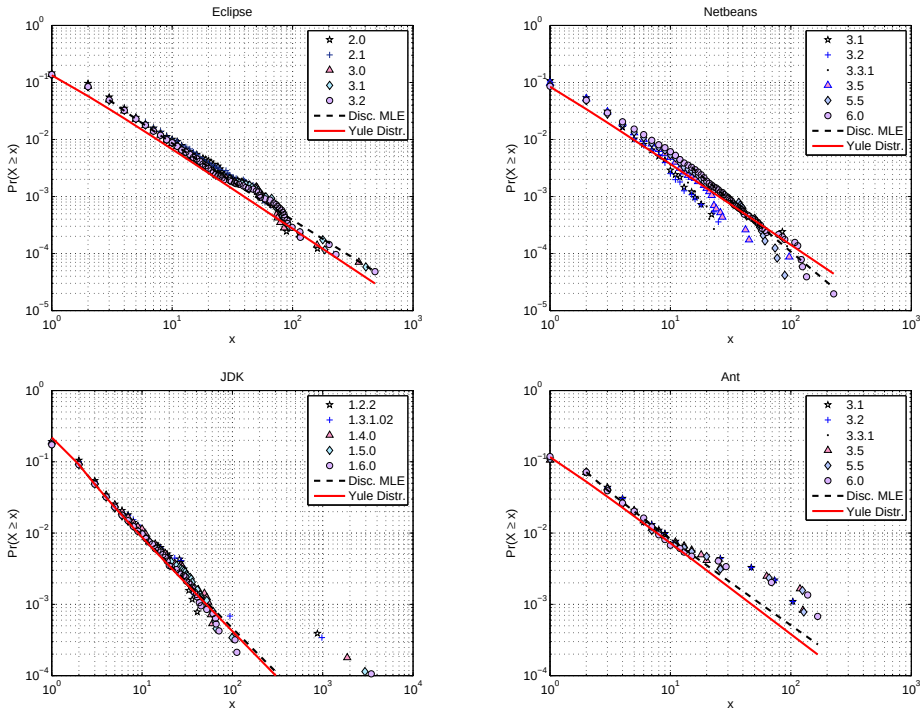


Figure 5.4: *Survival distributions of the number of immediate subclasses for each system analysed: Eclipse, Netbeans, JDK and Ant. Solid and dashed line represent best fits to the data using the methods described in section 5.3.*

Table 5.11: *Empirical data computed on each project for property "Number of subclasses".*

Project	nr. Entity	nr. Property	Project	nr. Entity	nr. Property
Eclipse2.0	8221	4783	JDK1.2.2	2560	2375
Eclipse2.1	10113	5839	JDK1.3.1-02	2925	2729
Eclipse3.0	14193	7746	JDK1.4.0	5588	5283
Eclipse3.1	17174	9451	JDK1.5.0	8785	7978
Eclipse3.2	20802	11524	JDK1.6.0	9433	8731
Netbeans3.1	4140	1135	Ant1.5.2	913	519
Netbeans3.2	5563	1471	Ant1.5.3-1	915	520
Netbeans3.3.1	7413	2061	Ant1.6.0	1202	704
Netbeans3.5	11478	3274	Ant1.6.1	1208	709
Netbeans5.5	24092	7053	Ant1.6.5	1280	743
Netbeans6.0	51031	16100	Ant1.7.0	1479	893

one by one, as a subclass of an existing class. If a Yule process is assumed, this should necessarily have $k_0 = 0$, because generally a new added class has no subclasses [3]. We also estimated the values of m and c applying the fitting

[3]It might happen that in a refactoring operation we extract a common superclass from two

method reported in section 5.1 and the values of $m_{\mathbf{i}}$ and $c_{\mathbf{i}}$ using eqs. 5.9 and 5.10, respectively.

Table 5.12 shows the results for the four systems studied. Also for this property the distributions seem to be almost overlapping because the values of the Yule parameter m for each version are very similar.

Table 5.12: *Yule process parameters computed on each project for property "Number of subclasses".*

Project	m	$m_{\mathbf{i}}$	c	$c_{\mathbf{i}}$	Project	m	$m_{\mathbf{i}}$	c	$c_{\mathbf{i}}$
Eclipse2.0	0.582	0.582	0.244	0.244	JDK1.2.2	0.928	0.928	0.319	0.319
Eclipse2.1	0.577	0.558	0.221	0.136	JDK1.3.1.02	0.933	0.970	0.312	0.266
Eclipse3.0	0.546	0.467	0.229	0.257	JDK1.4.0	0.945	0.959	0.316	0.320
Eclipse3.1	0.550	0.572	0.219	0.175	JDK1.5.0	0.908	0.843	0.279	0.221
Eclipse3.2	0.554	0.571	0.226	0.260	JDK1.6.0	0.926	1.162	0.272	0.193
Netbeans3.1	0.274	0.274	0.214	0.214	Ant1.5.2	0.568	0.568	0.157	0.157
Netbeans3.2	0.264	0.236	0.200	0.163	Ant1.5.3-1	0.568	0.500	0.156	0.000
Netbeans3.3.1	0.278	0.319	0.205	0.220	Ant1.6.0	0.586	0.641	0.150	0.130
Netbeans3.5	0.285	0.298	0.174	0.129	Ant1.6.1	0.587	0.833	0.148	0.000
Netbeans5.5	0.293	0.300	0.146	0.124	Ant1.6.5	0.580	0.472	0.156	0.347
Netbeans6.0	0.315	0.336	0.134	0.125	Ant1.7.0	0.604	0.754	0.173	0.298

The values of α computed using the Yule model and estimated with the Maximum Likelihood Estimator (MLE) are quite similar in all the four system studied (see Table 5.13).

Table 5.13: *The scaling parameter α computed using the Yule process and the MLE method on the last version of each system for property "Number of subclasses".*

	Eclipse	JDK	Netbeans	Ant
α (Yule)	2.41	2.29	2.42	2.29
α (MLE)	2.32	2.27	2.77	2.19
x_{min}	3	2	15	2

The distribution of the number of immediate subclasses in Java systems has been studied by Wheeldon and Counsell [81], who also report a power-law behavior.

5.5 Simulation results

In this section we use the simulation approach to verify our assumptions. More precisely, we want to assess if a Yule process, whose parameter m and c change

or more existing classes. This operation would yield the introduction of a new class having two or more subclasses, thus with $k_0 > 1$. In our model, we consider these refactoring activities rare, and thus neglect them.

at every step, as we computed from empirical data, is able to statistically model the power-laws found.

To simulate the Yule process described above, the simulator must be able to use as input the values empirically found. At the beginning of the process, there is just one entity, with property value set to k_0. At each time step there is a constant probability $a = \frac{1}{m+1}$ that a new entity is created, and a probability $1 - a$ that the value of an existing entity is increased by one. Let us remember that m is the average number of property increment in between the addition of two new entities. Consequently, m will be in general a fraction value. We consider a number of steps equal to the number of main versions of the project under study. The values of m, c and k_0 are kept constant during each step (i.e until the number of entities generated is equal to the number of entities of the considered version), and are changed in the next step, except for k_0.

For the sake of brevity, in the followings we report the simulation results only for Eclipse. Similar results have been obtained for all other systems. We studied five versions (steps) for Eclipse. Tables 5.3, 5.6, 5.9, 5.12, report the Yule parameters that we used in the simulation. More precisely, we set the parameters of the Yule process simulator at the corresponding values of $m_{\mathbf{i}}$ and $c_{\mathbf{i}}$ for each of the five steps. At the end of the simulation, in order to evaluate the goodness of this Yule process for modeling the studied phenomenon, we calculate the cumulated parameters m and c of the five snapshots corresponding to the five versions of the Eclipse project. Tables 5.14 and 5.15 show the simulation results of the cumulated parameters m and c, compared with those calculated from real data.

Table 5.14: *Comparison between the parameter m obtained from real data and simulation results (averaged over 20 runs) for each of the Eclipse properties examined. Standard errors of simulated parameters are reported in parenthesis.*

Variable Names		Method Names		Calls to Methods		Nr. Subclasses	
m Real	m Sim.	m Real	m Sim.	m Real	m Sim.	m Real	m Sim.
0.564	0.562 (0.002)	1.518	1.521 (0.002)	7.161	7.184 (0.011)	0.582	0.581 (0.002)
0.641	0.639 (0.002)	1.425	1.428 (0.002)	8.219	8.233 (0.012)	0.577	0.576 (0.002)
0.706	0.703 (0.001)	1.480	1.482 (0.002)	8.212	8.215 (0.009)	0.546	0.546 (0.002)
0.496	0.495 (0.001)	1.524	1.527 (0.002)	8.578	8.579 (0.009)	0.550	0.549 (0.002)
0.509	0.508 (0.001)	1.542	1.544 (0.002)	8.728	8.729 (0.007)	0.554	0.553 (0.002)

We found a very good correspondence for m, meaning that the estimate of m is reliable all over the phases of all examined projects. On the other hand, parameter c often exhibits a poor correspondence between values estimated from real data, and the results of a typical simulation, made using parameters estimated between consecutive versions. Note that m is the more important parameter, being directly related to the programmers propension to reuse ex-

Table 5.15: *Comparison between the parameter c obtained from real data and simulation results (averaged over 20 runs) for each of the Eclipse properties examined. Standard errors of simulated parameters are reported in parenthesis.*

Variable Names		Method Names		Calls to Methods		Nr. Subclasses	
c Real	*c* Sim.	*c* Real	*c* Sim.	*c* Real	*c* Sim.	*c* Real	*c* Sim.
-0.617	-0.619 (0.003)	-0.314	-0.313 (0.002)	0.647	0.648 (0.005)	0.244	0.244 (0.017)
-0.500	-0.511 (0.003)	-0.397	-0.389 (0.002)	0.580	1.073 (0.005)	0.221	0.225 (0.015)
-0.514	-0.417 (0.003)	-0.405	-0.376 (0.001)	0.604	0.797 (0.004)	0.229	0.106 (0.010)
-0.702	-0.662 (0.001)	-0.412	-0.364 (0.001)	0.615	0.902 (0.004)	0.219	0.145 (0.010)
-0.709	-0.664 (0.001)	-0.419	-0.371 (0.001)	0.653	0.901 (0.005)	0.226	0.179 (0.010)

isting names, calls and superclasses, and hence to adhere to OO best practices. The fact that the estimate of m throughout the process looks very stable, and consequently reliable, is positive. The meaning of c parameter is fuzzier, it being essentially a tool enabling a better fit of real data to Yule model. c tunes the probability to choose entities with respect to the existing value of their property. The high variation of c are not unexpected, given the difficulty to estimate it.

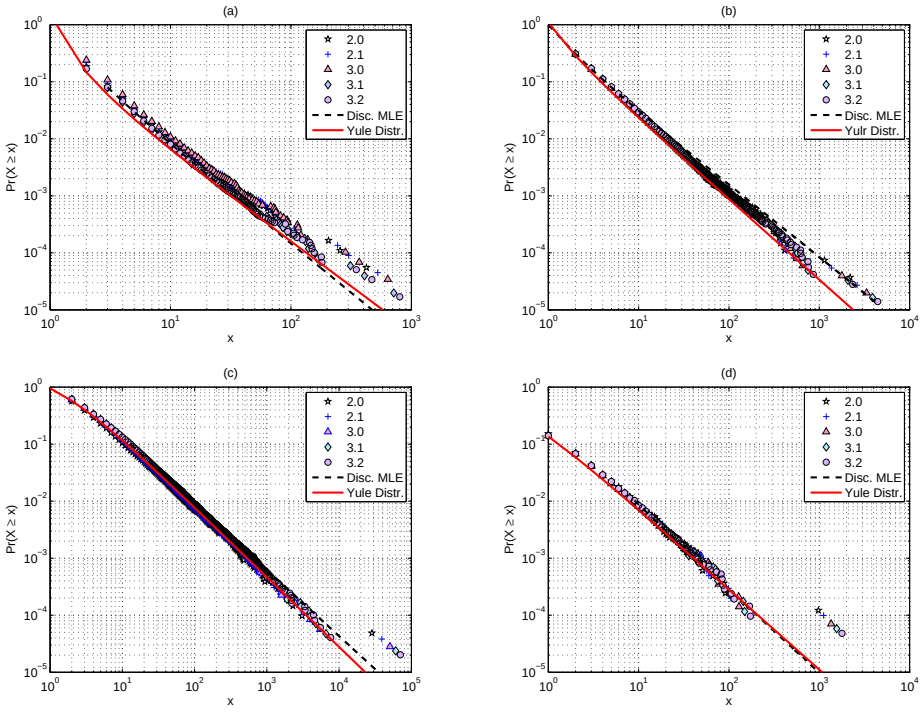


Figure 5.5: *Cumulative distributions obtained simulating the four properties analysed of Eclipse: (a) Names of instance variables, (b) Names of methods, (c) Number of call to methods, (d) Number of subclasses.*

Note that c is consistently negative in the case of the choice of variable and method names, meaning that names with just one occurrence in the system are less likely to be chosen with respect to a "classical" Yule process. In the case of method calls, c is consistently positive, meaning the opposite, namely a higher programmers propension to choose to call methods already called just once, or a few times. In the case of subclasses, c must be positive, because $k_0 = 0$ and if c was lesser than, or equal to zero, it would be impossible to choose as superclass a class with no subclasses. The distributions obtained simulating the five version of Eclipse for the properties studied are shown in Fig. 5.5. There is a good correspondence between these distributions and those obtained from empirical data. In fact, the slope in both kind of distributions is very similar. Obviously, the distributions obtained from simulation are more regular and also the Yule distribution fitting (eq. 5.2) is better. As mentioned above, the irregularities of the real distributions are probably due to the activity of deleting and refactoring classes.

We report in Table 5.16, for simulation data, the α exponent of the Yule model described above and that calculated using the methods of maximum likelihood. The results are quite close to each other, resembling those reported in section 5.5 for real data. Note that these results are relative to a specific simulation.

Table 5.16: *The scaling parameter α computed using the Yule process and the MLE method on the last version of each system for all the studied properties.*

	Variables Names.	methods Names	Call methods	Nr. of Subclasses
α (Yule)	2.65	2.41	2.22	2.40
α (MLE)	2.72	2.27	2.16	2.42
x_{min}	3	1	13	9

Finally, we also investigated the distribution of the differences between the properties of two main consecutive versions, i.e obtained by subtracting, one by one, the number of occurrences of every property between two main versions.

In Fig. 5.6 we report the result for the property "name of methods" in case of simulation (a) and real data (b). We do not plot the statistical distributions of the differences among all the possible couples of versions, but only a few representatives, together with the data for two main releases, to see how the slope remains the same, within the limits of statistical errors. Comparing the results of the differential analysis for the simulated model and for real data we see that the latter shows the same behavior, in the limit of statistical errors. Similar results are obtained for all other properties and all projects studied.

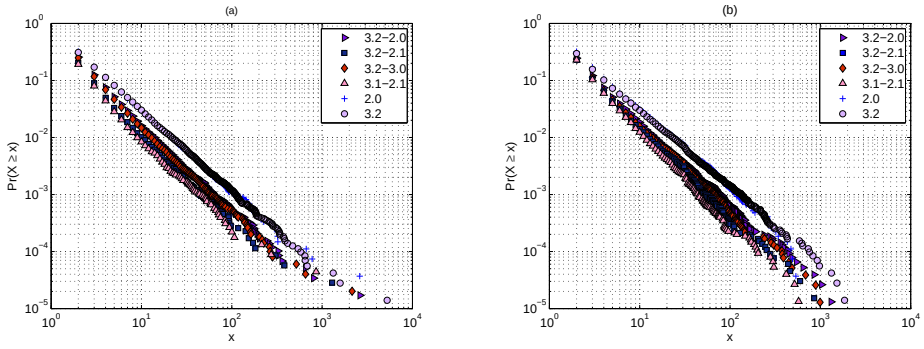


Figure 5.6: *Survival distributions of the differences of methods names frequences among couples of distribution from simulation data (a) and real data (b). We plot only few representative couples, toghether with the survival distributions of two main releases as comparison.*

5.6 Discussion

Each of the properties analyzed show systematically a power-law behavior in different OO software systems, and through different releases or versions. A single Yule model, for each software system, cannot take into account for the overall software evolution process through all the versions, since the Yule parameters calculated for each release differ from each other, even if this difference is small. The introduction of a Yule model whose parameters vary with time, on the other hand, is able to account for all the power-laws exhibited by the system properties. This can also explain little mismatches from a perfect power-law behavior shown by empirical data.

Note that the power law exponents for the cumulative distributions provide themselves with useful information about the software systems, regardless of the existence of an underlying Yule process. For instance, following, for example, the reasoning reported in [48], if some property x is distributed among entities y according to a power-law of the kind $y \simeq x^{-\alpha}$, from the knowledge of α one can easily determine the fraction of entities containing a pre-determined amount of total properties. Thus, one can rank the entities and select this fraction to recover the desired amount of total properties (see [48] for more details). Moreover, we know that, if data distribution follow a power-law, an estimate of the most likely maximum value of a property, x_{max}, can be inferred from the total number of entities, n using formula $x_{max} \propto n^{1/(\alpha-1)}$ [57]. This property can be valuable to forecast extreme values, which can be critical in software development.

The Yule process allows us to infer information about the power-law expo-

nent without a fitting procedure, starting from the data, and avoiding crtitical issues related to the fitting. For instance, estimating α through the Hill MLE procedure introduces a cut-off on the data from the value x_{min}, which artificially becomes a critical parameter itself. We know that any other fitting procedure introduces similar biases. The assumption of a Yule process varying in time relates the exponent of the power-laws exhibited by the system properties to general features of the process of software production, like k_0 or m, that can be easily computed or estimated.

Monitoring the variation of these Yule parameters can help the developers to check if the process of software production is on track. Similar values of m in all releases denotes that the same policy of reuse of method names, or instance variable names is adopted throughout the process, while strong variations of m denote a possible issue on naming policies. The properties we studied are related to software quality. The number of immediate subclasses corresponds to the NOC metric of Chidamber and Kemerer quality metrics suite [19], while the other metrics reflect the effectiveness of naming conventions, as said above. Thus, the information about their distributions can be directly translated into information about the evolution with different releases of parameters related to software quality.

Note that, as almost always in software metrics, there is no a specific value for a metrics, able to qualify a system as "good". It is possible, however, to control the evolution of a large system to assess whether some key metrics are under control. For instance, it is considered good OO programming practice to name in the same way variables and methods having the same meaning in different classes. This should result in a system having a few names very used throughout the system, while retaining the property that most names are used just once, or very few times. As regards subclassing, while in some cases it is sensible to have superclasses with very many subclasses (think for instance to the superclass of all supported devices), it is known that the overuse of subclassing in place of composition is a typical design error made by unexperienced OO programmers. As a large system grows, it is natural that the number of different variable and method names, or of the number of subclasses of existing classes grow. Since such metrics are known to follow a power-law, it is difficult to assess whether the growth of these names, or of the number of subclasses, is within natural limits or it is too fast, using only simple statistics such as the mean and standard deviation. Using power-law theory and stochastic models such as those presented in this paper, on the other hand, it is possible to better control if the values of these parameters tend to stay limited, or tend to increase above an acceptable level. In the latter case, actions might be taken to take the system under control again.

As an example of the use of our model, it is possible to directly measure

the values of the incremental parameter m_i, for instance for the distribution of method names. Having an history of the evolution of of m_i, it is possible to measure its value for a new version of the system. Since m_i is the average number of method names reused in between the addition of two new method names, a reduction of m_i means that method names tend to be reused less than in previous versions. If this reduction is high enough, corrective actions can be taken to verify if the problem is actually due to worsening programming practices regarding method names, and if it is the case, to ask programmers to revert to proper naming guidelines.

We had to limit our study to only four important aspects of software development. However, from our preliminary studies there is evidence that many other properties of software designs that exhibit a power-law behavior can be modeled with the same approach. Among these we may quote in-link distribution of the class graph, and number of bugs per class or per source file.

Other properties of software systems, on the other hand, follow distributions patently different from power-law, such as the number of methods of a class, the number of lines of code of classes and methods, the number of out-links of class graph node [23]. For these properties, the proposed model is clearly not suitable. Another OO metrics, such as the depth of inheritance tree (DIT) of classes [19] has maximum value always less than 11 in the studied systems, so its statistics is not relevant under the perspective of power-law distribution.

Appendix

In this appendix we report, for all main version of the examined software systems and for all the four studied properties, the entities with the highest value of their properties. This gives an idea of the ”fatness“ of their distribution tails.

Table 5.17: *Instance variable names with the maximum number of occurrences for the main version of the examined software systems*

Project	Instance Variable name	Occurrences
Eclipse2.0	name	99
Eclipse2.1	name	131
Eclipse3.0	RESOURCE-BUNDLE	181
Eclipse3.1	serialVersionUID	336
Eclipse3.2	serialVersionUID	390
Netbeans3.0	serialVersionUID	1189
Netbeans3.1	serialVersionUID	1285
Netbeans3.2	serialVersionUID	1404
Netbeans3.3.1	serialVersionUID	1501
Netbeans3.5	serialVersionUID	1873
Netbeans5.5	serialVersionUID	1477
Netbeans6.0	serialVersionUID	1937
Ant1.5.2	project	36
Ant1.5.3-1	project	36
Ant1.6.0	project	51
Ant1.6.1	project	51
Ant1.6.5	name	54
Ant1.7.0	FILE-UTILS	98
JDK1.2.2	serialVersionUID	170
JDK1.3.1.02	serialVersionUID	240
JDK1.4.0	serialVersionUID	293
JDK1.5.0	serialVersionUID	843
JDK1.6.0	serialVersionUID	1160

Table 5.18: *Method names with the maximum number of occurrences for the main version of the examined software systems*

Project	Method Name	Occurrences
Eclipse2.0	visit(1)	536
Eclipse2.1	visit(1)	860
Eclipse3.0	visit(1)	1435
Eclipse3.1	visit(1)	1714
Eclipse3.2	visit(1)	854
Netbeans3.1	getHelpCtx(0)	411
Netbeans3.2	getHelpCtx(0)	544
Netbeans3.3.1	getHelpCtx(0)	668
Netbeans3.5	getName(0)	939
Netbeans5.5	getName(0)	1359
Netbeans6.0	visit(1)	3427
Ant1.5.2	execute(0)	192
Ant1.5.3-1	execute(0)	192
Ant1.6.0	execute(0)	242
Ant1.6.1	execute(0)	228
Ant1.6.5	execute(0)	230
Ant1.7.0	execute(0)	241
JDK1.2.2	toString(0)	193
JDK1.3.1.02	toString(0)	218
JDK1.4.0	toString(0)	377
JDK1.5.0	toString(0)	770
JDK1.6.0	toString(0)	783

Table 5.19: *Maximum number of occurrences of calls to a method with a given name for the main version of the examined software systems*

Project	Name of the called method	Nr. of calls
Eclipse2.0	getString(1)	3079
Eclipse2.1	getString(1)	4218
Eclipse3.0	getString(1)	5503
Eclipse3.1	getName(0)	4140
Eclipse3.2	getName(0)	5026
Netbeans3.1	getDefault(0)	356
Netbeans3.2	getName(0)	604
Netbeans3.3.1	getName(0)	710
Netbeans3.5	getName(0)	1286
Netbeans5.5	getDefault(0)	2812
Netbeans6.0	getName(0)	5064
Ant1.5.2	log(2)	355
Ant1.5.3-1	log(2)	355
Ant1.6.0	getProject(0)	619
Ant1.6.1	getProject(0)	625
Ant1.6.5	getProject(0)	705
Ant1.7.0	executeTarget(1)	866
JDK1.2.2	size(0)	723
JDK1.3.1.02	size(0)	796
JDK1.4.0	equals(1)	1464
JDK1.5.0	equals(1)	2386
JDK1.6.0	equals(1)	2477

Table 5.20: *Classes with the maximum number of immediate subclasses for the main version of the examined software systems*

Project	Class	Nr. of subclasses
Eclipse2.0	org.eclipse.jface.action.Action	161
Eclipse2.1	org.eclipse.jface.action.Action	219
Eclipse3.0	org.eclipse.jface.action.Action	355
Eclipse3.1	org.eclipse.jface.action.Action	406
Eclipse3.2	org.eclipse.jface.action.Action	483
Netbeans3.1	org.netbeans.editor.BaseAction	85
Netbeans3.2	org.netbeans.editor.BaseAction	89
Netbeans3.3.1	org.netbeans.editor.BaseAction	93
Netbeans3.5	org.netbeans.editor.BaseAction	97
Netbeans5.5	org.netbeans.modules.j2ee.verification.AbstractRule	89
Netbeans6.0	org.netbeans.modules.uml.core.AbstractUMLTestCase	227
Ant1.5.2	org.apache.tools.ant.Task	104
Ant1.5.3-1	org.apache.tools.ant.Task	104
Ant1.6.0	org.apache.tools.ant.Task	126
Ant1.6.1	org.apache.tools.ant.Task	126
Ant1.6.5	org.apache.tools.ant.Task	128
Ant1.7.0	org.apache.tools.ant.BuildFileTest	167
JDK1.2.2	java.lang.Object	847
JDK1.3.1.02	java.lang.Object	981
JDK1.4.0	java.lang.Object	1870
JDK1.5.0	java.lang.Object	2934
JDK1.6.0	java.lang.Object	3448

Chapter 6

Bug distribution in OO systems.

The distribution of bugs in software systems has been shown to satisfy the Pareto principle, and typically shows a power-law tail when analyzed as a rank-frequency plot. Zhang showed that the Weibull cumulative distribution is a very good fit for the Alberg diagram of bugs built with experimental data [85]. We further discuss the subject from a statistical perspective, using as case studies five versions of Eclipse, to show how log-normal, Double Pareto and Yule-Simon distributions may fit the bug distribution at least as well as the Weibull distribution. In particular, we show how some of these alternative distributions provide both a superior fit to empirical data and a theoretical motivation to be used for modeling the bug generation process. While our results have been obtained on Eclipse, we believe that these models, in particular the Yule-Simon one, can generalize to other software systems.

6.1 Related Works.

The distribution of defects, or bugs, in software systems is a very important issue both from a theoretical and a practical software engineering perspective. Limiting our attention to object-oriented systems, recent papers highlighted the presence of a so called "Pareto principle" in the bug distribution, that is 20% of source files tend to include about 80% of bugs[1] [8], [33], [45], [85]. This is a sub-problem of the more general issue of a statistical description of different properties of large software systems.

Modern software systems, in fact, may be so huge that a statistical analysis of their components may be appropriate. This is witnessed by some recent works [23], [48] devoted to the investigation of the distribution of general prop-

[1] In fact, the actual percentage of bugs hitting 20% of modules with most bugs is typically 60-70%, but the substance of the principle remains.

erties in large software systems, like the number of out-links of a class, the number of lines of code, the distribution of variables names across classes, modules, packages etc., from the perspective of complex system theory [57].

Researchers agree in finding power-laws in the tails of the distribution for various software properties, computed at different levels of granularity of software components (classes, packages, files, and so on) [13], [22], [81]. These findings follow a more general Pareto 80-20 principle [57]. Note that, when needed, we use the term "modules" in a general sense, as a synonym of any of the above described basic software components. Fenton and Ohlsson [33] verified the Pareto principle for the defect distribution in packages, and this suggests that the defect distribution may also satisfy a power-law, although there is not evidence for this in literature. Their finding was recently confirmed by Andersson and Runeson [8]. Les Hatton used concepts from statistical mechanics to model how the software component sizes obey a power-law Pareto distribution when the system is subject to external constraints [44]. Zhang [85] pointed-out that fitting the Alberg diagram of bugs over modules by using a pure power-law may result in a poor approximation. He used data at the package level, which is similar to the level of "modules" used in [8]. He showed that an outstanding fit is obtained using the Weibull cumulative distribution. In order to obtain this result Zhang used the Weibull distribution to perform a best-fit of the Alberg diagram built with experimental data.

In the next we want to clarify some issues related to the distribution of bugs in the Eclipse system, which is a software development platform for Java. First, we discuss the relative significance of using statistical distributions to model how bugs are distributed among modules, and of using rank/frequency and cumulative diagrams such as Alberg's. Then, we present some key distributions used in literature to model software properties, introduce the generative models able to produce such data, and apply these models to exploit the mechanism of bug introduction in software modules. Finally, we perform an empirical computation on five versions of Eclipse, a large, publicly available Java system, which has already been used for this purpose by other authors, showing that the Yule-Simon distribution is the best fit for our empirical bug data, since it is also supported by a theoretical model suitable for explaining the generative mechanism of the measured empirical distributions.

6.2 Bug distribution and its quantification

In this work, we deal with large software systems whose source code is divided among modules, and also with bugs affecting these modules. Often, during software development, when a bug is found it is traced in a bug-tracking system

– it is assigned a unique code, a description and a status. Developers in charge of fixing the bug figure out where to change the source code to fix it, and when they make these changes they record its unique code in the comment associated to the change. In this way, with proper analysis of the logs of bug tracking and configuration management systems, it is possible to associate source files – and even classes – with bug fixing activities related to a specific bug.

Clearly, not all source modules changed due to a bug are to be considered "faulty". Some changes can happen to realign a correct piece of code with another piece of code that was modified to fix the bug. So, what we measure is to what extent a bug hits one, some or many source modules, and not whether they are really faulty.

Note that the number of bug fixings made to each module is not necessarily an indicator of how much the module is "buggy" at a given time, but of how many bugs have been introduced in the module during its development since its creation, revealed by mean of some testing procedure, and fixed. Thus, a module with more bugs discovered and fixed has been corrected more, and may even be more bug-free than other modules with less bug fixes, at a given time.

Studying large software systems, we have a high number of source modules, of the order of several thousands, or even tens of thousands, each of which can be hit by zero, one or several bugs, in the sense explained before. So, it is sensible to study the bug distribution among source modules using a statistical approach.

Traditionally, such studies are performed by recording the histogram of the number of bugs hitting the modules, and then trying to match these empirical data with some known probability distribution function (PDF), so that one can synthesize mathematically their statistical behavior. In practice, the key diagram is the complementary cumulative distribution function (CCDF) $P(x) = P[X \geq x]$, that is the probability that the number of bugs, X, is greater than or equal to a given value x.

Dealing with bug data, however, often software engineers prefer to use an Alberg diagram [33]. The Alberg diagram is obtained first by drawing a rank/frequency plot of the data – obtained in our case by ranking the modules according to their number of bugs, and then plotting this number versus the rank, r. Then, the rank-frequency plot is cumulated, and both axes are rescaled from 0 to 100 to reflect percentage values.

It is well known that the CCDF $P(x)$ and the rank/frequency plot $x(r)$ bear in fact the same information, and can be obtained from inverting one another and properly rescaling the axes. Consequently, the Alberg diagram conveys the same information of CCDF. In fact, saying that "the module with the rth largest number of bugs has x bugs" is equivalent to saying that "r modules have x or

more bugs" [7]. So, the probability $P[X \geq x] \simeq \frac{r}{N}$, where r is the rank of the sample immediately greater or equal to x, and N is the number of samples.

If we know, or estimate, the CCDF $P(x)$, its inverse $x(P) \simeq x(\frac{r}{N})$ can be used to compute the rank/frequency plot. The Alberg diagram can thus be computed using the formula:

$$A(r) = \frac{\sum_{i=1}^{r} x(\frac{i}{N})}{\sum_{i=1}^{N} x(\frac{i}{N})} \tag{6.1}$$

with A and r properly rescaled to $0-100$ intervals, to yield percentages.

We used Weibull, Double Pareto, log-normal and Yule-Simon functions to fit the empirical bug distribution. We analyzed the Eclipse software system using the same releases as in [85], plus other releases.

Double Pareto, log-normal and Yule-Simon distributions have an advantage with respect to the Weibull distribution, because they have an associated generative model that may be suitable for modeling the process of software production and the addition of bugs into software modules. On the other hand, among these only the Yule-Simon distribution can model null values. Thus, when fitting empirical data with the other distribution functions we must discard modules with zero bugs, which represent a substantial fraction of the entire system.

6.3 Bug distribution

We analyzed five main releases of Eclipse, a large Java system, from 2.1 to 3.3. For each release, we computed the number of bugs for each module. In this specific case the software modules are files of the Eclipse packages, containing one or more classes.

Table 6.1 shows the basic statistics about bug incidence, including the percentage of bugs pertaining to 20% of modules most hit by bugs.

Table 6.1: *Basic statistics of studied Eclipse releases.*

Project	nr. of modules	total bugs	modules w. bugs	% bugs in 20% most hit modules
Eclipse 2.1	7885	6416	2545	85.0
Eclipse 3.0	10584	15839	4721	79.9
Eclipse 3.1	12174	12848	4819	81.4
Eclipse 3.2	13347	9986	4071	86.0
Eclipse 3.3	14564	9427	4142	87.0

For each release, we had to discard modules with no bugs for those distribution functions not compatible with null values, retaining all data for the

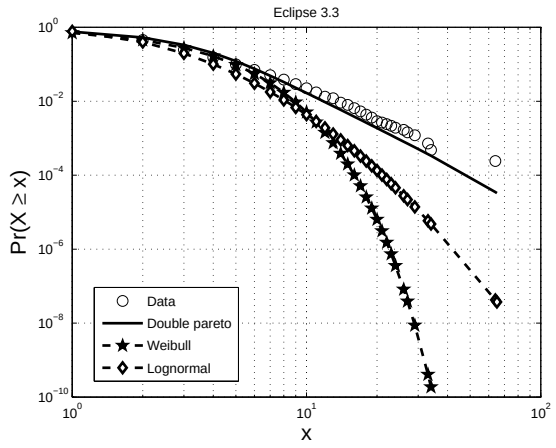


Figure 6.1: The CCDF of bugs in Eclipse 3.3. Modules with no bugs are discarded in these data, since the Double Pareto and log-normal models cannot fit zero values.

others, and performed a best-fit to these distributions using standard Matlab functions. For this reason, we reported the Yule-Simon fit on a separate plot .

Fig. 6.1 shows in a log-log plot the best-fit CCDF for empirical bug data referring to Eclipse 3.3. This plot clearly supports the Double Pareto distribution as best fit of the data, throughout its range. The log normal and Weibull distributions are a good fit for low values of bugs, but both fail in the tail. Similar results are obtained for all examined Eclipse releases.

Table 6.2: R^2 *coefficient of determination. Modules with no bugs are included only in the Yule-Simon model.*

Project	R^2 Weibull	R^2 LogNorm.	R^2 DoublePar.	R^2 Yule-Simon
Eclipse 2.1	0.929	0.949	0.961	0.99995
Eclipse 3.0	0.947	0.967	0.975	0.99994
Eclipse 3.1	0.930	0.952	0.962	0.99994
Eclipse 3.2	0.929	0.947	0.959	0.99992
Eclipse 3.3	0.921	0.944	0.957	0.99998

The goodness of Double Pareto fit (Table 6.2) highlights that the Recursive Forest model [54] can be suitable as a dynamic model for bug generation in this object-oriented software, when discarding modules without bugs.

Once we computed the best-fit distribution parameters, we plotted an Alberg diagram from the studied distributions, in order to compare our results with Zhang's ones, generating the corresponding ideal rank-frequency diagram, and then applying eq. (6.1), as described in Section 6.2. Fig. 6.2 reports the Alberg diagram referring to empirical data, for Eclipse 3.3 (discarding zero values)

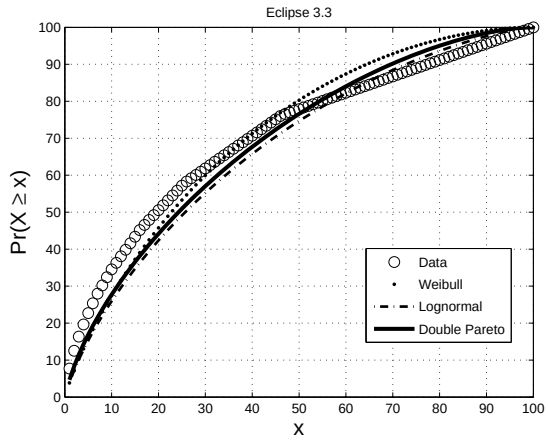


Figure 6.2: The Alberg diagram obtained from converting the fitting of the CCDF representation, for the same data as in Fig. 1. Modules containing no bugs are excluded.

and the corresponding Alberg diagrams generated from the three best-fit distributions.

In this case, at a visual inspection all three distributions look roughly equivalent for generating an Alberg diagram which may fit the Alberg diagram calculated with empirical data. Computing the coefficient of determination (R^2) statistics, as in [85] to compare the fits, we found very similar results for the three distributions, with values of the order of 0.95. The R^2 of the Double Pareto distribution is slightly better than the Weibull and the log-normal R^2's, because of the larger number of parameters available for optimizing the best fitting curves.

Note that, comparing Fig. 6.2 with a similar plot reported by Zhang [85], in our case we first fit a statistical distribution using bug data, and then generate an Alberg diagram using the theoretical distribution. In [85] files with zero bugs are not discarded and the Alberg plot is directly fitted using the functional form of the distributions. This obviously yields a better fit, with R^2 values close to 0.998.

In Fig. 6.3 we show the CCDF plot in log-log scale separately for the Yule-Simon distribution. In this case we used all the system modules, including those with no bugs. In all log-log scale plots, the points corresponding to zero bugs are automatically deleted. However, they are included in the fitting procedure. The parameter p_0 is computed as the ratio among the number of modules with zero bugs and the total number of modules, while h_0 is null since the minimum amount of bugs in a module is zero. The fitting procedure optimizes with

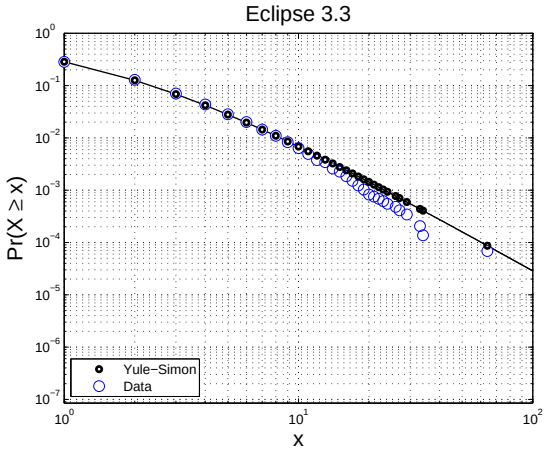


Figure 6.3: The CCDF of bugs and its best fitting Yule-Simon CCDF in Eclipse 3.3. This data set includes also modules with zero bugs.

respect to the parameters α and c of Eq. 4.8.

The coefficient of determination for the Yule-Simon distribution is quite high for all the Eclipse releases analyzed (Table 6.2), and clearly outperforms the other distributions.

6.4 Discussion

The number of bugs of our empirical data corresponds to the number of corrections applied to a module when developing a given release. In some sense, the more bugs are found and fixed in a module, the more they are reduced. Thus, in practice, our empirical distributions do not describe how many bugs are present in a module at a given time, but are related to how many bugs have been introduced into modules during software development.

Log-normal model.

Qualitatively, a module containing worse quality code should present more bugs (discovered at 'pre' or 'post' release, after testing, and so on). The other way round, a module with more bugs should contain worse quality code (bugs are the main code quality indicator). Thus, our assumption that new bugs are introduced into modules in amounts proportional to the bugs already introduced is a reasonable first approximation.

On the downside, the log-normal model presents two significant issues. First, it does not explicitly consider the system growth. New software modules are introduced during the development, while the log-normal model only allows the increase in the number of bugs – in an amount proportional to the

number that has been already discovered. The system growth is not explicitly included.

Second, and more importantly, the log-normal PDF does not include zeros. This implies that most of the software system is automatically not considered, and the model may be consistently used only for a subset of it.

Double-Pareto model.

The Double-Pareto distribution is obtained, according to Mitzenmacher, by a geometric mixture of log-normal distributions. This model explicitly includes system growth. In fact, the original model considers the distribution of files size in file systems. First, a log-normal distribution is generated starting from a single file, assuming that it is copied and manipulated to generate another file with size changed by a multiplicative factor. The files are hierarchically related, the first being the root. Then, the same procedure is applied repeatedly to one of the existing files selected at random. Asymptotically the distribution of files' size will be log-normal for files at the same hierarchical level [55]. This model is then generalized to allow for the introduction of completely new files – not created by replicating and manipulating an already existing file – as well as for file deletion. With a certain probability per step a new file is introduced, while with complementary probability an existing file is selected at random to be copied and modified into another file, according to the process described above. The resulting file size distribution is a geometric mixture of log-normal file size distributions, yielding a global Double-Pareto distribution [55].

It is possible to adapt Mitzenmacher's model to the software systems studied, assuming that the files correspond to the software modules, and the bugs to the file size. In this model, new bugs are introduced into existing modules, selected at random, in amounts proportional to the bugs already inserted (multiplicative factor). Such a model explicitly includes also the system growth, because modules may be newly created as well as discarded. Our best fitting results suggest that the Double-Pareto distribution closely represents the empirical data.

The main drawback of this distribution is that it cannot include zeros. The Double-Pareto distribution is well defined only for positive values. Thus, like the log-normal, it can model only a subset of the modules of the entire software system.

Another minor drawback is the number of parameters, four, larger than in the other distributions, while models with fewer parameters are generally preferable.

Weibull model.

The Weibull distribution models a technical system in which components present failures during time. The number of components is fixed at the beginning, and growth only occurs in the fraction of failed components. Eventually,

this fraction saturates to one. Thus, even if we can equate software modules to components, and failures to bugs, the introduction of new components and of new bugs, as well as the bug fixing activities that actually reduce the number of bugs, are hardly accounted for in this model. On the other hand, its CCDF is very flexible and presents only two parameters to be tuned for best fitting purposes.

Yule-Simon model.

This model may be easily applied to the software development process and to the introduction of bugs into modules, according to the associated statistical model described in section 3. In fact, it accounts for the addition of new entities, and for the discrete increase with time of their properties – the entities are the software modules, and their properties are the number of bugs fixed in each module. The critical point in order to obtain a system with power-law distribution in the tail is the applicability of the preferential attachment mechanism. As already mentioned, this states that the existing entities whose property is incremented are chosen with a probability proportional to the current value of this property. This condition suffices for the generation of the Yule-Simon distribution, which presents a power-law in its tail.

In our case, the entities are software modules and the properties are the bugs introduced. As we already pointed-out, what we count in our empirical distribution is the number of bugs already added, as revealed by testing procedures, by end-users, or by other means. The preferential attachment means that new bugs are more likely to be introduced into software modules which already have more bugs, since they are preferentially selected for bug introduction.

In order to verify this assumption we performed the following statistical check. We grouped together, for each single release, modules with the same number of bugs, and calculated the average number of bugs which affected these modules in the next release. If the preferential attachment mechanism can be applied to our software systems, modules are preferentially selected according to the bugs they host. Thus, on average, in the next release we should find more bugs introduced in the groups which already have more bugs, since they have a larger chance of being selected. We can also expect a linear relationship, meaning that a group having a double number of bugs with respect to another group will be, on average, preferentially selected twice, having in the next release, on average, approximately twice the number of bugs.

In Table 6.3 we report the measured data for releases 3.0 and 3.1. We selected in release 3.0 the modules with $0, 1, 2, ...14$ bugs. We discarded the remaining 80 modules with more than 14 bugs because, due to the power-law distribution in the tail, there are many missing points – no modules with 24 and 27 bugs for example – as well as large fluctuations in the existing values.

We then computed the average number of bugs fixed for the same modules in the next release, 3.1. Table 6.3 shows how modules with more bugs in a release, on average, exhibit more bugs in the next release.

Table 6.3: Average number of bugs hitting modules in release 3.1, for modules with a number of bugs between 0 and 14 in release 3.0.

Release 3.0 #Bug	Release 3.0 #Modules	Release 3.1 Average Bug Number
14	20	7.6
13	24	5.9
12	33	4.9
11	32	5.0
10	42	4.1
9	61	5.2
8	90	4.2
7	109	3.5
6	111	3.1
5	221	2.3
4	332	2.2
3	537	1.5
2	952	1.2
1	2016	0.6
0	5863	0.3

Modules with zero bugs have a non zero chance of receiving bugs, even if such event is more unlikely than for other modules. This is again in agreement with the Yule-Simon model, where the constant c take this possibility into account.

The rule that can be inferred by data in Table 6.3, is that the bugs inserted in the next release are preferentially introduced into modules with more bugs in the current release. For instance, modules with 3 bugs in Eclipse 3.0 receive on average 1.5 bugs in Eclipse 3.1, while modules with 6 bugs receive on average 3.1 bugs. Considering modules with 7 and 14 bugs in release 3.0, release 3.1 provides respectively 3.5 and 7.6 new bugs, on average, in the corresponding modules. These proportions are in agreement with the preferential attachment mechanism. All these considerations hold within the limit of statistical fluctuations, which are higher when we consider modules with more bugs.

Fig. 6.4 shows data for modules with up to 14 bugs. For each release, the proportionality among bugs in modules of the current release and bugs intro-

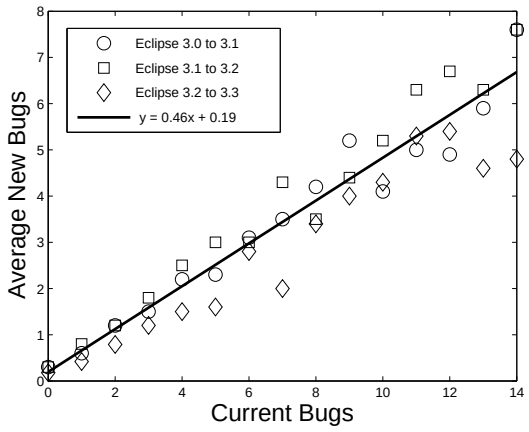


Figure 6.4: The average number of bugs introduced in the next release in the modules having bugs from zero to 14 in the current release, for Eclipse 3.0, 3.1 and 3.2. The line refers to linear interpolation of the data set "Eclipse 3.0 to 3.1".

duced in the same modules in the next release is patent, within statistical fluctuations, in agreement with the hypothesis that modules with more bugs are preferentially selected for introducing new bugs. The Figure shows, as a reference, the interpolating straight line for the data "Eclipse 3.0 to 3.1". The linear correlation coefficients for the three groups of data are all above 0.97, confirming the goodness of our model. All data show almost the same non null probability for introducing bugs also in bug-free modules. This cannot hold with the log-normal and Double-Pareto models.

Note that this finding can be exploited to forecast the average number of bugs reported in the next release, for the given classes of modules. This might be used for software quality control.

The presented data, if we exclude modules with zero bugs, may also support log-normal or Double-Pareto models, whose mechanisms imply proportionality between the number of bugs introduced and the already existing bugs. Both Yule-Simon and Double Pareto models produce fat-tails. Their differences regard only the head of the distribution, namely the part of modules with few bugs.

In conclusion, given that the Yule-Simon model is also able to include modules with no bugs, and fits the head of the data very well, we believe that it is the optimal candidate for statistically modeling the introduction and the appearance of bugs in the modules of a software system in continuous growth and evolution.

6.5 Conclusions

We have discussed and extended the use of fat-tail statistical distributions to bugs affecting large software systems, up to now roughly described as the 80-20 Pareto principle. We analyzed different releases of the Eclipse software system, adopting statistical distributions already employed in the literature to describe general properties of large software systems.

The motivations for this study came from a previous study of Zhang, which used best fitting procedures on the Alberg diagram to support the Weibull distribution as optimal candidate for explaining the bug distribution in modules. We showed how all the information contained in the Alberg diagram is retained if we use the CCDF. In fact, our approach of using a CCDF distribution function can easily take advantage of generative models available in the literature, which can explain the genesis of such distributions. We believe that an approach supported by a generative model is to be preferred with respect to one only based on best fitting arguments.

We performed a study of the bug distribution in modules of various versions of the Java Eclipse system, comparing the goodness of fit of log-normal, Double Pareto, Yule-Simon and Weibull distributions.

Using an empirical analysis, we showed that new bugs are introduced, on average, in larger amounts in modules which where more bug affected in previous releases. This explicitly supports the "preferential attachment mechanism" at the basis of the Yule-Simon model, not excluding mechanisms underlying the log-normal and the Double Pareto models. The last two models are, however, less useful because they cannot account for zero values, namely modules with no bugs, and may be only useful to model a reduced part of the entire software system.

The Weibull distribution is instead related to the modeling of a system made of components which fail at a given rate in time, and it is successfully used in reliability engineering and in survival analysis. Thus, while it is flexible enough to provide very good fits for fat-tail distributions by adjusting only two parameters, it cannot easily be accommodated for modeling systems where the number of components, as well as the components' properties, grow in time, as in the case of modules and bugs in software systems.

This works contributes to the body of empirical and theoretical research on software fault behavior in large complex systems, studying different versions of the Eclipse software system. This work takes into account the conventional wisdom about bug distribution across software modules, showing the strengths and weaknesses of the proposed models.

From the perspective of software practitioners, our results might be useful to keep software quality under control. In fact, a statistical model able to esti-

mate the future rate of defects in classes of modules might help management to organize pre- and post-release testing.

Chapter 7

Social Networks Metrics and Object Oriented Software.

We already introduced the SNA metrics derived from Social Networks, and illustrated their meaning in the context of software networks. Now we show the application to object-oriented software of such metrics, and examine their relationship with software quality, as expressed in terms of number of bugs affecting the system. Social Networks metrics, as for instance the EGO metrics, allow to identify the role of each single node in the information flow through the network, being related to software modules and their dependencies. These metrics are compared with other traditional software metrics, like the Chidamber-Kemerer suite, and software graph metrics.

We examine the empirical distributions of all the metrics, bugs included, across the software modules of several releases of two large Java systems, Eclipse and Netbeans. We provide analytical distribution functions suitable for describing and studying the observed distributions. We study also correlations among metrics and bugs.

We found that the empirical distributions systematically show fat-tails for all the metrics. Moreover, the various metric distributions look very similar and consistent across all system releases, and are also very similar in both the studied systems. These features appear to be typical properties of these software metrics.

We present a study of a set of releases of two large Open Source OO systems, Eclipse [3] and Netbeans [4] from the software network perspective, and compute the observed complementary cumulative distribution functions (CCDF) [57] of SNA metrics applied to software networks and of several others. We study such systems because both the source code of several versions, and complete data about Bugs and Issues of their software modules are available.

We study the relationships between these metrics and software fault-proneness

– measured as the number of Bugs affecting software modules – and between them and more traditional software metrics. We also study the possibility of estimating the metric features for the future releases. For all the observed distributions we performed best fits, finding analytical distributions able to model the system.

The systems analyzed are written in Java. All their classes are contained in Java source files, called Compilation Units (CU). A CU generally contains just one class, but less frequently it may contain two or more classes. We extracted the Bugs affecting files merging information found in bug-tracking repositories, specifically Bugzilla [15] and Issuezilla [43], with information taken from source code repositories, namely Concurrent Versioning System (CVS) [2]. The information about Bugs and software changes (commit logs) is reported at CU level, and not at class level. Therefore, we extended the concept of software graph to CU level, building a graph in which nodes are Compilation Units and edges are the relationships between these CU's, extracted from the classes belonging to each CU. We used this graph for computing all the metrics analyzed as well as for computing the Bug distributions.

We found that most of the studied metrics are distributed according to the Yule-Simon distribution [67] [22], to a high degree of accuracy, and show a persistent or universal character across all different releases, for both systems analyzed. The high degree of accuracy of the analytical fitting distributions and their persistent character allowed us to estimate metrics values for the subsequent releases.

7.1 Research Questions.

The Pareto principle (80-20 rule), and the presence of power-laws in the tail of the distributions of many properties of software systems, including Bugs, have already been observed [23], [8] [85]. In [79], a high order statistic coefficient was proposed to analyze software metrics exhibiting highly skewed statistical distributions, that was efficient in observing changes in software systems and in monitoring the development process.

We investigate if the new proposed SNA metrics possess the same properties and have similar empirical distributions. Moreover, the new metrics might possibly show correlations with Bugs and/or with other metrics and properties. Thus, it is desirable to study these correlations.

We also investigate if there are analytical distribution functions which may be used to describe such empirical distributions and possibly to forecast future properties of the software systems.

Consequently, our research questions are the following:

- **RQ1**- Are there analytical distribution functions describing the empirical data? Have these functions power-law behavior in their tails? What is the significance level of fitting empirical data with these distributions?
- **RQ2**- Are these distributions similar in all the releases and in different systems, or tend to vary significantly?
- **RQ3**- Is it possible to use these distributions to estimate the metrics values in subsequent releases?
- **RQ4**- Are there SNA metrics significantly correlated with software Bugs, and to which extent?
- **RQ5**- Are there SNA metrics significantly correlated to traditional CK metrics, and to which extent?

7.2 CU Software Networks and CU-CK Metrics.

In this study we do not distinguish among the various possibilities of software relationships, and with regard to SNA metrics, for simplicity we do not even consider edges orientation, which would imply the construction of different EGO networks for the different kinds of links. Ours is a static analysis. Furthermore, since our software nodes are CUs, as explained later, many relationships among Java classes lose their original meaning at this granularity level. Our purpose is to focus on the role of the interactions among the software elements. The number and orientation of edges allow to study the coupling between nodes, that is between classes. In this graph, the in-degree of a class, or Fan-in, is the number of edges directed toward the class. The out-degree of a class, or Fan-out, is the number of edges leaving the class. Besides Fan-in and Fan-out metrics, we computed also, for each class, four CK metrics which were observed to be significantly correlated with the number of Bugs. They are:

- Weighted Methods per Class (WMC). A weighted sum of all the methods defined in a class. We set the weighting factor to one, to simplify our analysis.
- Coupling Between Objects (CBO). The counting of the number of classes which a given class is coupled to.
- Response For a Class (RFC). The sum of the number of methods defined in the class, and the cardinality of the set of methods called by them and belonging to external classes.

- Lack of Cohesion of Methods (LCOM). The difference between the number of non cohesive method pairs and the number of cohesive pairs.

We also computed the lines of code of the class (LOC), excluding blanks and comment lines. This is useful to keep track of the class size, because it is known that a "big" class is more difficult to maintain than a smaller class.

Every class is contained in a Java file, called CU. While most files include just one class, there are files including two or more classes. In Eclipse, about 10% of CUs host more than one class, whereas in Netbeans this percentage is about 30%.

While OO metrics and class graphs are usually referred to classes, Bugs and Issues are typically associated to CUs, because the logs of coding efforts aimed to fix Bugs are associated to changes to the source code, which are made to files (the CUs). Since the number of Bugs is of paramount importance to define software quality, to make Issue tracking consistent with source code we decided to base our analysis on CUs. Consequently, we extended CK metrics from classes to CUs. CUs represent therefore the main element of our study.

We defined a CU graph, whose nodes are the CUs of the system. Two nodes, A and B, are connected with an edge directed from A to B if at least one class inside the CU represented by A has a dependency relationship with one class inside the CU represented by B. Referring to this graph, we can compute In-links and Out-links of a CU-node. We reinterpreted LOC and CK metrics for this CU-graph:

- CU LOC is the sum of the LOCS of the classes contained in the CU;
- CU CBO is the number of out-links of each node, excluding those representing inheritance. This definition is consistent with that of CBO metrics for classes;
- CU LCOM and CU WMC are the sum of LCOM and WMC metrics of the classes contained in the CU, respectively;
- CU RFC is the sum of weighted out-links of each node, each out-link being multiplied by the number of specific distinct relationships between classes belonging to the CUs connected to the related edge.

For each CU we have thus a set of 7 metrics: In-links (Fan-in), Out-links (Fan-out), CU-LOCS, CU-LCOM, CU-WMC, CU-RFC and CU-CBO. These metrics were computed for CUs of all versions of Eclipse and Netbeans.

We analyze the correlations among all of the SNA metrics, as well as with the other metrics and with Bugs. For some metrics we analyzed the statistical distributions and performed best fits with analytical distribution functions.

7.3 Issues Extraction.

Bug Tracking System (BTS) are commonly used to keep track of Bugs, enhancements and features – called with the common term 'Issues' – of software systems. The open source systems studied, Eclipse and Netbeans, make use of BTS Bugzilla and Issuezilla, respectively.

Each Issue inside a BTS is univocally identified by a positive integer number, the Issue-ID. BTS store, for each tracked Issue, its characteristics, life-cycle, software releases where it appears, and other data. In Bugzilla, a valid Bug is an Issue with a resolution of 'fixed', a status of 'closed', 'resolved' or 'verified', and a severity that is not 'enhancement', as pointed out in Eaddy et al. [28]. Thus, Bugs are a subset of Issues. For Issuezilla, it is possible to adopt an equivalent definition: a Bug is an Issue with a resolution and status as above, and with type 'defect'.

Software configuration management systems like CVS (Concurrent Version System) keep track of all maintenance operations on software systems. These operations are recorded inside CVS in an unstructured way; it is not possible, for instance, on query CVS to know which operations were done to fix Bugs, or to introduce a new feature or enhancement. In order to identify Issues (Bugs) affecting systems CUs, we had to match data stored in BTS with other data recorded in CVS of Eclipse and Netbeans.

All commit operations are committed to the CVS log messages as single entries. Each entry contains various data – among which the date, the developer who made the changes, a text message referring to the reasons of the commit, and the list of CU's interested by the commit. To obtain a correct mapping between Issue(s) and the related CU(s) the only way is to analyze the CVS log messages, to identify commits associated to maintenance operation where Issues are fixed. If a maintenance operation is done on a CU to address an Issue, we consider the CU as affected by this Issue.

In our approach, we first analyzed the text of commit messages, looking for Issue-IDs. In fact, in commit messages there may be strings such as 'Fixed 141181' or 'bug # 141181', but sometimes only the Issue-ID is reported. Unfortunately, every positive integer number is a potential Issue-ID, but sometimes numbers can refer to maintenance operations not related to Issue-ID resolution, such as branching, data, number of release, copyright updating, and so on.

To avoid wrong mappings between Issue-IDs and CUs, we applied the following strategies:

- For each release a CU can be hit only by Issues which are referred to in the BTS belonging to the same release.

- We did not consider some numeric intervals particularly prone to host false positive Issue-IDs.

The latter condition is not particularly restrictive in our study, because we did not consider the first releases of the studied projects, where Issues with 'low' ID appear. All IDs not filtered out are considered Issues and associated to the addition or modification of one ore more CUs, as reported in the commit logs. This method might not completely address the problems of the mapping between bugs and CUs [9].

In any case we checked manually:

- 10% of CU-bug(s) associations (randomly chosen) for each release

- each CU-bug association for 6 sub-projects (3 for Eclipse and 3 for Netbeans) without finding any error. A bias may still remain due to lack of information on CVS [27].

The total number of Issues affecting a CU in each release constitutes the Issue-metric we consider in this study, while the subset of Issues satisfying the conditions as in Eaddy et al. is the Bug-metric [28]. Clearly, not all source modules changed due to a Bug are to be considered "faulty". Some changes can happen to realign a correct piece of code with another piece of code that was modified to fix the Bug. So, what we measure is to what extent a Bug hits one, some or many CUs, and not whether they were really faulty.

7.4 Empirical results regarding metric distributions

We systematically analyzed several main releases of Eclipse and Netbeans projects, namely releases from 2.0 to 3.4 of Eclipse, and releases from 3.2 to 6.1 of Netbeans. For each release, we computed the class graph and the consequent CU graph, and computed all the above quoted metrics at CU level. We analyzed the statistical distributions of the metrics among the systems CU's, which are our graph nodes, as well as the Bugs and Issues distributions. Note that we used CU metrics to be able to study more easily their relationships with Bugs and Issues. However, we verified that the behavior of CU metrics is absolutely similar to the behavior of the corresponding class metrics, for all considered metrics.

Tables 7.1 and 7.2 show the number of CUs in the various releases considered of Eclipse and Netbeans, respectively, together with their release date. Both the size and the release date of the considered systems vary considerably. The sizes – in number of CUs – vary of one order of magnitude in Netbeans, and about three times in Eclipse.

We started the analysis by computing the empirical CCDF's of the software network metrics for the various system studied. The empirical distributions of

Table 7.1: *Number of CUs of Eclipse for each release*

Release	2.0	2.1	3.0	3.1	3.2	3.3	3.4
Number of CU	6391	7545	10288	11854	14138	15439	17387
Release date	06-2002	03-2003	06-2004	06-2005	06-2006	06-2007	05-2008

Table 7.2: *Number of CUs of Netbeans for each release*

Release	3.2	3.3	3.4	4.0	6.0	6.1
Number of CU	3346	4383	6264	9317	31425	35034
Release date	04-2001	11-2001	08-2002	12-2004	12-2007	04-2008

all considered SNA metrics show the same shape for all releases, both in Eclipse and Netbeans. Therefore, we show only the figures for some selected metrics for the last considered releases of the studied systems, namely Eclipse-3.4 and Netbeans-6.0.

Fig. 7.1 shows graph and SNA metrics for Eclipse 3.4. All CCDF are reported for convenience in log-log plots. Most CCDF show a small cut-off in the extreme tail, which is typically due to the finite size of the sample. Fig. 7.2 shows the same data for Netbeans 6.0. The behavior of Netbeans metrics is very similar to Eclipse's, with smaller cut-off in the extreme tail, perhaps owing to the higher numbers of CUs.

In order to compare the empirical distributions across the releases, we show in the same plot two SNA metrics, Effective-Size and Brokerage, for both Eclipse and Netbeans, to highlight their overlap. Fig. 7.3 shows the persistency of the distributions of these metrics across three different releases, starting from the earliest to the most recent. In Eclipse the curves slightly differ only in the tail, while in Netbeans they are almost coincident.

The empirical distributions of all considered metrics highly preserve the same shape, meaning that, for each specific metric, a single distribution function may account for the empirical data for all the system releases. Moreover, the distributions of the same metric looks also very similar in Eclipse and Netbeans releases. Thus, once this distribution is known for one metric in one release, it is possible to infer the properties of the same metric in other releases, provided that the number of CUs is known.

Regarding what specific distribution function can best fit our empirical data, we experimented with the three distributions cited above – power-law, lognormal, and Yule-Simon distributions.

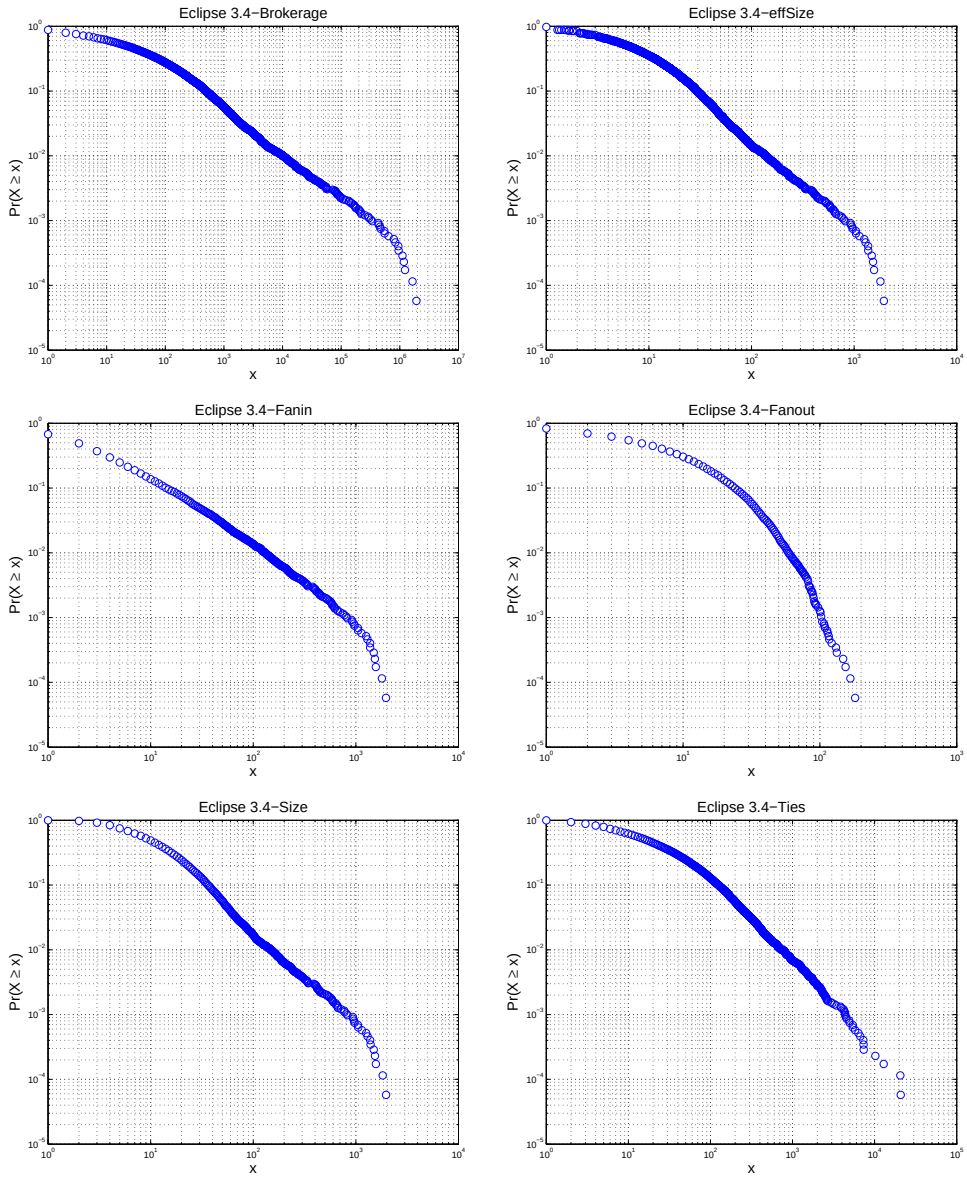
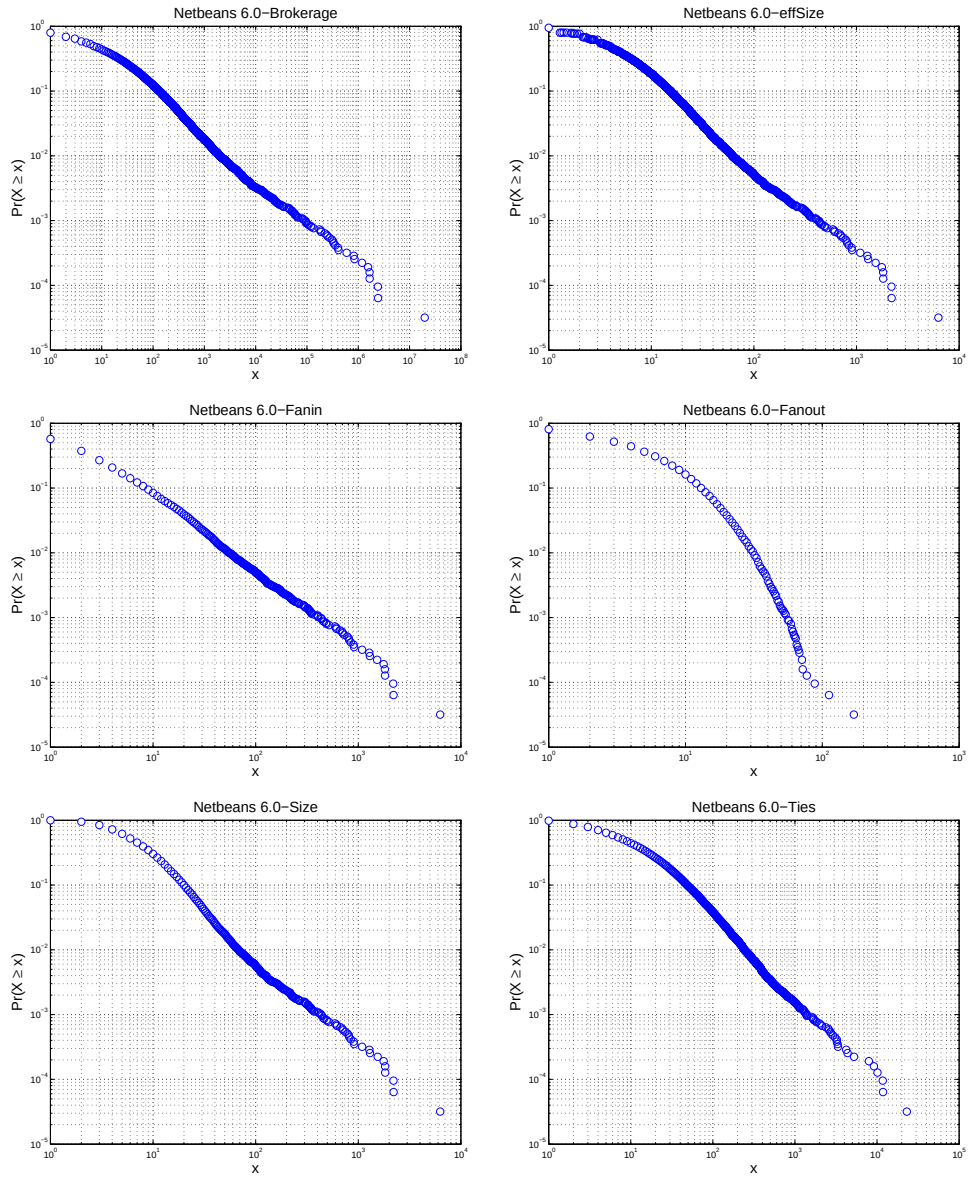


Figure 7.1: *CCDF of SNA metrics for Eclipse 3.4 release. The name of the metrics is in the top of the box. The power-law behavior in the tail is patent for all metrics.*

Fig. 7.4 shows Fan-in, Fan-out, LOC, Size and Ties, together with best-fit functions, for Eclipse-3.1. For the LOC metric, only the data with the Yule-Simon best-fit curve is shown, while for the other metrics data and best-fits with all the three distribution functions are shown in two different figures.

The fit using a truncated power-law is almost always very good. Note, however, that this fit is made starting from a minimum value x_0, denoting the value



e

Figure 7.2: *CCDF of SNA metrics for Netbeans 6.0 release. The name of the metrics is in the top of the box.*

from which the power-law tail is apparent. This makes easier to get good fits. The fit with a lognormal is usually the poorest. This distribution is able to fit very well the bulk of the samples with small values, but in general it tends to zero too quickly with respect to empirical data. The fit with Yule-Simon distribution is sometimes very good, both for small values and in the tails. Other

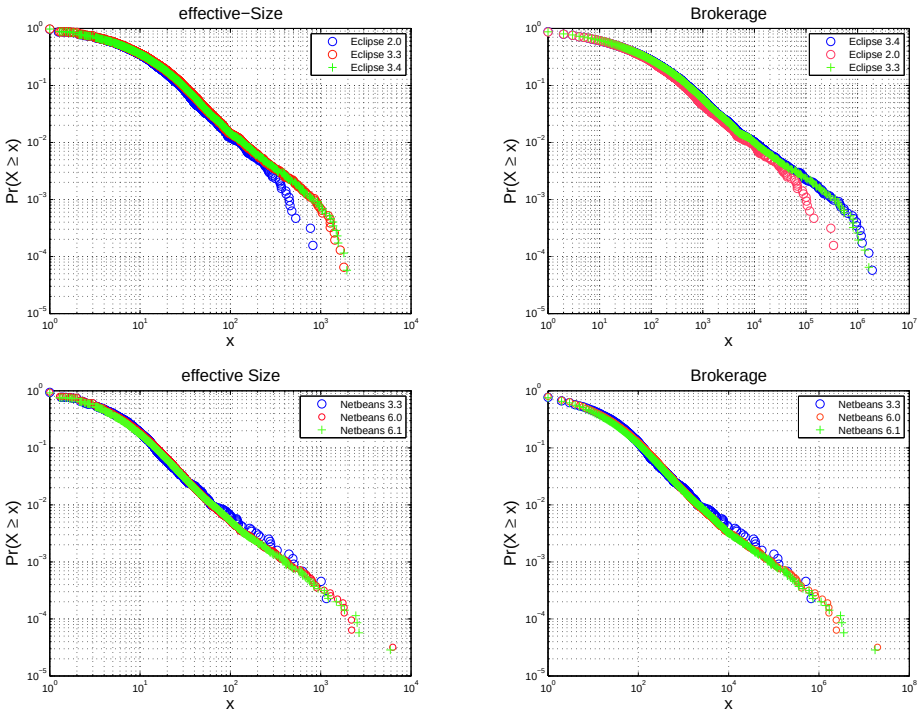


Figure 7.3: *CCDF of EffSize and Brokerage metrics for various Eclipse and Netbeans releases. A very similar behavior is patent for all metrics and across all releases of the same system.*

times, it fails to get a good fit in the tail.

In order to evaluate fit accuracy we used the determination coefficient R^2, defined by $R^2 = 1 - SE/ST$, with:

$$SE = \sum_i (f_i - y_i)^2 \qquad (7.1)$$
$$ST = \sum_i (\bar{y} - y_i)^2$$

where y_i are the empirical CCDF values and f_i the corresponding best fitting values. All the fits have very high determination coefficients , sometimes up to 0.999 (Table 7.3). This suffices to answer to our research questions. It is in fact known that when experimental data are roughly power-law distributed, it is in general extremely difficult to assess the difference among a true power-law and other fat-tail distributions, since typically any statistical test does not rule out one or the other distribution function. In fact they are often compatible with many different distribution functions [57].

Eclipse 3.1

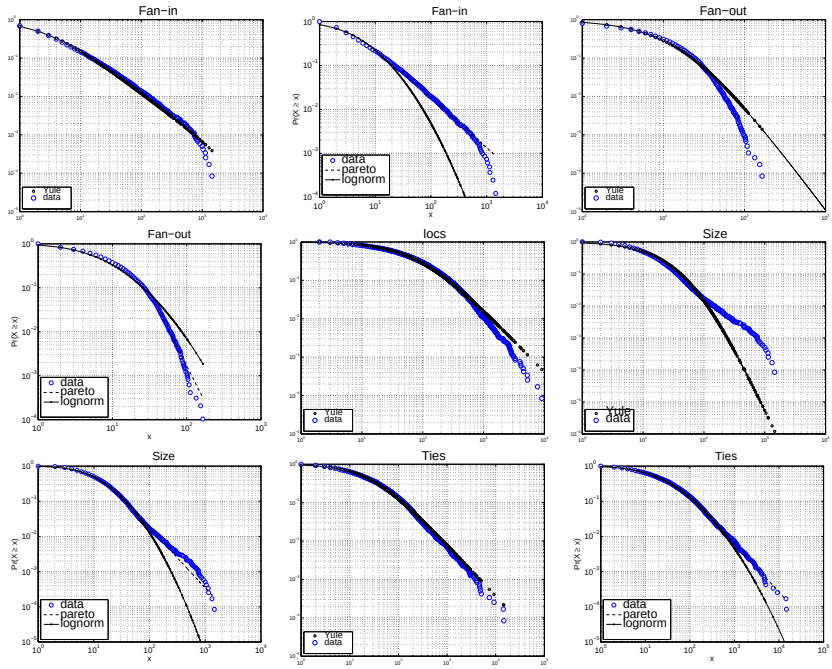


Figure 7.4: *Empirical CCDFs of various metrics in Eclipse 3.1, with their best-fit theoretical distributions. Yule-Simon fit is shown separately.*

Our purpose is, on the contrary, to provide a reasonable statistical descrip-

tion of the empirical data, and to find the analytical distribution function with the best fit. This allows us to make statistically reliable forecasts on the value assumed by some metrics in the future system releases. In our case power-law is not in principle more interesting than the log-normal or Yule-Simon distributions, as long as these provide reliable estimates and good descriptions of the empirical data. Any other statistical speculation in order to discriminate among power-law or other distributions is out of our purposes.

Note that the determination coefficients are evaluated on the linear scale, whereas all the figures are in a log-log scale. In this scale, the discrepancy between best fitting curves and empirical curves are visually enhanced, especially in the tail, whereas in the original scale the fitting curves and the empirical ones visually overlap. On the other hand our fitting procedure does not rely on any log-log representation of the data.

Fig. 7.5 shows the corresponding data and best fits for Netbeans 3.2. Also for this system the curves provide a very good fitting of empirical data, for the various releases and for the different metrics. Again the coefficient of determination is always close to one (Table 7.4). The power-law provides an excellent approximation for the data in the tail above the x_0 cut-off, whose value depends on the metrics and on the system version.

Table 7.3: Determination coefficients for the three distribution functions (Eclipse-3.1).

R^2	Yule-Simon	Lognormal	Power-law
Fan-in	0.999	0.971	0.998
Fan-out	0.995	0.989	0.997
Size	0.987	0.999	0.998
Ties	0.998	0.999	0.999

Table 7.4: Determination coefficients for the three distribution functions (Netbeans-3.2).

R^2	Yule-Simon	Lognormal	Power-law
Fan-in	0.999	0.978	0.998
Fan-out	0.998	0.982	0.996
Size	0.980	0.995	0.998
Ties	0.999	0.998	0.999

The empirical studies presented above answer our first two research questions:

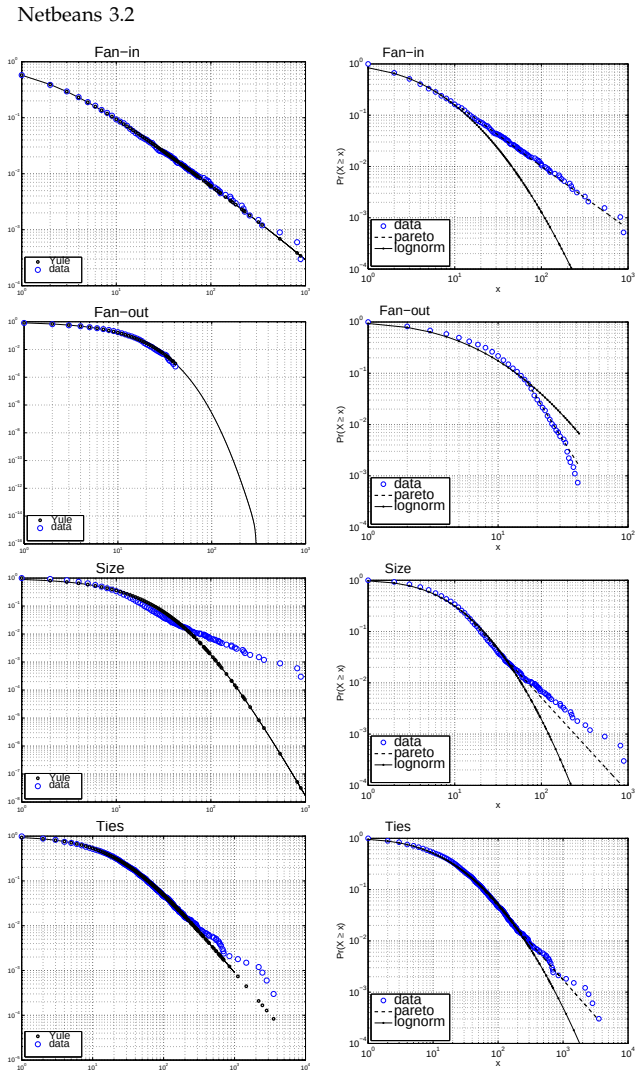


Figure 7.5: *Empirical CCDFs of various metrics in Netbeans 3.2, with their best-fit theoretical distributions. Yule-Simon fit is shown separately.*

R1: *Are there analytical distribution functions describing the empirical data?*

Have these functions power-law behavior in their tails? What is the significance level of fitting empirical data with these distributions?

We definitely found that all studied metrics, traditional OO, network-based, and derived from Social Network Analysis, tend to follow precise analytical distributions to a high degree of significance level, according to our best-fitting criteria. These distributions are power-law – from a minimum value of data, x_0 – lognormal and Yule-Simon distributions. All three distributions are compatible with a power-law behavior in their tail – regarding the lognormal distribution, this is true for datasets of finite size.

The fit using a truncated power-law are always very good. However, they depend on an *ad hoc* setting of the value x_0, and the power-law regards only the samples whose value $x \geq x_0$. Lognormal distribution shows good fits, according to the value of the determination coefficients, but not as good as power-law. Yule-Simon distribution, on the other hand, shows determination coefficients very similar to those of power-law, but the fit is over all the range of values. So, in general Yule-Simon distribution can be considered the best for most considered metrics.

R2: *Are these distributions similar in all the releases and in different systems, or tend to vary significantly?*

We found that all considered metrics have a very consistent statistical behavior across all the releases of the same system, even when these releases span over years, and have very different numbers of classes (and CUs).

For completeness, we studied also other Java systems, belonging to the Qualitas Corpus [61] and found that the considered metrics, in systems with over one thousand classes, show behaviors very similar to those reported in this paper for Eclipse and Netbeans.

Next, we analyzed also the metrics related to Issues and Bugs. We found that also the distributions of Bugs and Issues follow similar patterns, in both Eclipse and Netbeans. In Figs. 7.6 and 7.7 we show the empirical distributions of Issues and Bugs, for the releases 3.3 of Eclipse, and 6.0 of Netbeans, together with the best fitting curves of the three considered distribution functions. All Issues and Bugs distributions are very similar throughout all Eclipse and Netbeans releases, so these figures can be considered typical.

The distributions of these metrics are well fitted by the simple power-law, according to the determination coefficient, above a threshold x_0, which depends on the particular data, and very well fitted by the Yule-Simon distribution since the beginning of the data. The log-normal distribution provides a worse fit, even if the determination coefficients R^2 are always above 0.94. Note again that the log-log scale enhances visually the distances in the tail, but the absolute values of the difference among fitting curves and empirical distributions are very small.

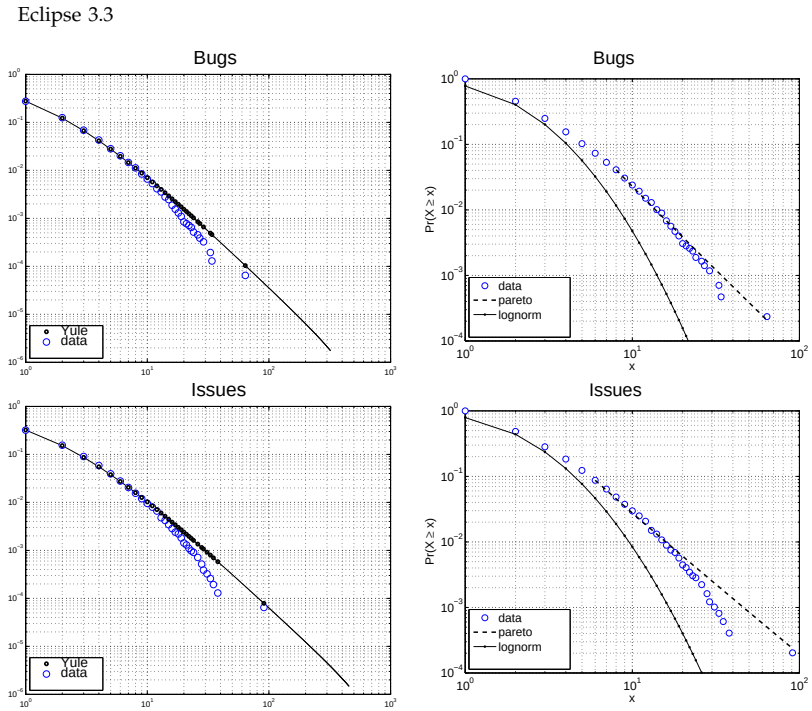


Figure 7.6: *Empirical CCDFs of Bugs and Issues in Eclipse 3.3, with their best-fit theoretical distributions. Yule-Simon fit is shown separately.*

7.5 Correlations

In this section we report the correlations among SNA metrics, CK metrics and Bugs. Since the empirical distributions of all metrics are strongly not normal,

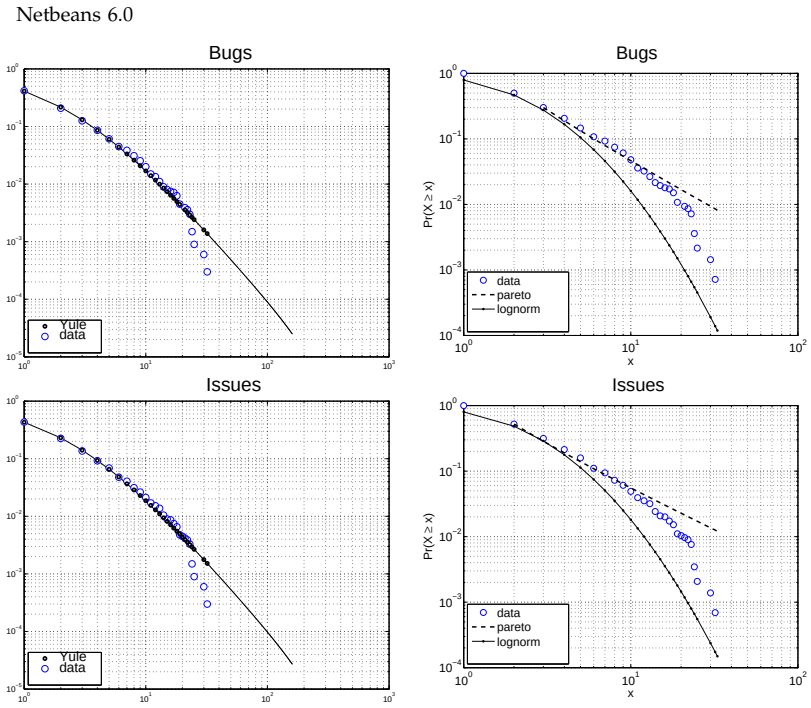


Figure 7.7: *Empirical CCDFs of Bugs and Issues in Netbeans 6.0, with their best-fit theoretical distributions. Yule-Simon fit is shown separately.*

correlations are better described using the Spearman coefficient. In our study

we computed also Pearson correlations, which are reported only in one case, for comparison. Our considerations, however, will refer only to Spearman correlation. Using the latter, data must be ranked, the correlation coefficient being given by:

$$\rho_{SP} = 1 - \frac{6\sum_i d_i^2}{n(n^2-1)} \tag{7.2}$$

where d_i are the differences among the ranks of each observation.

We report the correlations only for Eclipse-2.1 and for Netbeans-3.2, as representative of all the other releases. Tables 7.5 and 7.6 report correlation data for Eclipse-2.1, using Pearson and Spearman coefficients, respectively. Table 7.7 reports Spearman coefficients for Netbeans-3.2. The correlation coefficients in all other releases of the same system are substantially similar to those reported here, for both Eclipse and Netbeans.

The higher correlations are among Issues and Bugs, as it is natural, being one a subset of the other. This means that nodes having an high number of Issues also tend to have a high number of Bugs. In other words, the number of Bugs is always about the same fraction of Issues. Thus only one of them will be included in the subsequent analysis.

We computed the correlation matrix among Issue, Bug, CK metrics, LOC, Fan-out, Fan-in and EGO-metrics. Correlations are almost the same in each release, with fluctuations generally below 10%.

In Eclipse, CK metrics, LOCS, Fan-Out and EGO metrics generally show a moderate correlation with respect to Issues (Bug). In Netbeans, we have similar correlations, though usually slightly smaller. In both cases, the predictive power of these metrics is similar for the same software system. In both systems, LOC metric is the most correlated with Issues. This is expected, because bigger files have a larger chance to produce Issues and Bugs. However, other good predictors of Issues – comparable with LOC – are RFC, Fan-out, Size and, to a lesser extent, LCOM, Ties and Brokerage. In general, we observe that many SNA metrics are quite correlated with the number of Issues (and Bugs), showing the importance of considering these metrics.

In both Eclipse and Netbeans, Fan-in always shows a small – though significant – correlations with Issues. The different correlation between Fan-in and Fan-out with respect to Issues, indicates that to identify a fault-prone node it is important to take into account not only the number of links but also their direction. An Out-link directed from a compilation unit A to a compilation unit B may be considered like a channel easing the propagation of defects from B to A, but not vice-versa. This fact highlights the importance of an analysis of a software system as an oriented graph.

CK and LOC metrics correlations with Issues are in line with results previously showed in [12]. In Eclipse, correlations between CK metrics and Eff-Size, Closeness, Size, Ties, brokerage are quite large. Correlation with Nweak-comp, Infocentrality, Dwreach and Closeness are smaller. Only a minor correlation exists between CK metrics and Reach-Efficiency.

In Netbeans, correlations between CK metrics and Eff-Size, Size, Ties, Brokerage are also large. Smaller correlations hold between CK metrics and Closeness, Nweakcomp, Dwreach. Only minor correlations, like in Eclipse, exist between CK metrics, Reach-Efficiency and Info-Centrality.

In both Eclipse and Netbeans, the only metrics that are anti-correlated with the number of Issues are Info-Centrality and Nweak-Comp, suggesting that it is better for a CU to have a high Information Centrality and Normalized number of Weak Components, to be less prone to get Issues and Bugs.

Most Eclipse and Netbeans EGO metrics are not strongly correlated with each other. For example, Reach-Efficiency has small correlation with Eff-Size, Size and Brokerage, and no correlation with Nweakcomp. Size metric is the most correlated with the others EGO-metrics, and shows an almost perfect correlation with Eff-Size and Brokerage. Consequently, it is clearly needed to consider just one of these metrics. We suggest to use Size, which is easier to compute and, at least in the considered systems, looks slightly better correlated to Issues.

These findings related to correlations answer our last two research questions:

R4: *Are there SNA metrics significantly correlated with software Bugs, and to which extent?*

The data reported, and data very similar to them related to all other considered releases of Eclipse and Netbeans, confirm that there are significant correlations between several SNA metrics and the number of Bugs. These correlations are of the same order of magnitude of more traditional CK metrics – whose predictive power in predicting faulty classes has been studied and assessed for a long time [12] [37]. Note that all CK metrics, and most SNA metrics, are basically complexity metrics, denoting high coupling and/or low cohesion of the measured module. This is consistent with the positive correlation between these metrics and the fault-proneness of the module. However, some SNA metrics are anti-correlated to a fairly high extent with the number of Bugs, and this property might be further studied and exploited.

R5: *Are there SNA metrics significantly correlated to traditional CK metrics, and to which extent?*

The study of Tables 7.6 and 7.7 confirms that all SNA metrics are significantly correlated to all the four considered CK metrics – WMC, RFC, CBO and LCOM. Some SNA metrics – namely Eff-Size, Size, Ties and Brokerage – show

Table 7.5: Eclipse 2.1. Pearson correlation among metrics

	numissue	numbug	LOCS	WMC	RFC	LCOM	CBO	fanin	fanout	reach efficiency	effsize	closeness	dwreach	infocentrality	size	ties	nweakcomp
numbug	97% **	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
LOCS	53% **	53% **	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
WMC	49% **	48% **	57% **	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RFC	58% **	59% **	68% **	92% **	-	-	-	-	-	-	-	-	-	-	-	-	-
LCOM	32% **	30% **	19% **	79% **	62% **	-	-	-	-	-	-	-	-	-	-	-	-
CBO	54% **	55% **	65% **	41% **	70% **	11% **	-	-	-	-	-	-	-	-	-	-	-
fanin	18% **	17% **	10% **	30% **	26% **	26% **	4% **	-	-	-	-	-	-	-	-	-	-
fanout	51% **	52% **	62% **	30% **	58% **	3% **	94% **	-1%	-	-	-	-	-	-	-	-	-
reachAefficiency	0%	1%	-4% *	-4% **	-1%	-3% *	9% **	-14% **	14% **	-	-	-	-	-	-	-	-
effsize	30% **	29% **	25% **	36% **	40% **	26% **	29% **	96% **	26% **	-10% **	-	-	-	-	-	-	-
closeness	-2%	-2%	-1%	-1%	-2%	0%	-3%	-1%	-3% *	-5% **	-2% *	-	-	-	-	-	-
dwreach	27% **	27% **	23% **	17% **	29% **	4% **	46% **	19% **	50% **	44% **	32% **	-18% **	-	-	-	-	-
infocentrality	-2% *	-2% *	-2%	-2%	-3% *	0%	-4% **	-1%	-5% **	-6% **	-2% *	94% **	-25% **	-	-	-	-
size	32% **	31% **	28% **	38% **	42% **	26% **	32% **	95% **	29% **	-10% **	100% **	-2%	34% **	-3% *	-	-	-
ties	32% **	31% **	27% **	43% **	45% **	37% **	27% **	87% **	23% **	-9% **	89% **	-1%	21% **	-2%	89% **	-	-
nweakcomp	-23% **	-23% **	-27% **	-18% **	-28% **	-2% *	-39% **	-14% **	-40% **	-2% *	-21% **	4% **	-15% **	5% **	-25% **	-22% **	-
brokerage	16% **	15% **	9% **	30% **	26% **	35% **	8% **	85% **	3% *	-5% **	83% **	0%	12% **	-1%	82% **	88% **	-7% **

** Correlation is significant at the 0.01 level. * Correlation is significant at the 0.05 level.

Table 7.6: Eclipse 2.1. Spearman correlation among metrics

	numissue	numbug	LOCS	WMC	RFC	LCOM	CBO	fanin	fanout	reach efficiency	effsize	closeness	dwreach	infocentrality	size	ties	nweakcomp
numbug	95% **	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
LOCS	46% **	46% **		-	-	-	-	-	-	-	-	-	-	-	-	-	-
WMC	38% **	38% **	84% **	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RFC	46% **	46% **	90% **	89% **	-	-	-	-	-	-	-	-	-	-	-	-	-
LCOM	34% **	34% **	66% **	85% **	74% **	-	-	-	-	-	-	-	-	-	-	-	-
CBO	45% **	45% **	78% **	61% **	86% **	50% **	-	-	-	-	-	-	-	-	-	-	-
fanin	8% **	7% **	7% **	26% **	7% **	26% **	-14% **	-	-	-	-	-	-	-	-	-	-
fanout	44% **	44% **	77% **	60% **	84% **	51% **	95% **	-15% **	-	-	-	-	-	-	-	-	-
reachAefficiency	16% **	16% **	29% **	17% **	35% **	13% **	48% **	-29% **	52% **	-	-	-	-	-	-	-	-
effsize	41% **	41% **	59% **	59% **	68% **	56% **	63% **	45% **	66% **	21% **	-	-	-	-	-	-	-
closeness	38% **	39% **	51% **	45% **	59% **	42% **	62% **	14% **	66% **	63% **	72% **	-	-	-	-	-	-
dwreach	40% **	40% **	54% **	48% **	62% **	45% **	66% **	15% **	70% **	68% **	76% **	96% **	-	-	-	-	-
infocentrality	-33% **	-34% **	-45% **	-35% **	-48% **	-35% **	-53% **	-2% *	-55% **	-51% **	-59% **	-79% **	-83% **	-	-	-	-
size	43% **	42% **	63% **	62% **	71% **	58% **	66% **	46% **	69% **	22% **	98% **	72% **	76% **	-59% **	-	-	-
ties	42% **	42% **	64% **	60% **	69% **	56% **	65% **	41% **	67% **	17% **	89% **	65% **	69% **	-65% **	94% **	-	-
nweakcomp	-28% **	-28% **	-48% **	-41% **	-47% **	-37% **	-44% **	-23% **	-44% **	-3% **	-42% **	-31% **	-34% **	51% **	-52% **	-71% **	-
brokerage	42% **	42% **	61% **	60% **	69% **	57% **	65% **	45% **	67% **	21% **	100% **	73% **	76% **	-59% **	99% **	91% **	-46% **

** Correlation is significant at the 0.01 level. * Correlation is significant at the 0.05 level.

quite high Spearman correlation coefficients with all these CK metrics.

7.6 Providing Estimates

In this section we discuss how it is possible to estimate some values for the metrics starting from the knowledge of the analytical fitting functions. We assume that all the data are known for one system release, and assume the persistence of the distributions across releases.

Let us consider, for instance, the metric Ties, and the Eclipse releases from 2.1 to 3.3. Let us start with the lognormal distribution. If we compute the estimate of the mean values using the best fitting parameters found, using the usual formula ($exp(\mu + \frac{\sigma^2}{2})$), they match actual values with an error of about

Table 7.7: Netbeans 3.2. Spearman correlation among metrics

	numissue	numbug	LOCS	WMC	RFC	LCOM	CBO	fanin	fanout	reachAefficiency	effsize	closeness	dwreach	infocentrality	size	ties	nweakcomp
numbug	98%**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
LOCS	47%**	46%**	%**	-	-	-	-	-	-	-	-	-	-	-	-	-	-
WMC	44%**	42%**	87%**	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RFC	44%**	42%**	87%**	95%**	-	-	-	-	-	-	-	-	-	-	-	-	-
LCOM	41%**	39%**	70%**	87%**	81%**	-	-	-	-	-	-	-	-	-	-	-	-
CBO	33%**	32%**	58%**	53%**	71%**	43%**	-	-	-	-	-	-	-	-	-	-	-
fanin	13%**	12%**	19%**	34%**	30%**	31%**	13%**	-	-	-	-	-	-	-	-	-	-
fanout	45%**	44%**	65%**	58%**	68%**	54%**	71%**	3%*	-	-	-	-	-	-	-	-	-
reachAefficiency	17%**	16%**	18%**	14%**	18%**	15%**	18%**	-19%**	49%**	-	-	-	-	-	-	-	-
effsize	38%**	36%**	52%**	58%**	62%**	54%**	53%**	56%**	70%**	20%**	-	-	-	-	-	-	-
closeness	38%**	36%**	43%**	43%**	46%**	43%**	38%**	14%**	70%**	73%**	61%**	-	-	-	-	-	-
dwreach	37%**	35%**	45%**	45%**	48%**	44%**	40%**	17%**	73%**	77%**	66%**	94%**	-	-	-	-	-
infocentrality	-26%**	-25%**	-30%**	-23%**	-23%**	-24%**	-14%**	9%**	-38%**	-37%**	-26%**	-54%**	-60%**	-	-	-	-
size	39%**	38%**	55%**	61%**	65%**	57%**	56%**	57%**	73%**	23%**	98%**	64%**	69%**	-28%**	-	-	-
ties	39%**	37%**	54%**	57%**	60%**	53%**	50%**	48%**	65%**	10%**	83%**	52%**	57%**	-47%**	88%**	-	-
nweakcomp	-26%**	-25%**	-37%**	-32%**	-32%**	-31%**	-23%**	-16%**	-30%**	12%**	-25%**	-17%**	-20%**	60%**	-36%**	-63%**	-
brokerage	38%**	37%**	53%**	59%**	63%**	55%**	54%**	56%**	71%**	21%**	100%**	62%**	67%**	-28%**	99%**	84%**	-30%**

** Correlation is significant at the 0.01 level. * Correlation is significant at the 0.05 level.

Table 7.8: The best fitting parameters for the three different distributions for the metric Ties. For each version of Eclipse, empirical first and second moment, number of CU and maximum value are also reported.

Ties		lognorm		Pow-law		Y-S				
Rel	μ	σ_{LN}	α_{PL}	x_0	α_{YS}	c	$\#CU$	$< x >$	$< x^2 >$	x_{max}
2.1	2.85	1.54	2.38	174	2.23	20.6	7545	59.6	227.3	9799
3.0	2.80	1.54	2.39	141	2.21	19.1	10288	59.2	257.6	11901
3.1	2.85	1.56	2.37	143	2.16	18.6	11854	64.9	294.7	14711
3.2	2.82	1.57	2.35	141	2.14	17.7	14138	65.3	316.2	17029
3.3	2.83	1.57	2.33	145	2.13	17.4	15439	66.8	336.3	18819

Table 7.9: Estimates for the extreme values of the metric Ties. In the last column the two values refer to the estimate obtained using parameters from release 2.1, or using parameters from the immediate previous version, respectively.

Rel	Actual	$< x_{max} >_{logn}$	$< x_{max} >$ (Eq. 7.5, α_{p-law})	$< x_{max} >$ (Eq. 7.5, α_{YS})
3.0	11901	12962	12268	12609 / ==
3.1	14711	13634	13594	14148 / 14234
3.2	17029	14511	15446	16327 / 16838
3.3	18819	14967	16463	17539 / 18363

15% (see Table 7.9). With regard to the standard deviation, however, the estimate of the lognormal fails. In fact, empirical data show a systematic increase of their standard deviation, while the lognormal provides a constant value, since the best fitting parameters are almost constant.

It is also possible to estimate the expected maximum value for a lognormal population of finite size n, which depends on n, using the formula [68]:

$$log(x_{max}) = \mu + \sigma\sqrt{2log(n)} - \sigma\frac{(loglog(n) + log(4\pi))}{2\sqrt{2log(n)}} + \epsilon \qquad (7.3)$$

where ϵ is a small error term. We approximated our estimates using the first two terms, since the third is negligible in our case. The predicted extreme values for the Ties distribution are reported in Tab. 7.9, which shows a discrepancy with the empirical values of about 15/20 %, which increases with the system size.

If we consider the best-fit power-law distribution, its exponent α_{PL} has always values between 2 and 3, and this is consistent with the power-law property that, for such values of α, the mean is finite, while the standard deviation diverges. In the case of a finite number of samples, this means that the standard deviation has obviously a finite value, but it tends to increase with the number

of samples [57]. This is exactly the behavior which we observed. Therefore, when the number of CU increases from a release to another, so does the standard deviation. Note that the power-law cannot fit the bulk of the data, since the cut-off starts at about 140. So, it cannot be used to estimate the mean of the samples.

Using the power-law, however, we may provide an estimate for the maximum value, a quantity more relevant than the estimate of the mean. It is well known that the following formula holds [57]:

$$< x_{max} > \sim n^{\frac{1}{\alpha-1}} \tag{7.4}$$

so, for two generic releases we can write:

$$\frac{< x_{max_1} >}{< x_{max_2} >} = (\frac{n_1}{n_2})^{\frac{1}{\alpha-1}} \tag{7.5}$$

and we can use one extreme value measured from release i to estimate the extreme value of release $i+1$, when CU numbers are known. Using the values in Table 7.8 the error is about 15%, as reported in Table 7.9.

The Yule-Simon distribution is a good compromise between the two other considered distributions, because it fits both the bulk and the tail of the data. We numerically estimated the average using the best fitting parameters of the Yule-Simon distribution in Table 7.8, and they are in agreement with the empirical values. The power-law exponent obtained from the Yule-Simon best fit is among two and three, and it is consistent with the empirical standard deviation, which seems to diverge with the number of CUs. Furthermore, since Eq. 7.4 holds asymptotically, we can use the power-law exponent as obtained from the Yule-Simon best fitting in Eq. 7.5, to estimate the extreme values as before. These are in excellent agreement with the empirical results (Table 7.9).

We may now answer to the third research question **R**3: *Is it possible to use these distributions to estimate the metrics values in subsequent releases?*

We found that mean values, as obtained from the analytical distributions, are in agreement with the empirical ones. From the knowledge of the best fitting parameters of the Yule-Simon distribution in one release, assuming persistence, we estimated the extreme values of subsequent releases using the CU number. Such estimates are in agreement with the empirical values with an error of $\frac{\Delta x}{x} = 456/18819 \simeq 2.5\%$.

These results have been obtained for the metric Ties for Eclipse but similar considerations hold also for the other metrics which are best fitted using Yule-Simon distribution.

7.7 Conclusions

In this paper we studied for the first time the distribution of SNA metrics in OO software networks, comparing their properties with those of CK metrics and other graph-related metrics. We used as a central concept the Compilation Unit and not the class, to be able to better study the impact of metrics on Bugs and Issues, which always refer to CUs and not to classes, in commonly used configuration management systems.

The empirical distributions of all the studied metrics systematically present power-laws in their tails. This property holds also for bug distribution. It must be noted that bug distributions may be biased due to the lack of information in CVS commits, thus our results on bug distributions are as reliable as the information about bugs extracted from CVSs. All metrics have very similar features and shapes across all the system releases, and also show very similar behavior in both Eclipse and Netbeans systems.

We found analytical distribution functions suitable for fitting the empirical data. Power-law always outperforms other fittings in the tails, whereas Yule-Simon distribution follows the shapes of most metrics empirical distributions very well. In particular, Ties and Fan-in metrics are fitted by Yule-Simon distribution from the very beginning of values, the determination coefficients being over 0.98. We have shown – using the metric Ties – how it is possible to provide reliable estimates for averages and extreme values of subsequent releases from the knowledge of the best fitting parameters and system size. The knowledge of extreme values of metrics could be exploited to keep under control the quality of software systems, because in general high values of these metrics denote high coupling among classes.

Regarding correlations among SNA metrics and Bugs, they are generally good, and when using the Spearman coefficient to assess them, they are comparable to those of CK metrics. It is known that LOC is one of the metrics best correlated with the number of defects. Nevertheless, as it holds for some other complexity metrics, they focus only on single software elements, while the use of SNA metrics allows to take into account the role of interactions between elements, and how these interactions correlate with defects. Consequently, we can state that the new SNA metrics are worth studying in greater detail, to better assess their predictive power regarding Issues and Bugs, maybe in conjunction, and not as an alternative to more traditional OO metrics.

Future developments of this seminal work will include controlled experiments to better understand the effect of SNA metrics on bug proneness and if they are able to identify different kind of bugs, and the construction of software graphs where the link direction and type is taken into account.

Chapter 8

Analysis of SNA metrics on the Java Qualitas Corpus.

In the previous section we analyzed the statistical properties of SNA metrics on software networks, showing how fat-tail empirical distributions are always very good representations of the empirical data. Next we present the analysis of the software graphs of 96 systems of the Java Qualitas Corpus, obtained parsing the source code and identifying the dependencies among classes. We analyzed 12 software metrics on these 96 graphs, nine SNA, and three more traditional software metrics, such as Loc, Fan-in and Fan-out. We analyzed their correlations at system level, and studied the correlation statistics at data-set level.

The analysis shows that these correlations are independent from the specific software system and are general properties of Java software systems. We will see how the metrics can be partitioned in groups for almost the whole Java Qualitas Corpus, and that such grouping can provide insights on the topology of software networks.

For two systems, Eclipse and Netbeans, we computed also the number of bugs, identifying the bugs affecting each class, and finding that some SNA metrics are highly correlated with bugs, while others are strongly anticorrelated. This suggests that practitioners and software engineers might take advantage of such metrics to keep control of software quality.

8.1 Related Works.

Only recently, Zimmermann and Nagappan used SNA metrics to investigate a network of binary dependencies [87]. If we restrict the attention to the study of OO software systems, only Tosun et al., to the authors' knowledge, applied SNA metrics to OO source code to assess defect prediction performance of these

metrics [74]. In particular, there are no studies investigating the correlations among SNA metrics, traditional metrics, and Bugs metrics.

Here we extend the analysis performed by us and conjugate the network approach with the need to measure software [47], using different SNA metrics. These metrics are useful to characterize the different roles of nodes in the network, their importance or responsibilities, to quantify how much a node interacts with other nodes, and so on.

The motivations for this study come from the lack of researches devoted to the investigation of Object Oriented software systems as a network of nodes exchanging information by means of links supporting messages, and of the role of such nodes in the information flow. From this point of view, the software network is very similar to a social network, where the nodes are individuals and the links represent social connections among these individuals.

From a software engineering point of view, such analysis may provide insights on how much a class is used by other classes, and about what measures are important in order to characterize the modular structure of the system. We also studied, on two large systems for which we already had bug information available the correlations between SNA metrics and bugs, highlighting interesting properties.

The goals of this study are thus to analyze the correlations among these new software metrics in Java systems, which is yet an unaccomplished task, to identify general rules valid for all analyzed Java systems, and to investigate new factors related to software quality.

Consequently, the research questions we address in this paper are the following:

- **R**Q1 Are the correlations among SNA metrics system-dependent, or are they generally the same for all Java systems?

- **R**Q2 Does the social role of a node influence its bug proneness?

- **R**Q3 Are there SNA metrics useful for measuring software quality in Java systems?

- **R**Q4 Are the correlations among SNA metrics related to a particular software structure?

8.2 The Dataset and SNA metrics for the software networks.

The systems analyzed are 96 systems from the Java Qualitas Corpus [61], for which the source code was easily available. This is a large set of Java open source projects [73], conceived for the most disparate purposes, and having different sizes, aimed to provide a standard dataset to improve repeatability of results in empirical software engineering. Their sizes range from 46 to 3512 classes, with an average of 690, for a total of about sixty five thousand classes and six million Loc. Furthermore we analyzed two large software systems, Eclipse 3.0 [3] and Netbeans 4.0 [4], with about 12500 and 15000 classes respectively.

For each system, we parsed the source code in order to identify classes and class dependencies, as defined above. These play the role of network nodes and edges of our software networks, respectively. Our analysis is thus a static analysis, and we do not consider run-time dependencies. In these Java systems, from a structural point of view, classes depend on one another because of *inheritance*, *composition* and *dependence* relationships. In fact, a class may depend one another because its code calls methods defined in other classes, or uses temporary variables belonging to other classes. The former is called *data dependency*, the latter *call dependency*.

After parsing the source code, the result is an oriented graph, where nodes are the classes and interfaces, and edges are class dependencies. On this software network we computed, for all the 98 systems, the following metrics: Loc, Fan-in, Fan-out, dwReach, Closeness, Brokerage, Ties, Size, nWeakComponent, infoCentrality, reachEfficiency, EffectiveSize.
The first three are traditional software metrics, the latter nine are used in Social Network Analysis. Some of them are also called EGO metrics, since they refer to the EGO-Network, a sub-network composed by a given node, the EGO, and all the nodes directly connected to it. We did not compute CK metrics, but choose Loc and Fan-out, because they are correlated to the CK metrics known to have high correlation with bugs, namely WMC, CBO and RFC. For computing the SNA metrics, we used un-directed edges.

All previous metrics are among those studied in [87]. For Eclipse and Netbeans we computed also the number of bugs in each class.

For each class we obtained 12 metrics (plus bugs for Eclipse and Netbeans classes), and we analyzed their Spearman cross-correlations inside each system. We obtain, for each system, a 12 by 12 Spearman correlation matrix, with 66 independent entries. Then we studied how these correlation coefficients are distributed among the 98 systems. In general, we found that there are metrics for which the correlations values are centered among a mean value with little

variance. This means that these metrics are roughly equivalent for describing the system properties, *and this holds for all studied Java systems*. We found that, out of the twelve metrics, six or seven different groups may be identified. In some other cases, the correlations values are not clustered among a central value, but are spread across almost all the allowed range [-1 , +1]. Finally, there are also particular cases for which the correlations are almost the same for all Java Qualitas Corpus systems, but with a few very different or opposite values of the correlations. This analysis was repeated twice. One time using all the Java Qualitas Corpus, another time using only the systems with more than 1000 classes. Since there are several systems with a few classes (36 systems with less than 300 classes), the overall statistics of the correlation coefficients are affected by larger fluctuations for such small systems, making the data quite noisy. Repeating the same analysis retaining only the largest systems allows to reduce such noise and to identify general features more clearly.

Another approach for identifying multivariate correlations among these twelve metrics is by mean of a PCA (Principal Component Analysis). Thus we performed a PCA for the software systems with more than 1000 classes, along with a cluster analysis, in order to identify groups of metrics with another tool. The PCA allows to quickly identify the major dimension of the variance and the linear covariances among the different metrics. It must be noted, however, that the statistical distributions of the metrics across each software system analyzed are usually fat-tail distributions, as examined in the previous chapter. In these cases the computation of a non linear cross-correlation may be more appropriate. Thus the PCA and the Spearman cross-correlation analysis may in principle provide different results.

8.3 PCA and Cluster Analysis

We performed a PCA on the 23 software systems with more than 1000 classes, in order to have enough statistics. The main indication provided by this analysis is that all the systems present a main principal component, explaining the majority of the variance, and a second smaller component, which together explain always about 95% of the variance. These are accompanied by a very small third component, which only in three cases is above 4%. Only one system (checkstyle) presents the exception of two principal components almost equivalent. The results are reported in Tab. 8.1 and show that all the software systems have highly homogeneous features.

The clustering shows how the dwReach and Loc metrics provide major contributions to the first and to the second principal components, respectively. In Fig. 8.1 we present an example of a typical situation for the system Jrefactory,

Table 8.1: Percentages of the variance explained by the first three principal components (PCs).

System	1st PC(%)	2nd PC(%)	3rd PC(%)
ant	88.233	9.2252	1.7530
argouml	90.993	6.2521	2.1902
aspectj	80.241	14.681	4.1342
azureus	90.684	6.7305	2.1945
checkstyle	55.458	40.782	1.8304
columba	87.778	8.3074	3.6385
compiere	79.256	17.205	2.8861
derby	83.833	10.046	4.8409
eclipse 3.0	93.982	3.8925	1.9505
exoportal	85.941	11.600	1.9112
gt2	87.815	9.4388	2.3476
hibernate	90.822	6.9576	1.9135
jedit	84.032	12.384	2.5257
jena	91.167	4.9201	3.4452
jrefactory	71.830	24.015	2.3003
jtopen	80.206	16.738	2.5273
nakedobjects	90.335	6.0823	2.9967
netbeans 4.0 ide	86.956	11.755	0.7101
sandmark	86.251	10.350	3.0362
springframework	91.555	6.6683	1.3424
squirrel sql	88.846	8.4011	2.1022
tomcat	82.683	14.087	2.3919
xalan	74.625	20.063	4.0803

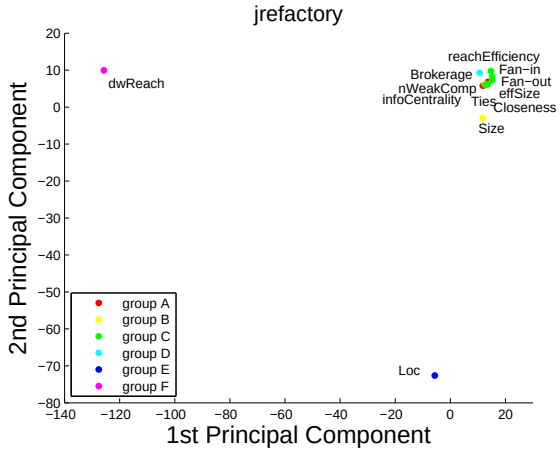


Figure 8.1: First and second Principal Components for the system Jrefactory. Different points correspond to different metrics and the same color stands for metrics belonging to one same group after clustering.

where we used six groups for the clustering.

This result can be quite naturally explained for the Loc, since it is the only metric that is not influenced by the graph structure of the software system, and thus is well separated from the others. The reason for the separation of the dwReach is more obscure, and further analysis is necessary. Looking at the third principal component also the Brokerage appears well separated from the other metrics, as in reported in Fig. 8.2, for the same system.

This is due to the fact that Brokerage grows approximately in a quadratic fashion with Size, and a PCA reveals such a non linear correlation. The Spearman correlation coefficient instead take into account non linear correlations and provide different result, as we will discuss later on.

8.4 Statistics of Correlations and Bugs

We started by computing the correlations among all the 12 metrics, for a total of 66 independent values for each of the 98 systems. Then we analyzed the statistics of these 66 variables within a population of 98 samples, in order to gain insights about general features and properties of the metrics for the Java code.

We used the boxplot representation in order to obtain information on the statistics of correlations among their metrics. We also computed the mean and the STD statistics. The boxplot is a visual representation in which a box with whiskers is reported for each set of data. The boxes have lines at the lower

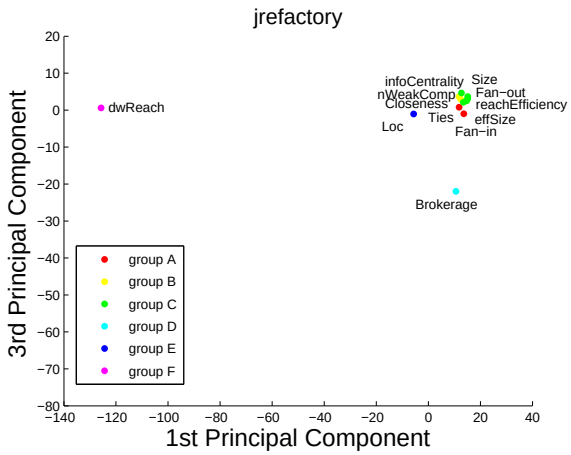


Figure 8.2: First and third Principal Components for the system Jrefactory. Different points correspond to different metrics and the same color stands for metrics belonging to one same group after clustering.

quartile, median, and upper quartile values. The whiskers are lines extending from each end of the boxes to show the extent of the rest of the data. Outliers are data with values beyond the ends of the whiskers, and are represented by crosses. The maximum whisker length has been set to 1.5.

First, we computed the boxplots for the correlations among all the 12 metrics for the whole Java Qualitas Corpus, and then retaining only the 23 systems with more than 1000 classes, for a total of 24 boxplots. For convenience, we choose to report only few representative cases: the boxplot for the correlations among Size and the other metrics, for the whole Java Corpus (Fig. 8.3) and for the largest 23 systems (Fig. 8.4), and the boxplots only for the largest 23 systems of the correlations for Closeness (Fig. 8.5), infoCentrality (Fig. 8.6) and Loc (Fig. 8.7), since we found that they may be considered representative of different groups of metrics.

The figures show how some pairs of metrics present almost the same values for their cross-correlation, with a little spread around the median, while other pairs have values ranging from negative to positive values among the 98 Java systems, with a large spread. From the correlations we can identify four different groups of metrics.

Fig. 8.3 shows how the metrics Ties and Size show almost perfect correlation in all the systems, and are very highly correlated also with Brokerage and effSize. These four metrics can be included into a single group, carrying basically the same information. The picture is even clearer if we consider Fig. 8.4, reporting only the 23 Java systems with more than 1000 classes, where fluctuations of

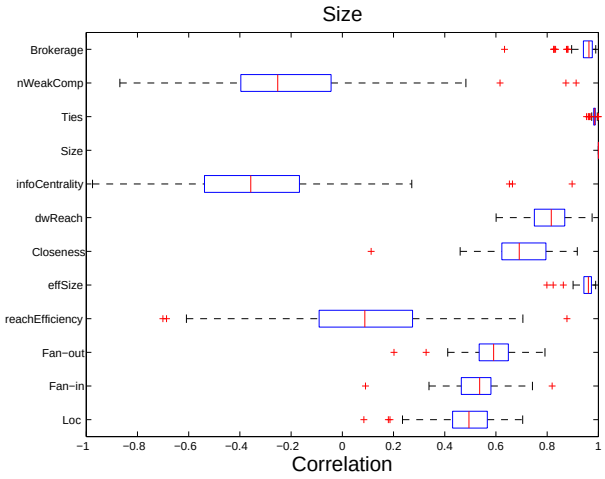


Figure 8.3: Boxplot of the correlations among Size and the other metrics for all the Java Qualitas Corpus.

the cross-correlations due to the smallness of systems are largely reduced. In this case, the number of outliers is reduced too, and the cross-correlation values for these four metrics are almost the same in every system, suggesting that this result does not depend on the particular Java system analyzed but is a general property of open source Java systems, analyzed as software networks.

A second group can be identified for the metrics Closeness and dwReach, which have correlation above 0.8 for almost all the Java systems, with a few outliers presenting in any case correlation around 0.4. If we consider only the 23 systems with more than 1000 classes (Fig. 8.5), there is only one outliers with correlation around 0.4, while all the others have correlation around 0.8 or above. Furthermore, the correlations of these two metrics with the other metrics are always very similar, thus Closeness and dwReach can be considered in general equivalent metrics for describing the software graph of the examined Java systems.

A third group includes infoCentrality and nWeakComp. They are well correlated in all the Java Corpus, with two exceptions, and well correlated for all the systems with more than 1000 classes (Fig. 8.6). With regard to the other metrics, they are always anticorrelated, apart for few outliers which show strong correlation with other metrics. For the systems with more than 1000 classes, only in one or two system these two metrics have strong positive correlation with other metrics, while in all other systems their correlation with all other metrics is generally negative or zero.

Finally, Loc and Fan-out have high correlation in all the Java Corpus systems, with few outliers, and even higher correlation in the 23 largest systems

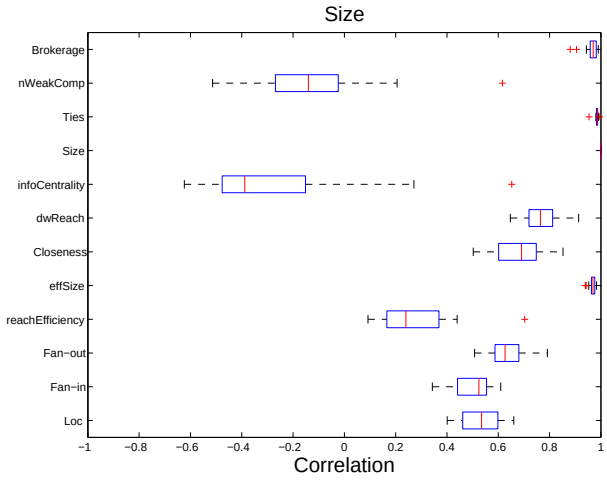


Figure 8.4: Boxplot of the correlations among Size and the other metrics for the 23 largest systems.

(Fig. 8.7), and show high correlation with the metrics of the first group as well.

Next, we discuss how the relationships among the correlations of some metrics can be related to the structure of the software network.

8.4.1 Metrics Size, Ties, Brokerage, effSize.

The metrics of the first group are all EGO metrics, namely are computed on the EGO network of a given node. Given a Size N, since we consider un-directed links and exclude multiple links, the value of Ties ranges from a minimum of N-1, when all the nodes are connected only to the EGO, to a maximum of N(N-1)/2, when the network is completely connected (every node is connected to every other node). The strong correlation among Size and Ties can be easily explained if Ties is, for most nodes, around the minimum value. In fact in such case Ties and Size result, for most nodes, directly proportional to each other, thus determining a strong correlation coefficient. If this were the case, most nodes are expected to form a software network characterized by a star-like structure, a network with 'hubs'. For the central node, Size would be N and Ties would be N-1. For the peripheral nodes, Size would be 2 while Ties would be 1, respecting the same rule. Thus for a network with many hubs, Size and Ties have a correlation coefficient close to one.

With regard to effSize, such a structure would also show strong correlations with Size (and so with Ties). In fact each node would have an average number of ties with other nodes of the EGO of (N-1+N-1)/N = 2-2/N. effSize would then be N - 1 - 2 + 2/N, for a Size N. This provides a direct proportionality even for N

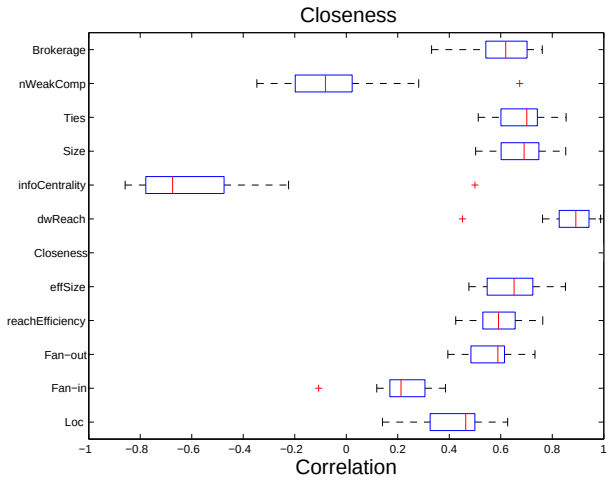


Figure 8.5: Boxplot of the correlations among Closeness and the other metrics for the 23 largest systems.

not too large, explaining why effSize is to be expected strongly correlated with Size, and to a lesser extent, with Ties.

Finally, considering the Brokerage in the hypothesis of hub-structured networks, we have to distinguish two cases: the central node and the peripheral ones. For the central node Brokerage is of $O(N^2)$. For other nodes, which are the majority, the EGO network consists only of two nodes, and Brokerage is zero. For these same peripheral nodes also the other metrics are small (Size is two, Ties is one, effSize is 0.5). Supposing that N is in general not too large, we would again obtain a strong correlation among Brokerage and the other three EGO metrics.

8.4.2 Metrics Closeness and dwReach.

Closeness and dwReach are not EGO metrics, and their computation involves all the network's nodes. Their high correlation can be again understood with the hypothesis of a network characterized by many hubs, or with many star-like structures. In fact, they can be computed using the following relationships: Closeness = $1/(n_1 + 2n_2 + 3n_3...) = 1/(\sum_i i * n_i)$, dwReach = $n_1 + n_2/2 + n_3/3...$ = $\sum_i n_i/i$, where n_i is the number of nodes at a distance of i steps from the given node, and with a proper normalization factor for Closeness. Since the total number of nodes in the network is fixed, the sum of n_i is a constraint of the system. So, both Closeness and dwReach have maximum value for nodes at distance one to all other nodes, and decrease for nodes with a different topology. In the case of star-like structures, for peripheral points both Closeness and

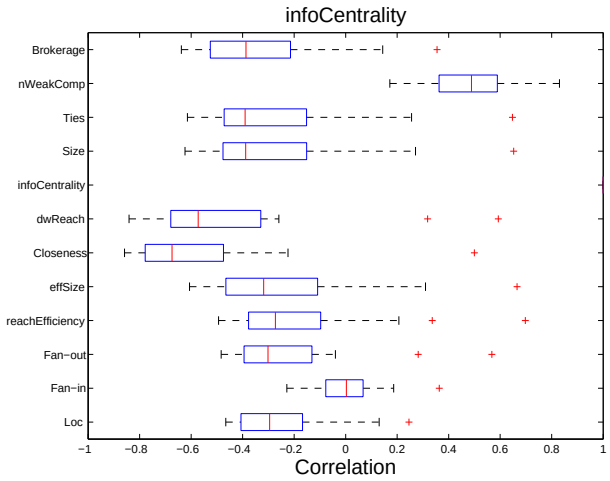


Figure 8.6: Boxplot of the correlations among InofCentrality and the other metrics for the 23 largest systems.

dwReach decrease markedly, while this does not occur for dense (highly connected) networks. Thus, according to our hypothesis, our hub-structured software networks would possess points with a large spectrum of different values for Closeness and dwReach, with the peripheral points having the lowest values for both metrics, and with the hubs having the largest values for both metrics. This may explain the large correlation observed in our software networks.

8.4.3 Metrics nWeakComp and infoCentrality.

The high correlation between nWeakComp and infoCentrality is somehow more surprising. In fact, the first is an EGO metric while the computation of the second requires all network's nodes. The metric nWeakComp is normalized with Size of the EGO network. Thus, being inversely proportional to it, it is somehow anti-correlated with Size. This is confirmed by the empirical results. On the other hand, infoCentrality is proportional to the harmonic mean of the length of paths starting from all network's nodes. Thus, it may be computed according to $1/(n_1 + n_2/2 + n_3/3...)$, which is approximately inversely proportional to the metric dwReach, and to all the metrics positively correlated to it. This explains why nWeakComp and infoCentrality are positively correlated on the large.

8.4.4 Metrics Loc and Fan-out.

Finally, we consider the fourth group. Loc and Fan-out are not SNA metrics; one represents an inner metric, while the other measures to what extent a class

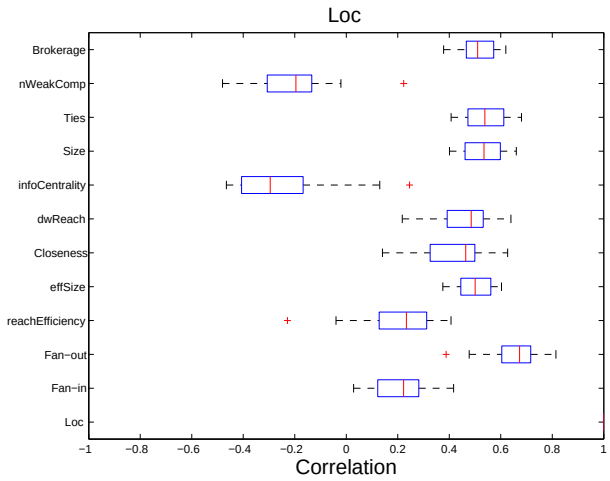


Figure 8.7: Boxplot of the correlations among Loc and the other metrics for the 23 largest systems.

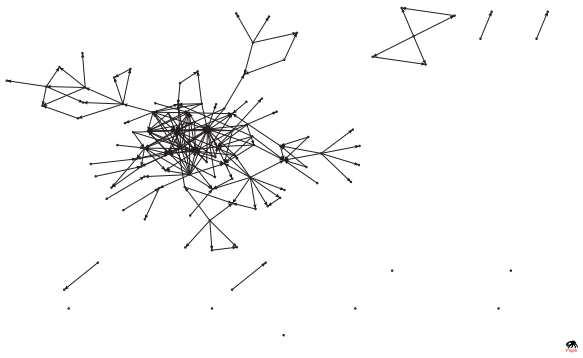


Figure 8.8: Graph representation of the software system *Cobertura*.

uses other classes. The high correlation among the two is quite natural, since the larger the Loc of a class, the larger is the chance of finding lines of code containing calls to methods or variables belonging to other classes. On the contrary, Fan-in is not so correlated to Loc, since for a class, to have many lines of code does not directly imply to have a large chance of being used by other classes.

In Figures 8.8 and 8.9 we show some examples of software graphs, for systems with a relatively small number of classes. These graphs confirm our hypothesis of a software network made of many central hubs, to which the other classes generally refer, with only a few links among each other.

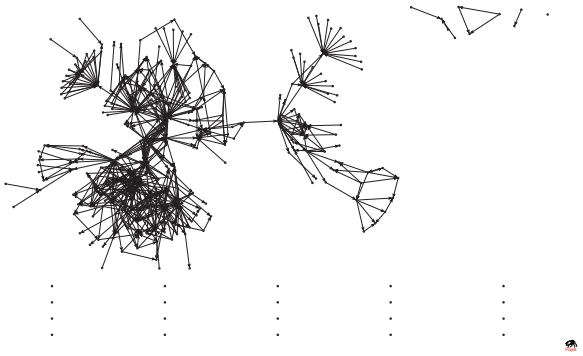


Figure 8.9: Graph representation of the software system *Drawswf*.

8.4.5 Metrics in Random Graphs.

In order to verify these considerations, we performed different simulations in which we generated networks with a particular topology, such as random networks and scale-free networks, and analyzed the correlations among the SNA metrics on these networks.

Figure 8.10 reports the correlation values among the metrics Size and Ties, Brokerage, effSize, for random networks with different parameters p. We built 30 random networks, with 100 nodes, for each value of parameter p, which represents the probability that a couple of nodes is connected by one edge, for different values of p among zero and one. For fixed p, we calculated the correlations among the metric Size and the metrics Ties (S-T), Brokerage (S-B), effSize (S-E), respectively, for the corresponding 30 networks, and measured their minimum and maximum values. For convenience, in Figure 8.10 we report, for S-B and S-E, only the maximum values at varying p. It can be seen that the correlations are always close to 1, and are close to 0.9 only for small values of p, namely for sparse networks, as it is the case of our Java software network. With regards to scale-free networks, we obtained for different networks correlation values between 0.8 and 0.9 (not reported in the Figure).

Thus, the correlations among the SNA metrics of the first group are similar to those of sparse random networks and of scale-free networks.

8.4.6 Correlations with Bugs.

Eventually, we consider the correlations among the 12 metrics and bugs, only for the systems Eclipse and Netbeans, whose bugs are available in their bug tracking systems. Table 8.2 reports the correlation coefficients.

In this case, we do not have a statistical sample of 98 systems, thus our con-

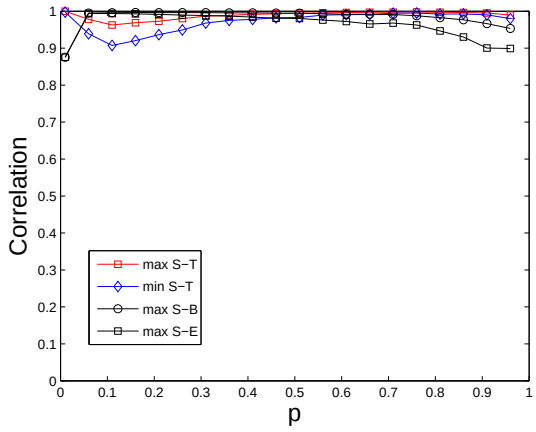


Figure 8.10: Correlations among four SNA metrics in random graphs.

siderations hold only for Eclipse 3.0 and Netbeans 4.0, even if they show some common features. First of all, we may note that correlations among the 12 metrics and bugs confirm the grouping discussed above. Size, Ties, Brokerage and effSize all show the same correlation with bugs, for both Eclipse and Netbeans, even if they have slightly larger values in Eclipse. This suggests that the larger are these metrics, the more are the bugs in the corresponding classes. Since in the examined software networks we found these metrics equivalent, we may discuss this result referring only to metric Size. The number of bugs in a class is computed through the code correction commits reported in repositories such as CVS. These corrections may be due to errors in the code of the given class, but also to changes applied in order to realign the code to code changed in other classes for bug fixing. Thus, the larger is Size, the larger is the probability of such events, and thus the number of bug corrections associated to the class. This reasoning is confirmed also by the high correlation of bugs with Loc and Fan-out, and by the lower correlation with Fan-in. In fact, the larger is Loc, the larger the probability that the code in the class makes calls to code in other classes, and thus the larger is the Fan-out as already discussed previously. But a large Fan-out implies a large chance to make code adjustments in the class when the code of the called class is modified for bug fixing. This causes a report in the CVS commits signaling a bug correction in the calling class. On the contrary, in the case of large Fan-in this mechanism does not work. Let A be a class with an In-link from another class B, a change in the code of class B does not generally imply the need to change the code in class A.

dwReach and Closeness show relatively high correlation with bugs. Since these two metrics are centrality metrics, namely represent how much a class is

Table 8.2: Correlation coefficients among metrics and bugs.

Eclipse rel 3.0	Loc 0.53	Fan-in 0.20	Fan-out 0.48
	reachEff 0.16	effSize 0.45	Closeness 0.39
	dwReach 0.41	infoCen -0.33	Size 0.47
	Ties 0.48	nWeakComp -0.33	Brokerage 0.46
Netbeans rel 4.0	Loc 0.40	Fan-in 0.19	Fan-out 0.37
	reachEff 0.23	effSize 0.36	Closeness 0.29
	dwReach 0.34	infoCen -0.17	Size 0.37
	Ties 0.37	nWeakComp -0.15	Brokerage 0.37

closely connected to all other classes, the larger are these metrics the more the class is connected to the rest of the network. This case can then be subsumed into the previous one.

Perhaps, the most interesting result is the anti-correlation of bugs with nWeakComp and infoCentrality. This means that the number of bugs is significantly reduced in classes with larger values of these two metrics, that, as we already observed, are strongly correlated to each other.

In general, these results show that the metrics related to the number of classes directly connected to a given class are the most correlated to bugs. Thus a class which play the role of a hub, or a star center, presents more chances to be 'buggy' or maybe that its bugs are actually discovered and fixed.

We can thus answer to our research questions:

- **R**Q1 *A*re the correlations among SNA metrics system-dependent, or are they generally the same for all Java systems?

  The answer to this question depends on the particular metric. We found that it is possible to categorize some SNA metrics in groups. Thus for the SNA metrics belonging to a group the answer is positive. In particular we find that the high correlations among four out of the five EGO metrics, Size, Ties, Brokerage and effSize are a general property of the Java systems, since the Java Qualitas Corpus represent a general statistical sample of Java software systems. These high correlations, moreover, hold also

for randomly generated networks.

The answer is positive also for Closeness and dwReach, because the analysis on the 23 largest systems show a little spread in the distribution of their correlation.

The answer is negative for the metrics of the third group, nWeakComp and infoCentrality, because even the data on the 23 largest Java systems show a large spread of the correlation distribution, meaning that the correlation value depends on the system analyzed.

- **R**Q2 *D*oes the social role of a node influence its bug proneness?

  Our classification in groups for the SNA metrics provides a positive answer to this question. In fact, since the EGO metrics are the most correlated with bugs, they can be used to argue that central nodes of the software network, for which Size and Brokerage take large values, are, in general, more bug affected, while peripheral nodes are less bug affected.

- **R**Q3 *A*re there SNA metrics useful for measuring software quality in Java systems?

  On the basis of the correlations obtained among bugs and SNA metrics, we can answer positively to this question. In fact, the SNA metrics belonging to the first group are well correlated with bugs, thus they can be used as rough indicators of software quality, measured in terms of bugs. Software nodes with high values of these SNA metrics have generally a larger number of bugs than other nodes.

  The same holds for dwReach and Closeness, even if to a lesser extent.

  Of particular interest is the case of nWeakComp and infoCentrality. Since these metrics look anti-correlated with bugs, a high value for these SNA metrics is a hint of good software quality, in terms of the number of bugs affecting the corresponding software nodes.

- **R**Q4 *A*re the correlations among SNA metrics related to a particular software structure?

  On the basis of our data we cannot answer to this question. In fact, while we described how the correlations among these metrics can be related to the software graph topology, our simulations on random and scale-free networks also show high correlation values among the same SNA metrics. Thus, we are not able to provide a statistically significant counter-prove that a different network topology shows different correlations among the SNA metrics. Further work is needed in order to answer this question.

8.5 Conclusions

We reported the analysis of twelve metrics computed on software graphs. Our results extend some previous findings about metrics correlated to bugs, like Loc and Fan-out. In particular, we extended the use of SNA metrics to Java software networks.

With the aim of finding new software quality indicators, we presented new metrics for measuring software systems, widening a field of research in which more traditional software metrics like the CK suite have been already investigated, providing contradictory results. A significant result is that some SNA metrics (Size, Ties, Brokerage and effSize) are consistently highly correlated for all Java systems studied, and also for networks randomly generated. As a consequence, we propose to use only size as best representative of these EGO metrics, dropping the use of other metrics correlated to it in future works on the application of these metrics to software systems.

We found that some structural metrics look negatively correlated with bugs, which may provide alternative insights to practitioners, with respect to the more popular and largely used metrics positively correlated to bugs.

Among the contributions of this analysis there is the finding that some SNA metrics can be useful as software quality indicators and that the role and the position of the software nodes are related to bug proneness. The analysis considered only a subset of all the metrics used in literature for measuring OO software systems. We believe that an extension of this analysis to other OO metrics can provide further insights on the quality of such software systems.

Chapter 9

Three Algorithms for Analyzing Fractal Software Networks

The last metric we propose for characterizing software quality is the Fractal Dimension of software networks. In this chapter we present an algorithm for computing the fractal dimension of a software network, and compare its performances with two other algorithms. Object of our study are various large, object-oriented software systems. We built the associated graph for each system as usually, analyzing the binary relationships (dependencies), among classes. We found that the structure of such software networks is self-similar under a length-scale transormation, confirming previous results of a recent paper from us. The fractal dimension of these networks is computed using a Merge algorithm, first devised by the authors, a Greedy Coloring algorithm, based on the equivalence with the graph coloring problem, and a Simulated Annealing algorithm, largely used for efficiently determining minima in multi-dimensional problems. Our study examines both efficiency and accuracy, showing that the Merge algorithm is the most efficient, while the Simulated Annealing is the most accurate. The Greeding Coloring algorithm lays in between the two, having speed very close to the Merge algorithm, and accuracy comparable to the Simulated Annealing algorithm.

In the next chapter we will use these results to analyze the opportunity of using the software network fractal dimension in order to characterize software quality in terms of number of bugs in the system.

9.1 Related Works.

It is already well known that software networks have the characteristics of complex networks, i.e. are scale-free and small-world [48] [76], [56], [23]. A recent

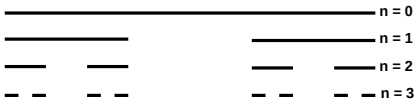


Figure 9.1: Fig. 9.1: Cantor set with three steps.

paper by Song et al. ([70]) demonstrated that the structure of complex networks can also be self-similar under a length-scale transformation, and showed how to calculate their fractal dimension using the "box counting" method. This finding was applied to software networks computed on the classes and class relationships of large Smalltalk and Java systems, which were shown to exhibit a consistent self-similar behavior [22]. Moreover, a significant correlation seems to hold between the fractal dimension computed for various OO systems, and standard metrics related with software quality. It is worth noting that the fractal dimension is just a single number that characterizes a whole network, and hence a whole software system, while complexity metrics are computed on every module of the system - think for instance to Chidamber and Kemerer OO metrics suite [19]. Obviously, the whole system can be characterized by some statistics computed on all modules, but this is not the same of having just one consistent, synthetic measure as with fractal dimension. For this reason, we believe that the fractal dimension of software networks is a significant metric describing the regularity of the software structure. It is therefore important to have efficient and reliable algorithms to compute it. In fact, as it will be shown in the following, the box counting algorithm is NP-complete, and its exact computation for large networks cannot be practically accomplished. Here we briefly recall the concept of fractal set and its application to complex networks, and then we present and compare three different algorithms to compute it - Greedy Coloring, a Merge Algorithm devised by the authors, and Simulated Annealing - discussing the results.

9.2 The Fractal Dimension of Software Networks

9.2.1 Fractals.

Before studying software networks scaling properties we need to introduce the basic concept of fractal dimension. Systems possessing fractal dimensions fill the space in a counterintuitive manner. One of the easy-to-understand and well known examples of fractals is the Cantor set.

Let us consider a segment of unit length. For the sake of clarity, we label its left extreme with the abscissa zero, and its right extreme with the abscissa

one. To obtain the Cantor set we delete pieces of this segment applying an infinite iterative process, and look at the remaining set of points (fig. 9.1). At the first step we subtract the inner third of the segment (any other fraction works well). The remaining set are two segments of length one third each. At the second step we subtract from each of these two segment their central third. The remaining set are four segments of length one ninth each. According to this process, at the n^{th} step we obtain 2^n segments of length 3^{-n} each, and the set's total length is (2/3)n. Let us examine the limit for large n. Clearly the number of segments grows to infinity, each segment becoming of zero length, namely a point. But such set is not a simple set of discrete points. In fact it is not countable. Actually it has the power-of-continuum, namely it is in a one-to-one correspondence with the original unit length segment. The reasoning is the following. Consider the first step. The two segments may be identified with two numbers, zero for the left segment, and one for the right segment. For the segments at the second step we can add another binary digit, again zero for the left segment and one for the right segment. Thus the four segments at the second step are identified by two binary digits: 00, 01, 10 and 11, from left to right. In general, each segment at the nth step may be identified by a sequence of n binary digits. In the limit of large n we can label each point of the remaining set by an infinite sequence of binary digits. But each point of the unit length segment is identified, in binary notation, by the same infinite sequences. In other words, all the points among zero and one may be coded in a fractional binary number. This provides a one-to-one mapping among the original segment and the remaining set of points. Apparently the subtraction of an infinite number of pieces does not modify the segment! Nevertheless they are clearly two different sets. We can also consider another peculiarity. Let us calculate the length of all the subtracted sub-segments. At the first step we subtracted one segment of length one third. At the second step we subtracted two other segments, each of length one ninth. In general at the nth step we have to consider $2^{(n-1)}$ segments of length 3^{-n}. We obtain the following series providing the total length:

$$L_{tot} = \sum_{n=1}^{\infty} \frac{2^{n-1}}{3^n} \tag{9.1}$$

This sum converges to one. Thus at the end we subtracted all the original segment's length! These are the paradoxes we have to face when dealing with fractal objects. Even if the two sets are in a one-to-one correspondence, they substantially fill the available space in a different manner. Roughly speaking, the segment fills all the available space, while the Cantor set (the set of remaining points) leaves lots of holes: it is non-continuous. This is the key observation to define the fractal dimension. In fact, if we partition the available space

in cells, with varying sizes or diameters, we can note the difference among the two kinds of sets if we look at how many cells are filled or empty. In the segment case, regardless of the cells diameter, all the cells will be full. Thus the number of filled cells grows with the inverse of the cells diameter. In the case of the Cantor set this is not true, and the number of filled cells grows with a fractional power of the inverse of the diameter. More formally, in the first case, we simply partition the segment in N identical subsegments of diameter ϵ (in one dimension all the partition cells are simply segments, while in dimension D we can use D- dimensional cubes). The relationship among the two is $N_\epsilon = 1/\epsilon$, and thus the number of filled cells scales with the inverse power of the diameter. If we use a different segment length we obtain a multiplicating factor in front of the previous formula. In two dimensions, instead of a segment we partition a unit square, using ϵ-side subsquares. Their number is given by $N = 1/\epsilon^2$, and scales with the second inverse power of the diameter. Generalizing to dimension D, a unit D-cube will be partitioned by ϵ-side D-cubes, and their total number will be $N_\epsilon = 1/\epsilon^D$, scaling with the D inverse power of the diameter. All these powers are integer, that we are use to call "integer dimensions". In the case of Cantor set, for the covering partition at step n we need $N = 2^n$ cells of diameter $\epsilon = 3^{-n}$. Then the scaling of N with ϵ is:

$$N_\epsilon \simeq \epsilon^{-\frac{log(2)}{log(3)}} \simeq \epsilon^{-K} \tag{9.2}$$

with $K < 1$. This defines the fractal dimension of the Cantor set as $d = \log(2)/\log(3)$, and indicates the scaling of the number of non empty cells with diameter, or, in a sense, how many points fill the space in a range of the diameter, a concept that will be extended for the network's fractal dimension. This way of proceed to calculate the fractal dimension is known as the box-counting method, since the partition divides the space into equal boxes. The value $\log(2)/\log(3)$ clearly indicates that Cantor set does not fill the space nor like a one dimensional segment, neither like a set of disjoint point, whose fractal dimension is zero. The Cantor set, by construction, shows an important signature of fractal sets, the self-similarity: at any length scale the set looks the same. The structure reveals always the same details when viewed at different magnifications, and there is not lower limit, or something like “atomic” components, from which all the set is built. While for mathematical sets this is exactly true at any length scale, for real objects the scaling regime and self-similarity hold only in a limited range of lengths, and finite size or granularity effects are revealed out of these limits.

9.2.2 Fractal Dimension of OO Networks

Fig. 9.2 shows the box counting analysis of the software network related to JDK 1.5.0 Java system. The log-log plot of N_B vs. l_B reveals a self-similar structure.

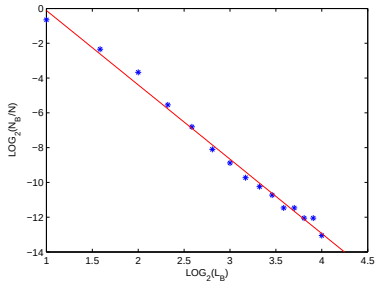


Figure 9.2: Fig. 9.2. Log-log plot of NB vs. lB for JDK 1.5.0.

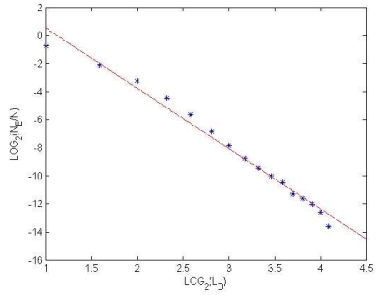


Figure 9.3: Fig. 9.3. Log-log plot of N_B vs. l_B for Eclipse 2.1.3.

The slope of the fit is 4.24; this value is the fractal dimension d_B for JDK 1.5.0

In fig. 9.3 we report the box counting result for Eclipse 2.1.3, whose software graph has partially been shown in the first figure. Also for this system the log-log plot of N_B vs. l_B is clearly linear, and the system exhibits the power-law scaling. The slope provides 4.31 for the box-counting fractal dimension of this software network

In fig. 9.4 we show the same plot for another software system, VWorks 7.3. Once again it shows a self-similar structure and a power-law relationship among N_B and l_B. Here the value provided by the box-counting method for the fractal dimension is 4.54.

9.3 Computing the Network Fractal Dimension

Song et. al. in their first paper [70] do not give details about how they actually computed the fractal dimension. Subsequently, Concas et al. shortly presented a simple algorithm for computing dB [22]. Later, Song et al. demonstrated

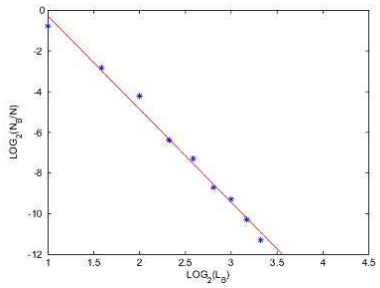


Figure 9.4: Fig. 9.4. Log-log plot of N_B vs. l_B for VWorks 7.3.

that this computational problem is equivalent to the graph coloring problem, and consequently took advantage of the many well-known greedy algorithms to perform this task [69]. Here we compare three algorithms both in terms of performance and precision - greedy coloring as in [69], a merge algorithm similar to that reported in [22], and simulated annealing, which is considered one of the best approaches to find the global minimum of difficult, multi-modal problems.

9.3.1 Greedy Coloring (GC)

Song et al. demonstrate that the box counting problem can be mapped to the graph coloring problem, which is known to belong to the family of NP-hard problems. Vertex coloring is a well-known procedure, where colors are assigned to each vertex of a network, so that no edge connects two identically colored vertexes [50]. We used the greedy algorithm described by Song et al. For this implementation we need a two-dimensional matrix cil of size N by l_{Bmax}, whose values represent the color of node i for a given box size l = l_B. The algorithm works in the following way [69]:

1) Assign a unique id from 1 to N to all network nodes, without assigning any colors yet.
(2) For all lB values, assign a color value 0 to the node with id=1, i.e. c_{il} = 0.
(3) Set the id value i = 2. Repeat the following until i = N.
(a) Calculate the distance l_{ij} from i to all the nodes in the network with id j less than i.
(b) Set l_B = 1.
(c) Select one of the unused colors $c_j l_{ij}$ from all nodes j<i for which $l_{ij} \geq l_B$. This is the color of node i for the given l_B value.
(d) Increase l_B by one and repeat (c) until $l_B = l_{Bmax}$
(e) Increase i by 1.

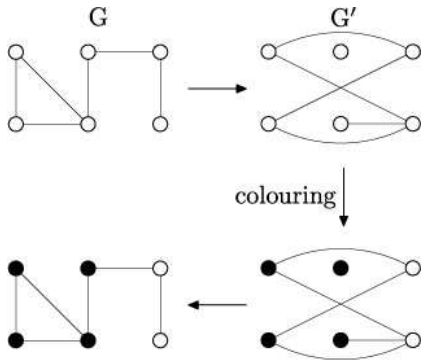


Figure 9.5: Fig. 9.5: Construction of the dual network G' for a given box size (here $l_B = 3$), where two nodes are connected if they are at a distance $l \geq l_B$. We use a greedy algorithm for vertex coloring in G', which is then used to determine the box covering in G, as shown in the plot.

This greedy algorithm is very efficient, since it can cover the network with a sequence of box sizes lB performing only one network pass. The main steps are illustrated in fig. 9.5, where the example shows the case $l_B = 3$. Starting from the network G representing the software graph, we build the dual network G', obtained from G connecting two nodes when their distance, calculated in the original graph G, is larger than, or equal to, l_B. Next we use a greedy algorithm for vertex coloring in G'. Then we go back to the original network determining the box covering on G

9.3.2 Merge Algorithm

This method is based on the union of two or more clusters into a third one. Two clusters are merged if the distance between them is less than l_B. MA uses the configuration at l_B to obtain the starting point for the successive aggregation at $l_{B+1} = l_B + 1$. In the initial configuration each cluster ck contains only a node, so each node is marked with a different label. Let n be the number of nodes of the network, and lmax the maximum value for l_B. The algorithm works in the following way:

$l_B = 2$;
C = c1, c2 , c3 ,..., cn;
while $l_B \leq l_{Bmax}$;
D = C;
repeat:

get a random cluster c_k from C;
C' = c_j C| d(c_k,c_j) ≤ l_B;
get a random cluster c_i from C';
c = merge(c_k,c_j);
C = C - c_k,c_j;
D = D U c;
until size(C) ≤ 2 or C'= C whenever c ∈ C;
D = D U C
N_B = size(D);
l_B := l_B +1;
C = D;
end while;
In order to find the set C' we use an efficient burning algorithm to determine in a single step all clusters belonging to C'.

9.3.3 Simulated Annealing(SA)

The MA described above is an efficient method to estimate the fractal dimension, and the base for Simulated Annealing algorithm. SA is a class of algorithms inspired by the annealing process in metallurgy. In the SA context, a box partition (box covering) is the state S of the physical system and the number of boxes NB is the "internal energy" in that state. In order to consider a neighbor state S' of the current state S we compute three fundamental operations:

- movement of nodes;
- creation of new clusters;
- union of clusters;

If S' is a solution worse that S, there is a probability to accept the state S' even if it has the energy E(S') ≥ E(S). A new state or partition with boxes of size l_B is obtained from the current state by moving nodes and merging clusters. Let A and B be two generic clusters of the current partition. We define the following operations:

- movement: a node is moved from A to B if B diameter doesń t exceed lB, and A includes at least two nodes;
- creation: a node is taken from cluster A to form a new cluster;
- merge: all clusters are merged by using the merge algorithm described above.

At each "temperatur" we perform k1 movements and k2 creations of nodes, and a single merge of all clusters by using MA. We always accept a better or equal solution, while we accept a solution S' worse than S with probability:

$$p = e^{-\frac{E(S')-E(S)}{T}} \tag{9.3}$$

At each step the system is cooled down to a lower temperature T_0 = cT , where c < 1 is the cooling constant. The typical starting temperature T is about 0.6 and the typical values of k1 and k2 are 5000 and 5, respectively. We performed about 5000 steps at each temperature, and then reduceed T. The number of outer cycles (temperature reductions) is k3, and it is set to 20, with cooling constant c set to 0.995.

9.4 Results

We implemented the three algorithms and compared their performance in terms of speed and quality of the result. In fact, being the box partitioning problem NP-complete, on large networks its exact solution is not feasible. Consequently, it is not enough to have a fast algorithm to compute the box partitioning, but the results must be trusted, in the sense that the partitioning found should be close enough to the global minimum to guarantee the consistency of the results. We tested the goodness of the results by repeatedly running the same algorithm, selecting randomly the initial configuration. We then checked the variance of the resulting estimate of $N_B(l_B)$ for various values of l_B, which in turn depends on the number of boxes found in each partitioning. We used for the tests the software network related to Java JDK 1.5 system, which includes the standard Java libraries and development tools. The JDK network has 8499 nodes and 42048 edges, so it can be considered a quite large network.

9.4.1 Execution speed

We computed the execution speed on the whole computation of d_B, which is what actually matters, running the three algorithms starting from random configurations of the initial box partitioning and performing 100 times the computation. The results for a PC with Windows XP and a processor Intel Core 1.4 GHz are reported in Tables 9.1 and 9.2.

The most efficient algorithm is MA, and this is confirmed also by other test runs on other networks, not reported here for the sake of brevity. GC is still very efficient, while SA is much worse as regards execution speed, being at least one order of magnitude slower. It has to be pointed out that these results may

Table 9.1: Average execution times for d_B computation on JDK 1.5 class graph (8499 nodes and 42048 links).

Algorithm	Time (s)	d_B
GC	410	3.96
MA	289	4.24
SA	8807	4.06

Table 9.2: Average execution times for d_B computation on the E. Coli protein interaction network graph (2859 nodes and 6890 links).

Algorithm	Time (s)	d_B
GC	13	3.44
MA	21	3.57
SA	1177	3.47

depend not only from the system analyzed, but also from the particular release. In fact, changes of the software structure among various releases are reflected in changes of their fractal dimension. Depending on the system, these changes may be more or less pronounced.

In fig. 9.6 we report the results of the fractal dimension for various versions of different software systems. It can be noted that, for Glass Fish, Eclipse Birt, Netbeans, the fractal dimension is quite stable among releases, while for JDK and Eclipse, it shows major variations. This is an index of major topological changes in the structure of the graph associated to these systems, and, consequently, of major modifications applied onto the software. Regarding the quality of results, they look similar but not exactly the same. This is discussed in detail in the next section.

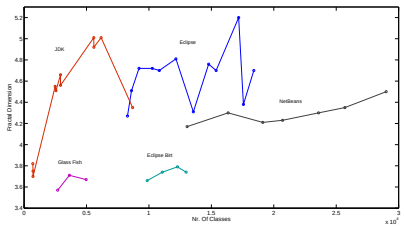


Figure 9.6: Fig. 9.6. Fractal dimension for different versions of the analyzed systems, as a function of the release version, according to the number of classes of each version.

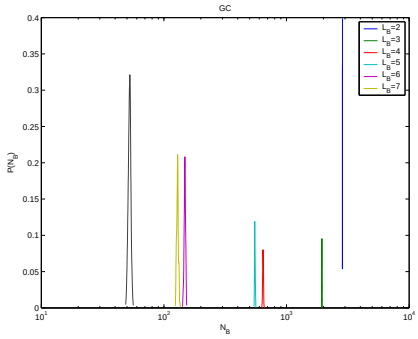


Figure 9.7: Fig. 9.7 Empirical distributions of the values of N_B for six values of l_B, for GC algorithm run 1000 times.

9.4.2 Result Quality

We computed the reliability of the three tested algorithms by testing for their repeatability in 1000 runs on a smaller network than the whole JDK 1.5

software graph, the E. Coli protein interaction network [70]. This network has 2859 nodes and 6890 edges. We varied l_B, from 2 to 7. Figs. 9.7, 9.8 and 9.9 show the empirical distributions of the values of N_B for each value of l_B, and for GC, MA and SA algorithms, respectively.

As you can see, GC and SA algorithms show a very small dispersion of the resulting values of NB, showing that both are highly reliable. On the other hand, the results of Fig. 9.8 regarding MA algorithm show a much higher dispersion. Consequently, despite its high performances, we deem that MS algorithm is not suitable for the computation of software networks fractal dimension. We report in Fig. 9.9 the standard deviation of the computed N_B for the three algorithms, for eight values of l_B, from 2 to 9. Fig. 9.10 confirms the previous results on the reliability of the three algorithms.

The standard deviation of MA results is consistently higher than that of GC and SA. The latter algorithms are quite similar, with a slightly better average performance of SA over GC on the eight test values of l_B.

9.5 Conclusion

The fractal dimension of software networks has the potential to be a significant, synthetic metric describing the regularity of the structure of a software system, and moreover it has been proven to be correlated to source code quality metrics of OO systems. It is therefore important to have efficient and reliable

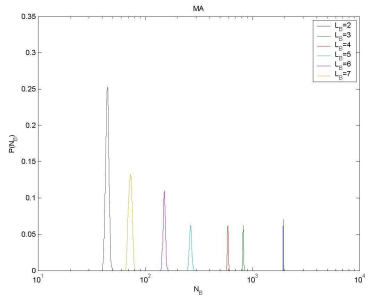


Figure 9.8: Fig. 9.8. Empirical distributions of the values of N_B for six values of l_B, for MA algorithm run 1000 times.

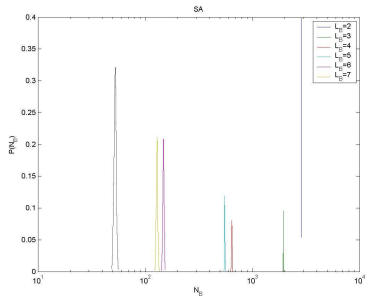


Figure 9.9: Fig. 9.9. Empirical distributions of N_B for six values of l_B, for SA algorithm run 50 times.

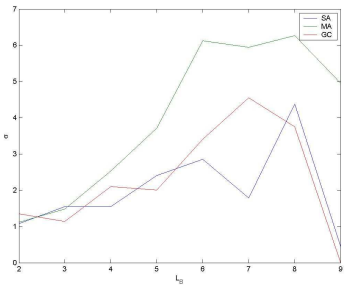


Figure 9.10: Fig. 9.10. Standard deviations of the values of N_B for eight values of l_B, for MA algorithm run 1000 times.

algorithms to compute it. We presented three different algorithms to compute the fractal dimension of networks, which to our knowledge cover all the approaches proposed in literature. These algorithms - Greedy Coloring, Merge Algorithm, and Simulated Annealing, have been described and compared using the software network related to Java JDK 1.5 open source system and, for the purpose of assessing the algorithm reliability, also using a smaller protein interaction network. We found that SA is the best algorithm in terms of precision, but it is by far the worst in terms of speed. The time performance of MA is better than GC for large networks but the greedy coloring produces more precise solutions. In conclusion, the Greedy Coloring algorithm, based on the equivalence of the box counting problem with the graph coloring problem, looks the best compromise, having speed comparable to MA, and accuracy comparable with SA.

Chapter 10

Fractal Dimension Metric and Object-Oriented Software Quality

In this chapter We present a study were software systems are considered as complex networks which have a self-similar structure under a length-scale transformation. On such complex software networks we computed a self-similar coefficient, also known as fractal dimension, using “the box counting method ”.

We analyzed various releases of the publically available Eclipse software systems, calculating the fractal dimension for twenty sub-projects, randomly chosen, for every release, as well as for each release as a whole. Our results display an overall consistency among the sub-projects and among all the analyzed releases.

We found a very good correlation between the fractal dimension and the number of bugs for Eclipse and for twenty sub-projects. Since the fractal dimension is just a scalar number that characterizes a whole system, while complexity and quality metrics are in general computed on every system module, this result suggests that the fractal dimension could be considered as a global quality metric for large software systems. Our results need however to be confirmed for other large software systems.

In this part, we focus on analyzing the fractal dimension of software networks. A significant correlation was found between the fractal dimension computed in various software systems and some CK metrics related with software quality [22], [25]. We investigated the correlation between the fractal dimension and the number of bugs of different releases of the Eclipse software system, analyzing the releases as a whole system as well as computing the fractal dimension of many system of its sub-projects. Our results show a very high correlation among the fractal dimension and the presence of bugs. For this reason, we believe that the fractal dimension, being a scalar number that characterizes the whole system, could be a synthetic metric describing the complexity and

the quality of a software system.

10.1 Research Questions

The motivation for this study rises from the many recent results in literature showing that large software systems can be naturally associated to software networks which in turn display complex properties. The finding that the concept of fractal dimension may be computed also for a complex graph, suggests that a single global metric may be representative of the complexity of the whole system. With regard to software systems, it is well known that the higher the system complexity, the harder is its maintainability, and the easier is to introduce defects into it. These are all features which contribute to decrease software quality, and which are generally associated to a larger presence of bugs, which is a reliable quality metric. It is thus natural to ask if the fractal dimension is somehow related to the bugs affecting the system, and to which extent. Note that the fractal dimension may be not necessarily well defined for a software network. In fact not all networks posses a structure for which the number of boxes covering the network scales with the linear size following a power law, as required by the definition of fractal dimension. This may be particularly tricky in the sub-systems, where the smaller size may prevent the power-law scaling to occur over a range of the linear size large enough. In order to make clear the previous consideration, we present and discuss the following research questions:

RQ1: Is the fractal dimension well defined for the Eclipse software network and for the sub-projects?

RQ2: Is the fractal dimension correlated to the presence of bugs in the sub-projects of a same release?

RQ3: Are the fractal dimension and bugs globally correlated in time, as the Eclipse systems evolves from one release to another?

RQ4: Is the fractal dimension a reliable global metric for software quality?

We studied the class graphs of Eclipse [3] and of twenty of its sub-projects, whose source code is freely available on the Internet. Using the same method followed by Song et al. [70], we calculated the fractal dimension d_B for each of the considered graphs. All these graphs revealed a self-similar structure. In fig. 10.1 we report the box counting analysis of the software network related to Eclipse 3.2. Similar plots are observed also for the other versions of Eclipse analyzed, and for the Eclipse sub-projects the power law behavior is patent.

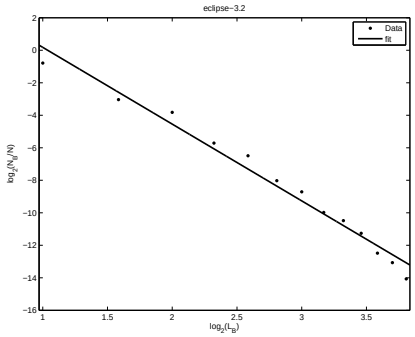


Figure 10.1: *Log-log plot of N_B vs. l_B for Eclipse 3.2.*

10.2 Evolution of the fractal dimension

Software systems evolve in time, as new features are added, and bugs are fixed. The corresponding graphs also evolve, both in terms of number of nodes (classes) and links (relationship between classes). For instance, in the time span considered, Eclipse evolved from about 8500 nodes (Eclipse 2.1) up to 17800 (Eclipse 3.3). The evolution of some software systems may be easily studied because programmers use version control systems for the source code, as for instance CVS (Concurrent Version System) [2] and SVN (Subversion) [6]. In our study, we analyzed five versions of Eclipse and of twenty of its sub-projects. As a consequence the fractal dimensions also change in time, since they reflect the topology of the software network. But these changes may not be completely arbitrary, since each software release preserves a bulk of the previous release, and is built upon it. This means that the fractal dimensions of the various releases are not independent on each other. This may be particularly true in the case of sub-projects, where the modifications may be very limited, or even null, among two consecutive releases. For the entire software system instead it usually happens that new sub-projects are added in a subsequent release, changing even radically the network topology, and the value of the fractal dimension may thus fluctuate more. It must be noted that the fractal dimension does not necessarily increase as the system evolves, as does its size instead, since it reflects the complexity of the network topology, that is in principle uncorrelated with system size.

Table 10.1: *Fractal dimension coefficients for 20 sub-projects of some Eclipse versions.*

Sub-Project	2.1	3.0	3.1	3.2	3.3
compare	2.376	2.935	2.473	3.204	2.492
core.resources	3.057	3.261	3.485	3.528	3.489
core.runtime	2.733	3.428	2.994	2.029	2.075
debug.core	2.275	2.921	2.750	2.873	2.209
debug.ui	2.380	3.204	3.407	4.307	2.779
help	2.325	3.160	2.623	3.522	2.812
jdt.core	3.556	3.628	3.622	4.214	3.812
jdt.debug	2.810	3.598	2.597	3.185	2.745
jdt.junit	2.286	3.234	2.582	3.252	3.010
jdt.launching	2.430	3.031	2.588	3.021	1.899
jdt.ui	3.649	3.857	3.807	4.021	3.649
jface	2.437	3.242	2.525	3.257	2.604
pde.core	2.876	3.466	3.343	3.866	3.408
pde.ui	2.765	3.500	2.988	3.920	3.512
swt	2.865	3.842	3.672	4.200	4.245
team.core	2.052	3.149	2.888	3.310	3.007
team.ui	2.371	3.196	2.478	3.517	3.005
ui.editors	1.555	2.746	2.564	2.910	2.500
ui.workbench	3.093	3.526	3.097	3.640	3.296
update.ui	2.905	2.991	3.031	3.060	2.765

10.3 Results

We analyzed five versions of Eclipse System and twenty sub-projects of the same versions. All the chosen sub-projects have at least 40 classes, to provide enough statistics for bugs and for computing the fractal dimension. Our analysis presents two points of view: the evolution in time of the fractal dimension as Eclipse, as well as its sub-projects, are developed, and an analysis of the fractal dimension at fixed time, comparing the fractal dimensions of the sub-projects of a same release. The correlation of fractal dimension with bugs is studied in both cases.

First of all, we found a range for the scaling of the number of boxes covering the networks with the linear size large enough to provide a well defined value of the fractal dimension for all the releases and sub-projects. This is a nice result in itself, since in principle the scaling may be not well defined for arbitrary networks. It must be pointed out that there is no a single choice for building the software graph from the software code, since different relationships may be chosen to represent the links among nodes. In particular many of these relationships should preferably be represented in a directed graph.

A first empirical result is that the behavior of the fractal dimension with time evolution is oscillatory, namely it fluctuates from one release to the next, not showing regular behavior. In particular, it does not grow as the system size does, as the releases evolve with software development, as reported in Tab. 10.2. The

same kind of oscillations are shown by the fractal dimension of the sub-systems (Tab. 10.1), where for most of them the fluctuations exactly replicate those presented in the time evolution for the whole releases, even if the fractal dimensions of the sub-systems are smaller than those of the whole releases. This latter fact may suggest that the assembly of the sub-systems software networks provide a software network which is more complex than each single component, being the fractal dimension a measure of complexity. In any case the fluctuations presented by the fractal dimension of the entire releases are coherent with those existing in most of the sub-systems.

The behavior of the fractal dimension in time for the whole releases is reflected by the bugs number (Tab. 10.2). In fact the bugs number oscillates from the first to the last release with the same pattern. This implies the existence of a high correlation among bugs and fractal dimension for the whole releases, as reported in Tab. 10.2, where the correlation is above 0.87. We found a high correletion also between fractal dimension and some traditional metrics [19] that have been found to be correlated with fault proneness of software systems (see Tab.10.2). Since CK metrics analyzed (CBO and RFC) refer to single classes, while the fractal dimension is a global measure of the system, we computed the mean value of each metric for each version of the analized systems.

The high correlation of fractal dimension with bugs is also shown across the sub-systems of a same release, as reported as an example in Tab. 10.3 for Eclipse 3.2. In this case there is no a-priori correlation among the different subsystems, since they do not share any parts of code, as it occurs in the case of the time evolution, where a subsequent release contains part of the code of the previous one. Thus this empirical result strongly point to the direction that the fractal dimension may by a reliable indicator for the system quality. We may now answer to the research questions risen in section 2.

RQ1: Is the fractal dimension well defined for the Eclipse software network and for the sub-projects?

The answer is positive. We found the scaling regions necessary for defining the fractal dimension for all the releases and for all the sub-systems with more than 40 classes. The results also show a high degree of coherence among the fractal dimensions of the sub-projects in a release and the fractal dimension of the entire release, as shown in Tabs. 10.1, 10.2.

RQ2: Is the fractal dimension correlated to the presence of bugs in the subprojects of a same release?

The fractal dimension is well correlated with the number of bugs affecting the sub-systems in all of the releases analyzed, as reported in Tab. 10.4. All the correlations are systematically around 0.7 and all the p-values show an extremely high level of significance.

RQ3: Are the fractal dimension and bugs globally correlated in time, as the

Eclipse systems evolves from one release to another?

Bugs number and fractal dimension display the same kind of oscillations with the evolution of the system releases. The correlation reported in Tab. 10.2 is particularly high, with a p-value of about 0.05, which may be considered a good level of significance, given that we had only five measurements.

RQ4: Is the fractal dimension a reliable global metric for software quality?

All the results point into this direction, since we found for the system evolution in time the same oscillations for fractal dimension and for bugs number, and a very high level of significance for the correlation among fractal dimension and bugs for the twenty sub-systems in all the releases analyzed.

Table 10.2: *Correlation coefficients between some metrics and the fractal dimension for all the considered versions of Eclipse.*

Version	Classes	Locs	CBO	RFC	Bug	FD
Eclipse2.1	8546	779130	5.201	17.21	7023	4.347
Eclipse3.0	12254	1118453	7.112	21.138	16986	4.652
Eclipse3.1	14235	1351957	5.300	17.375	13836	4.505
Eclipse3.2	17165	1638699	7.489	21.206	15481	4.722
Eclipse3.3	17881	1657986	5.205	16.931	10451	4.546
Corr.	0.658	0.661	0.860	0.827	0.873	1
P-Value	0.227	0.224	0.061	0.084	0.053	–

Table 10.3: *Correlation coefficients between bugs and fractal dimension for 20 sub-projects of Eclipse 3.2.*

Sub-Projects	Nr. Classes	Total Locs	Bugs	DF
compare	156	15722	292	3.2040
core.resources	277	26170	188	3.5281
core.runtime	29	2946	37	2.0290
debug.core	160	9371	97	2.8735
debug.ui	742	53507	1039	4.3070
help	1210	234333	1761	4.2138
jdt.core	481	36394	257	3.1850
jdt.debug	112	10877	112	3.0212
jdt.junit	2781	288276	1987	4.0213
jdt.launching	154	12282	111	3.2521
jdt.ui	379	34603	338	3.2572
jface	356	27271	59	3.5217
pde.core	434	29536	307	3.8662
pde.ui	991	79675	652	3.9196
swt	801	106224	1060	4.1996
team.core	202	13064	167	3.3103
team.ui	364	30316	647	3.5166
ui.editors	180	13822	37	2.9100
ui.workbench	1518	128865	971	3.6398
update.ui	134	11255	60	3.0600
correlation	0.6095	0.6217	0.7358	1
P-Value<0.05	0.0043	0.0034	0.0002	—-

Table 10.4: *Correlation coefficients and p-value between bugs and fractal dimension for 20 sub-projects of Eclipse.*

Version	bug-df	P-Value
Eclipse2.1	0.728	0.000277
Eclipse3.0	0.700	0.000582
Eclipse3.1	0.737	0.000207
Eclipse3.2	0.736	0.000217
Eclipse3.3	0.645	0.002152

10.4 Threats to Validity

The first threat to validity is that we considered just one software system, Eclipse. Though it is a very large system, composed of many sub-projects developed by different programmers, and though we studied several versions of it, Eclipse clearly cannot be representative of all Java systems. Further studies on different systems, open source and commercial are needed to further validate our study.

A second threat, related to the previous one, is that Eclipse may be considered representative of Java systems, but not of systems written not in Java code. Thus a full investigation on the possibility of using the fractal dimension as a single metric able describe the complexity of the whole software system and of its capability to distinguish high quality software systems, due to the correlation with bugs, must include a comprehensive study, spanning several languages and systems.

10.5 Conclusion

We presented a novel empirical analysis where a single measure of complexity, the fractal dimension, is applied to the concept of software network and to its sub-networks, in order to introduce a global quality metric for the whole software system. For this purpose we showed how the fractal dimension is related to the bugs affecting the whole system, which are an ubiquitous measure of system quality, both looking at the evolution in time across different releases as the system evolves, and at fixed time, across the sub-projects of a same release. Our results show strong and meaningful correlations among fractal dimension and bugs affecting the software, to the point that for the entire software systems they display a same oscillatory behavior. These results are limited to five releases of the same system, Eclipse, and thus may lack of generality. Further work is needed in order to extend the validity of our results to other systems, in particular to different programming languages or software paradigms. This will constitute the object of our future work.

Chapter 11

Concluding remarks

During these three years of Ph.D. I focused my research on the problem of measuring software quality using new approaches. I used use the concept of complex networks to study software networks, introducing new metrics which are promising for assessing software quality.

In particular I investigated how the presence of fat-tail distributions in software properties can be used to understand the process of bug introduction into software, how it can be used to forecast the evolution of software systems, and how it can be related to software quality.

I also investigated such properties in software systems under refactoring, analyzing how the presence of power-laws can be connected to specific types of refactorings and to the way they contribute to improve software quality. These studies can be found in the papers listed at the end of this thesis (see the *List of publications*).

The main results obtained during these three years can be resumed as follows:

I analyzed the bug distribution in modules of various versions of the Java Eclipse system, by comparing the goodness of fit of log-normal, Double Pareto, Yule-Simon and Weibull distributions and by using an empirical analysis. I showed that new bugs are introduced, on average, in larger amounts in modules which where more bug affected in previous releases. This explicitly supports the"preferential attachment mechanism" at the basis of the Yule-Simon model, not excluding mechanisms underlying the log-normal and the Double Pareto models. The last two models are, however, less useful because they cannot account for zero values, namely modules with no bugs, and may be only useful to model a reduced part of the entire software system.

I analyzed the distribution of SNA metrics in OO software networks, comparing their properties with those of CK metrics and other graph-related metrics, to study the impact of such metrics on Bugs. The empirical distributions of

all the studied metrics systematically present power-laws in their tails, a property which holds also for bug distribution. We found analytical distribution functions suitable for fitting the empirical data. Power-law always outperforms other fittings in the tails, whereas Yule-Simon distribution follows the shapes of most metrics empirical distributions very well. In particular, Ties and Fan-in metrics are fitted by Yule-Simon distribution from the very beginning of values, the determination coefficients being over 0.98. We have shown - using the metric Ties - how it is possible to provide reliable estimates for averages and extreme values of subsequent releases from the knowledge of the best fitting parameters and system size. The knowledge of extreme values of metrics could be exploited to keep under control the quality of software systems, because in general high values of these metrics denote high coupling among classes. I also shown how the correlations among SNA metrics and Bugs are generally good. I also found that some structural metrics look negatively correlated with bugs, which may provide alternative insights to practitioners, with respect to the more popular and largely used metrics positively correlated to bugs. Among the contributions of this analysis there is the finding that some SNA metrics can be useful as software quality indicators and that the role and the position of the software nodes are related to bug proneness.

Finally I presented a novel empirical analysis where a single measure of complexity, the fractal dimension, is applied to the concept of software network and to its sub-networks, in order to introduce a global quality metric for the whole software system. For this purpose I shown how the fractal dimension is related to the bugs affecting the whole system, which are an ubiquitous measure of system quality, both looking at the evolution in time across different releases as the system evolves, and at fixed time, across the sub-projects of a same release. The results show strong and meaningful correlations among fractal dimension and bugs affecting the software, to the point that for the entire software systems they display a same oscillatory behavior. These results are however limited to five releases of the same system, Eclipse, and thus may lack of generality. A more recent study, not reported in the thesis, shows that these results can be extended also to various versions of Netbeans and to its sub-projects, suggesting that the fractal dimension can be used as global index for the software quality for large Java systems.

Bibliography

[1] Apache Ant. http://www.apache.org/. [cited at p. 32]

[2] Cvs. http://www.nongnu.org/cvs/. [cited at p. 32, 66, 121]

[3] Eclipse. http://www.eclipse.org/. [cited at p. 31, 65, 89, 120]

[4] NetBeans. http://www.netbeans.org/. [cited at p. 32, 65, 89]

[5] Jdk. http://www.java.sun.com/. [cited at p. 32]

[6] Svn. http://subversion.tigris.org/. [cited at p. 32, 121]

[7] L. A. Adamic, “Zipf, power-law, pareto - a ranking tutorial", Information Dynamics Lab, HP Labs, HP Labs, Palo Alto, CA 94304, Tech. Rep., October 2000. [Online]. Available: http://www.hpl.hp.com/research/idl/papers/ranking/ [cited at p. 54]

[8] Andersson and P. Runeson. A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. IEEE Trans. Software Eng. 33 (2007) 273-286. [cited at p. 19, 51, 52, 66]

[9] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON ’07, pages 215–228, New York, NY, USA, 2007. ACM. [cited at p. 70]

[10] A. Barabasi and R. Albert. Emergence of scaling in random networks. Science 286 (1999) 509-512. [cited at p. 1, 6, 18]

[11] A. Barabasi, R. Albert, and H. Jeong. Scale-free characteristics of random networks: the topology of the world wide web. *Phys. A*, 281:69–77, 2000. [cited at p. 1, 6]

[12] V.R. Basili, L.C. Briand, and W.L. Melo, A Validation of Object Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering, vol. 22(10), pp.267- 271, 1996. [cited at p. 11, 13, 82]

[13] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero Understanding the shape of Java software. In Proc. of the 21st ACM SIGPLAN conference Object-oriented programming languages, systems, and applications (OOPSLA), Oct. 2006, Portland, USA. [cited at p. 6, 21, 52]

[14] F. Brito e Abreu The MOOD Metrics Set In Proc. ECOOP'95 Workshop on Metrics (1995) [cited at p. 11]

[15] Bugzilla. http://www.bugzilla.org/ [cited at p. 66]

[16] S.R. Chidamber and C.F. Kemerer Towards a Metrics Suite for Object Oriented Design Proc. Conf. Object Oriented Programming Systems, Languages, and Applications (OOPSLA 91), vol. 26, no. 11, pp. 197-211, 1991. [cited at p. 13]

[17] H. Bauke. Parameter estimation for power-law tail distributions by maximum likelihood methods. European Physical Journal B 44 (2007) 167-173. [cited at p. 31]

[18] G.J. Baxter and M.R. Frean Software graphs and programmer awareness. arXiv:0802.2306v1 (2008) [cited at p. 19]

[19] S.R. Chidamber and C.F. Kemerer A metric suite for object-oriented design IEEE Trans. Software Eng., 20 (1994) 476-493. [cited at p. 1, 11, 25, 46, 47, 106, 123]

[20] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis. IEEE Trans. Software Eng., 24 (1998) 629-639. [cited at p. 11, 13]

[21] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. E-print (2007) arXiv:0706.1062. [cited at p. 31]

[22] G. Concas, M. Marchesi, S. Pinna, and N. Serra. On the suitability of yule process to stochastically model some properties of object-oriented systems. Physica A 370 (2006) 817-831. [cited at p. 7, 14, 15, 19, 21, 25, 29, 52, 66, 106, 109, 110, 119]

[23] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large object-oriented software system. IEEE Trans. Software Eng. 33 (2007) 687-708. [cited at p. 5, 6, 9, 19, 20, 25, 29, 47, 51, 66, 105]

[24] G. Concas, M. Locci, M. Marchesi, R. Tonelli, and I. Turnu. Computing the fractal dimension - a global metrics for large software systems. In *Proc. of 14th International Conference on Computational Intelligence and Software Engineering (CISE), Wuhan, China, IEEE Press*, pages 1–4, 2010. [cited at p. 11, 15]

[25] G. Concas, M. Marchesi, A. Murgia, S. Pinna, and R. Tonelli. Assessing traditional and new metrics for object-oriented systems. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, WETSoM '10, pages 24–31, New York, NY, USA, 2010. ACM. [cited at p. 119]

[26] A. de Moura, Y. Lai, and A. Motter. Signatures of small-world and scale-free properties in large computer programs. Physical Review E 68 (2003) 017102. [cited at p. 18]

[27] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In CASCON, Toronto, CA, Oct 23-25 2007. [cited at p. 70]

[28] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan and A. V. Aho, Do Crosscutting Concerns Cause Defects?. IEEE Trans. Software Eng. 26 (2000) 786-796. [cited at p. 69, 70]

[29] D. de Solla Price. A general theory of bibliometric and other cumulative advantage processes. J. Amer. Soc. Inform. Sci. 27 (1976) 292-306. [cited at p. 18]

[30] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *Computer Communications Rev.*, 29:251–264, 1999. [cited at p. 2]

[31] J. Feder. Fractals. *Plenum press, New York*, 1988. [cited at p. 15]

[32] N. Fenton "Software Measurements, a Necessary Scientific Basis" IEEE Trans. on Softw. Eng. Vol. 20 n. 3, (1994). [cited at p. 1]

[33] N. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Trans. Software Eng. 26 (2000) 797-814. [cited at p. 19, 51, 52, 53]

[34] S. Focardi, M. Marchesi, and G. Succi. A stochastic model of software maintenance and its implications on extreme programming processes, In: G. Succi, M. Marchesi (Eds.). Extreme Programming Examined, The XP Series, Addison-Wesley, 2000, pp. 191-206. [cited at p. 5, 6]

[35] N. Ganesh, K. Gopinath and V. Sridhar, "Structure and Interpretation of Computer Programs". cs.SE arXiv:0803.4025v1

[36] C. P. Stark and N. Hovius, "The characterization of landslide size distributions", Geophysical Res. Lett., Vol. 28, No. 6, pp. 1091-1094, March 15, 2001. [cited at p. 21]

[37] T. Gymothy, R. Ferenc, and I. Siket Empirical validation of object-oriented metrics on open source software for fault prediction IEEE Trans. Software Eng. 31 (2005) 897-910. [cited at p. 11, 13, 82]

[38] M. L. Goldstein, S. A. Morris, and G. G. Yen. Problems with fitting to the power-law distribution. The European Physical Journal B - Condensed Matter and Complex Systems 41 (2004) 255-258. [cited at p. 31]

[39] A. Gorshenev and Y. Pis'mak. Punctuated equilibrium in software evolution. Physical Review E 70 (2004). [cited at p. 18]

[40] B. M. Hill. A simple general approach to inference about the tail of a distribution. The Annals of Statistics 3 (1975) 1163-1174. [cited at p. 30]

[41] M. Ichii, M. Matsushita, K. Inoue. An Exploration of Power-Law in Use-Relation of Java Software Systems. Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on (2008) 422-431. [cited at p. 19]

[42] C. Kai-Yuan, Y. Bei-Bei. Software execution processes as an evolving complex network. Information Sciences 179 (2009) 1903-1928. [cited at p. 18]

[43] http://netbeans.org/bugzilla/report.cgi [cited at p. 66]

[44] L. Hatton, "Power-law distributions of component size in general software system". IEEE Transactions on Software Engineering, 35(4) IEEE, July, pp. 566-572 (2009). [cited at p. 52]

[45] P. Liggesmeyer, G Engels, J Munch, J. Dorr, N. Riegel Proc. of Software Engineering 2009 Kaiserslautern, Germany, 2-6 March 2009, pp. 151-162. [cited at p. 51]

[46] W. Li. The big bang graph-a colored graph representation of software design. ACM Computer Science Research Trends (Ed. Casey B. Yarnall), Nova Science Publishers, Book Chapter (2008). [cited at p. 5]

[47] M. Lorenz, and I. Kidd Object Oriented software metrics: A pratical guide Endge wood cliffs, N.J.: Pretience Hall [cited at p. 11, 88]

[48] Louridas, P., Spinellis, D., and Vlachos, V. Power laws in software. ACM Trans. Softw. Engin. Method. 18, 1, Article 2 (September 2008), 26 pages. [cited at p. 6, 7, 19, 20, 21, 45, 51, 105]

[49] M. Marchesi, S. Pinna, N. Serra, S. Tuveri. Power laws in smalltalk. In: ESUG. Conference 2004 Research Track (2004). [cited at p. 18]

[50] D.W. Matula, G. Marble and J.D. Isaacson, Graph Coloring Algorithms. In Graph Theoryand Computing (Ed. R. Read). New York: Academic Press, pp. 109-122, 1972. [cited at p. 110]

[51] T. McCabe A Complexity Measure. IEEE Transactions on Software Engineering, Vol.2, No.4, 308-320. 1976 [cited at p. 13]

[52] S. Milgram. The small world problem. *Psych. Today*, 2:60–67, 1967. [cited at p. 6]

[53] E. Mills SEI Curriculum Module SEI-CM-12-1.1 Software Engineering Institute - Technical Report [cited at p. 1]

[54] M. Mitzenmacher. Dynamic Model for File Size and Double-Pareto Distribution. Internet Mathematics 3 (2003) 305-333. [cited at p. 7, 21, 22, 55]

[55] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. Internet Mathematics 1 (2004) 226-251. [cited at p. 17, 58]

[56] C. Myers. Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. Physical Review E 68 (2003). [cited at p. 2, 6, 9, 18, 105]

[57] M. Newman. Power laws, pareto distributions and zipf's law. contemporary Physics 46 (2005) 323-351. [cited at p. 13, 17, 20, 21, 23, 27, 45, 52, 65, 74, 85]

[58] T.J. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA '02), 27, 2002, pp. 55-65. [cited at p. 19]

[59] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the Location and Number of Faults in Large Software Systems. IEEE Trans.Software Eng. 31 (2005) 340-355. [cited at p. 19]

[60] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in object-oriented programs. Communications of the ACM 48 (2005) 99-103. [cited at p. 2, 5, 6, 18]

[61] Qualitas Research Group, Qualitas Corpus Version 20090202, http://www.cs.auckland.ac.nz/ewan/corpus. The University of Auckland, February 2009. [cited at p. 78, 89]

[62] W. J. Reed, "The Pareto Law of Incomes - An Explanation and an Extensio", Physica A, Vol. 319, 469-485. 2003. [cited at p. 22]

[63] W. J. Reed and B. D. Hughes, "From Gene Families and Genera to Incomes and Internet File Sizes: Why Power-Laws Are So Common in Natur", Physical Review E, vol. 66 (2002), 67-103. [cited at p. 21]

[64] W. J. Reed and M. Jorgensen, "The Double Pareto - Lognormal Distribution - A New Parametric Model for Size Distributions", Communications in Statistics: Theory and Methods, 33-8. 2004. [cited at p. 21]

[65] J.P. Scott Social Network Analysis Sociology, Vol. 22, No. 1, 109-127 (1988) [cited at p. 5]

[66] H. Seal. The maximum likelihood fitting of the discrete pareto law. Journal of the Institute of Actuaries 78 (1952) 115-121. [cited at p. 30]

[67] H. Simon. On a class of skew distribution functions. Biometrika 42 (1955) 425-440. [cited at p. 18, 22, 66]

[68] N. D. Singpurwalla, Extreme Values from a Lognormal Law With Applications to Air Pollution Problems Technometrics, VOL. 14, No. 3, (1972) [cited at p. 84]

[69] C. Song, L. Gallos, S. Havlin, and H. A. Makse. How to calculate the fractal dimension of a complex network: the box covering algorithm. *Journal of Statistical Mechanics*, P03006, 2007. [cited at p. v, 15, 16, 110]

[70] C. Song, S. Havlin, and H. Makse. Self-similarity of complex networks. *Nature*, 433:392–395, 2005. [cited at p. 2, 6, 9, 14, 15, 106, 109, 115, 120]

[71] R. Subramanyam, and M. Krishan Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects IEEE Trans.Software Eng. 29 (2003) 297-310. [cited at p. 11, 13]

[72] T. Tamai and T. Nakatani. Analysis of software evolution processes using statistical distribution models. Proc. of the International Workshop on Principles of Software Evolution, IWPSE '02, Orlando, Florida, 2002, pp. 120-123. [cited at p. 18]

[73] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton and J. Noble. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010. [cited at p. 89]

[74] A. Tosun, B. Turhan and A. Bener. Validation of Network Measures as Indicators of Defective Modules in Software Systems. Proceedings of the 1st International Conference on Predictor Models (PROMISE), 2009. [cited at p. 13, 14, 88]

[75] I. Turnu, G. Concas, M. Marchesi, S. Pinna, and R. Tonelli. A modified yule process to model the evolution of some object-oriented system properties. *Information Sciences*, 181:883–902, 2011. [cited at p. 9]

[76] S. Valverde, R. Ferrer-Cancho, and R. Solé. Scale-free networks from optimal design. Europhysics Letters 60 (2002) 512-517. [cited at p. 2, 5, 6, 18, 105]

[77] S. Valverde and R. Solé. Hierarchical small worlds in software architecture. Working Paper 03-07-044, Santa Fe Institute, Santa Fe, NM (2003) . [cited at p. 6, 18]

[78] S. Valverde and R. Solé. Network motifs in computational graphs: A case study in software architecture. Phys Rev. E 72 (2005) 026107. [cited at p. 1, 6]

[79] R. Vasa, M. Lumpe, P. Branch and O. Nierstrasz Comparative analysis of evolving software systems using the Gini coefficient icsm, pp.179-188, 2009 IEEE International Conference on Software Maintenance, 2009 [cited at p. 66]

[80] E.J. Weyuker "Evaluating Software Complexity Measures" IEEE Trans. on Softw. Eng., Vol. 14 n. 9, 1988. [cited at p. 1]

[81] R. Wheeldon and S. Counsell. Power law distributions in class relationships. Proc. 3rd IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM03) (2003) pp. 45-57. [cited at p. 6, 19, 41, 52]

[82] W.K. Wiener-Ehrich, J.R. Hamrick and V.F. Rupolo, "Modeling software behavior in terms of a formal life cycle curve: implications for software maintenance", IEEE Trans. Software Eng., vol. 10, no. 4, pp. 376-383, 1984. [cited at p. 22]

[83] K. Yamasaki, K. Matia, D. Fu, S. V. Buldyrev, F. Pammolli, M. Riccaboni, and H. E. Stanley. Preferential Attachment and Growth Dynamics in Complex Phys. Rev. E 74 (3) [cited at p. 28]

[84] G. Yule A mathematical theory of evolution based on the conclusions of dr. j.c. willis Philos. Trans. R. Soc. London B 213 (1925) 21-87. [cited at p. 22, 23]

[85] H. Zhang. On the Distribution of Software Faults. IEEE Trans. on Software Eng. 34 (2008) [cited at p. 19, 22, 51, 52, 54, 56, 66]

[86] H. Zhang and H. B. K. Tan, "An Empirical Study of Class Sizes for Large Java Systems", Proc. of 14th Asia-Pacific Software Engineering Conference (APSEC 2007), Nagoya, Japan, December 2007. IEEE Press, pp. 230-237. [cited at p. 21]

[87] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. Proceedings of the 30th international conference on Software engineering, May 10-18, 2008, Leipzig, Germany. [cited at p. 6, 11, 12, 13, 14, 87, 89]

List of Publications Related to the Thesis

Published papers

Journal papers

- 1) G. Concas, M .Marchesi, A. Murgia, R. Tonelli, and I. Turnu "On the distribution of bugs in the Eclipse system", IEEE TRANS. SW. ENG. (2011)

- 2) G. Concas, M .Marchesi, A. Murgia, R. Tonelli, "An Empirical Study of Social Networks Metrics in Object-Oriented Software", Advances in Software Engineering, Volume 2010, Article ID729826, 20 pages doi:10.1155/2010/729826

- 3) I. Turnu, G. Concas, M. Marchesi, S. Pinna, R. Tonelli "A modified Yule process to model the evolution of some object-oriented system properties", Information Sciences 181 (2011) 883-902 .

- 4) G. Concas, G. Destefanis, M. Marchesi, R. Tonelli "An empirical study of object oriented metrics for assessing the phases of an agile project.", Int'l Journal of Software Engineering and Knowledge Engineering (2012), to appear.

- 5) M. Locci, G. Concas, R. Tonelli, I. Turnu , "Three Algorithms for Analyzing Fractal Software Networks", Wseas Trans. on Information Science and Applications , Issue 3, Volume 7, March 2010.

- 6) R. Tonelli, G. Concas, M. Locci, "Three efficient algorithms for implementing the preferential attachment mechanis in Yule-Simon Stochastic Process", WSEAS Transactions on Information Science and Applications, Issue 2, Volume 7, February 2010.

- 7) A. Murgia, R. Tonelli, G. Concas, M. Marchesi, S. Counsell "Parameter-based refactoring and the relationship with fan-in/fan-out coupling" Journal of Object Technology – Conditionally Accepted.

Conference papers

- 8) G. Destefanis, R. Tonelli, G. Concas, M. Marchesi "An Analysis of Anti-Micro-Patterns Effects on Fault- Proneness in Large Java Systems" SAC'12, March 25-29, 2012, Riva del Garda, Italy.

- 9) R. Tonelli, G. Destefanis "Mixing SNA and Classical Software Metrics for Sub-Projects Analysis. "11th WSEAS International Conference on SOFTWARE ENGINEERING, PARALLEL and DISTRIBUTED SYSTEMS (SEPADS '12), 22-24 Feb. 2012, Cambridge UK.

- 10) A. Murgia, R. Tonelli, M. Marchesi, G. Concas, S. Counsell,"Parameter-based refactoring and the relationship with fan-in/fan-out coupling." 2011 Fourth International Conference on Software Testing, Verification and Validation Workshops, - 25 March 2011, Berlin, Germany.

- 11) A. Murgia, R. Tonelli, M. Marchesi, G. Concas, S. Counsell "An Empirical Study of Refactoring in the Context of FanIn and FanOut Coupling", 2011 18th Working Conference on Reverse Engineering - 17-21 Oct. 2011, Lymerich, Ireland.

- 12) A. Murgia, R. Tonelli, M. Marchesi, G. Concas, S. Counsell, J. McFall and S. Swift "Refactoring and its Relationship with Fan-in and Fan-out: An Empirical Study" 16th European Conference on Software Maintenance and Reengineering, 27-30 March 2012, Szeged, Hungary.

- 13) G. Concas, M. Locci, M. Marchesi, R. Tonelli, I. Turnu: "Computing the Fractal Dimension - a Global Metrics for Large Software Systems", Proceedings of CISE 2010, December 10-12, 2010 Wuhan, China.

- 14) R. Tonelli, Giulio Concas, Michele Marchesi, Alessandro Murgia: "An Analysis of SNA metrics on the Java Qualitas Corpus", Proceedings of ISEC (India Software Engineering Conference, Feb 23-26, 2011, Thiruvananthapuram, India.

- 15) I. Turnu, G. Concas, M. Marchesi, R. Tonelli: "The Fractal Dimension Metric and Its Use to Assess Object-Oriented Software Quality", Proceedings of WETSOM 2011, 24 May 2011- Honolulu, Hawaii, USA.

- 16) A. Murgia, G. Concas, M. Marchesi, R. Tonelli, "A machine learning approach for text categorization of fixing-issue commits on CVS", Proceeding of Empirical Software Engineering and Measurement, September 16, 2010, Bolzano, Italy.

- 17) G. Concas, M. Marchesi, A. Murgia, S. Pinna, R. Tonelli, "Assessing Traditional and New Metrics for Object-Oriented Systems", proceedings of ICSE, International Workshop on Emerging Trends in Software Metrics, (WETSOM) May 2010, Cape Town, South Africa.

- 18) R. Tonelli, G. Concas, M. Locci: "Efficient Implementation of the Yule-Simon Stochastic Process for Modeling Internet and Software Development Activities", proc. Of the 8th WSEAS int. conf. 2009.
- 19) Alessandro Murgia, Roberto Tonelli: "Empirical study on software development and issue growth", proceedings of GIIS 2009, Salerno, Italy.
- 20) A. Murgia, G. Concas, S. Pinna, R. Tonelli, I. Turnu: "Empirical study of software quality evolution in open source projects using agile practices", ETSM 2009, CoRR abs/0905.3287: (2009) Electronic Edition pubzone.org CiteSeerX Google scholar BibTeX.
- 21) A. Murgia, G. Concas, M. Marchesi, R. Tonelli, I. Turnu: "An Analysis of Bug Distribution in Object Oriented Systems", ETSM 2009, CoRR abs/0905.3296: (2009) Electronic Edition pubzone.org CiteSeerX Google scholar BibTeX.

Submitted papers

- 22) I. Turnu , G. Concas, M. Marchesi, R. Tonelli "Fractal Dimension of Software Networks: a Global Quality Metric.", submitted to Information Sciences.