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

Time evolution and distribution analysis of software bugs from a complex network perspective

Alessandro Murgia

Advisor: Prof. Giulio Concas *Co-Advisor*: Prof. Michele Marchesi
Curriculum: ING-INF/05 SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

XXIII Cycle
March 2011

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

Time evolution and distribution analysis of software bugs from a complex network perspective

Alessandro Murgia

Advisor: Prof. Giulio Concas *Co-Advisor*: Prof. Michele Marchesi
Curriculum: ING-INF/05 SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

XXIII Cycle
March 2011

Dedicated to my family

Abstract

Successful software systems are constantly under development. Since they have to be updated when new features are introduced, bug are fixed and the system is kept up to date, they require a continuous maintenance. Among these activities the bug fixing is one of the most relevant, because it is determinant for software quality.
Unfortunately, software houses have limited time and developers to address all these issues before the product delivery. For this reason, an efficient allocation of these resources is required to obtain the quality required by the market.
The keyword for a correct management of software product process is *measure*. As DeMarco states “you cannot control what you cannot measure”, and this thesis is mainly devoted to this aspect. This dissertation bears with software measures related to bug proneness and distribution analysis of software bugs. The aim is to describe the bug occurrence phenomena, identify useful metrics related to software bugs proneness and finally to characterize how bug population is distributed and evolve, discussing also the model able to explain this evolution. Studying the relationship between code evolution and bug distribution or bug-proneness, we foresee which software structure will come out. Thus, this research provides information and guidelines to managers, helping them to plan, schedule activities and allocate resources, during software development.

Acknowledgements

Many people accompanied and supported me during the endeavour of my three years of doctoral studies. It is a pleasure for me have now the opportunity to express my gratitude.
First of all, I would like to thank my supervisors, prof. Giulio Concas and prof. Michele Marchesi, for their encouragement, guidance and timely advices throughout the Ph.D. Special thanks go to Roberto Tonelli, for his professional advices, for helping me in writing papers, for the fruitful discussions and moreover his honest criticism. A special mention goes to Sandro Pinna who introduced me into the AGILE group. A particular remark to prof. Steve Counsell who support me during my researcher period at Brunel University of London, and for reviewing my dissertation giving his feedback.
I feel a deep sense of gratitude for my family for their love, for their unconditional support, and for believing in me —whenever and wherever.
Finally, I am grateful to everybody who accompanied and supported the successful realization of this dissertation, and I express my apology for each one that I could not mention personally one by one.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Purpose

Software systems may become large during their evolution, becoming less maintainable while their complexity rises. Software products have a continuous development process; according to Lehman [59], it occurs because software requires continual change, otherwise it becomes progressively less useful. During this process software can reach millions of lines of code, with lots of different people having taken part on its implementation for months or years.
These continuous changes determine the “software aging phenomenon” [81]. As a consequence the software is modified from people unaware of its original design concepts; for this reason this effect is also defined as “ignorant surgery”. Managers and customers make this problem more acute, pushing to obtain quickly bug[1] fixing and introduction of new features, without giving enough time to developers to fully understand software design. The final results are inconsistent changes, and a “code decay” [38]. Code decays when it is harder to change than it should be, and not because of its essential complexity.
To reduce code decay it is not possible to re-build a system from scratch each time the specification changes, because it is too expensive or sometime not possible. Therefore software systems are generally being maintained and expanded. The maintenance becomes harder and harder with time, because initial design choices have to be updated accordingly to changing environments, both to allow business reorientation, and modernization.

All these reasons clarify why maintenance of software is a resource-consuming activity; even with the increasing automation of software development activities, resources are still scarce. It has been reported that most of the software cost is devoted to evolution and system maintenance of existing source code [16, 43, 57, 62]. Maintainers spend at least 50% of their time trying to understand the program, so that maintenance and evolution are performed correctly [109]. Among maintenance activities, bug fixing is one where companies invest a large amount of money. According to a study conducted by the National Institute of Standards and Technology [99], U.S. industry spends 60 billion dollars a year to address software errors.

[1] The term *bug* is here introduced without effectively definition and can be considered with a broad meaning as synonymous of fault or defect. Section 2.1 will shed light on this mismatch.

Since time and people's availability is limited, software houses cannot address all bugs before the delivery of their products. Thus, to guide the allocation of these resources cost-effectively, a characterization of the bug occurrence phenomenon is required.
Software measurement is necessary to support the decision making process before and during software development life cycle. Among the different metrics that can be adopted as measures of software quality [70], the number of bugs, or better, the bug density are the most employed. The presence of bugs is in fact persistent in human created software; usually bugs are tracked from software inception, and bug density is considered an indicator of bug proneness at class level and then a measure of maintenance effort. Similarly, bug density is reasonably related with the extent of code change, which is again a surrogate measure of maintainability.
From a software engineering point of view it is useful to understand the impact of continuous change and code decay on bugs introduction. For this reason it is worth analysing how this phenomena occours during time evolution of software systems.
Change is a crucial part of a software's life-cycle and successful software products are always under development. Investigating how code evolution affects the bug distribution or the bug-proneness we can in fact foresee software quality. Providing that information and guidelines to managers help them to make decisions, plan and schedule activities and allocate resources for the different software development activities.

While metrics statistics, like mean and variance, can give indications regarding a developed software structure, they are not able to clarify questions about the large-scale structure that will result. For example, given a project release of a fixed dimension, how many modules will exist in the next release? How many bugs will occours in these modules? How many modules can be influences by only one bug? Which will be the maximum number of bugs host by a module?. All previous questions have to be addressed, considering different projects, and the same project during its evolution. In this manner we will be able to understand how large scale software is actually organized, built, and maintained in practise.

The dimension of large scale software systems suggests us to treat them as complex networks [49]. Software system comprises many interacting units and subsystems, at many levels of granularity (functions, classes, interfaces, libraries, source files, packages, etc.), and there are several kinds of interactions or dependencies among these entities that can be used to define graphs to provide a schematic representation of a system. These entities are characterized by features whose distribution in turn is object of different studies [83, 102, 103, 75, 107, 65].
In addition the representation of software as a network provides the possibility for analysing software networks using metrics taken from other disciplines, like Social Network Analysis (SNA) [93]. These SNA metrics can be used together with traditional software metrics, to gain a deeper insight into the properties of software systems.
Software engineering aims to describe how software could or should be written. However to verify if a methodology is working, a good understanding of the shape of existing software is previously required. For this reason the adoption of complex network theory in software engineering provides an useful perspective for the analysis of software properties.
Little is known about complex networks in real software systems, and this is one of the reasons which motivates this study. This thesis gathers data of three years of research on soft-

ware systems belonging to different projects, and focuses its attention on two open source projects: Eclipse and Netbeans.

1.2 Contributions

We present the following research contributions:

1. An empirical study of the statistical properties related to bug occurrence, in Object-Oriented (OO) systems represented as complex network. In detail:

   - we studied the bug distribution of OO systems;
   - we proposed a theoretical model able to justify these distributions

   Previous points are interesting from a software engineering perspective, because there is still no general consensus on which type of distribution these properties have. There is a lack of quantitative and testable predictive theories about the internal structures of large scale systems, or how those systems evolve. Comparing our research with respect to others studies we determine to what extent general conclusions on a distribution can be drawn.

2. An interpretation of the previous analysis from a software engineering point of view. To be precise, we discuss about network properties in terms of software quality. This result is achieved by:

3. a correlation analysis of software network properties with the most important object-oriented software quality indicator such as number of bug.

4. An analysis of the time-evolution of software. Instead of performing a plain study of a final product, we observe how some system properties change along different releases during the software life-cycle. We compare the statistics of previous software properties in different releases and test for the applicability of the same model of growth.

By publishing data of this empirical research that either supports or rejects both conventional wisdom and the results from other researches, more lessons can be learned on software metric distributions, software attribute related to bug proneness and software evolution. Therefore, this dissertation aspire to represents a valid and useful contribute to the body of empirical research on OO software.

1.3 Organization

The remainder of the dissertation starts by introducing an overview of the background material necessary to understand the dissertation itself and evaluate its contributions (Chapter 2). Then Complex Networks are introduced along with their characterization through key distributions of topological entities (Chapter 3). The method of data extraction from software repository and software graph construction is explained in Chapter 4. The case studies considered and collected results (Chapter 5) are presented along with their discussion. The limitations of the approaches are discussed in Chapter 6. This dissertation ends with the conclusions (Chapter 7). The appendix A of this document contains extra data not reported in the result chapter.

Chapter 2

Background

This chapter provides the necessary background to help the reader to understand this dissertation and judge its contributions. We start describing how we, and the literature, define Bug. The second paragraph introduces software metric theory and explains which metrics we have adopted and the reason for this choice.

2.1 Bug, Issue et similia

It is known that software bugs are responsible for different problems, ranging from minor glitches to loss of life or significant material loss[1], but what exactly is a bug?
Defining "Bug" is not trivial, in fact software literature spans a wide variety of terms to describe it and its "synonyms". Terms like Bugs, Issues, Defects, Errors, Faults, Problems, Troubles and Anomalies are often used interchangeably. For example Kajko-Mattson describes "Defect" as a particular class of cause that is related to software [58], whereas Fenton adopts "Defect" for Faults [45]. Bug tracking Systems such as Bugzilla[2] or Trac[3] define Bug as a problem or enhancement request, at the same time the latter definition corresponds to "Anomaly" for IEEE Std. 1044-1993. Such "confusion" is furthermore sharpened by the *modus operandi* of some developers who register minor enhancements as problems to accelerate their implementation and major problems as enhancement requests [70].
There is no standard definition of Bugs or Issues. For this reason, to understand the meaning of our measurements and also to make our experiments repeatable we require a concrete definition of both the terms. This research, like other past work, will define effectively Bug — using upper case — when it considers Bugzilla fields for its identification. On the other hand, we also use the terms bug or issue — in lower case — when the meaning cannot generate confusion or a precise definition is not required. Section 4.1.2 will define precisely how to distinguish Bugs and Issues.

[1] http://www.wired.com/software/coolap
[2] http://www.bugzilla.org/
[3] http://projects.edgewall.com/trac/

2.2 Software Measurement

This study deals with software quality and its relationship with software shape. To achieve better software products, we must be able to understand, control and improve their structures; as DeMarco stated “You cannot control what you cannot measure” [35]. For this reason we must be able to measure software, because measurements is fundamental during each step of software development:

- to support project planning
- to determine the strengths and weaknesses of the current processes and products,
- to evaluate the quality of specific processes and products.
- to assess project progress
- to take corrective actions and evaluate the impact of such actions.

Software metrics are the quantitative measures of these specific characteristics of software development. They are good at detecting outliers, and represent useful means to help software engineers to develop large and complex software systems. Metrics, on the other hand, are not a panacea. There are in fact some aspects of software design and of its quality which are hard to measure. This section describes software measurement practice and how this study fits in this context.

2.2.1 Software Metrics Classification

Since software properties can be analysed from different points of view, the number of potentially usable metrics are virtually infinite. Among the proposed metrics [45, 61, 63, 37] there are some metrics which reveal uncaught information of software, whereas the remaining only re-shape older metrics. In this scenario, it is useful to clarify how metrics can be classified. This will help us understand the context of metrics adopted in our research and moreover may be useful for future research on software analysis.
There is not a single way to classify software metrics [63, 45]. In this dissertation we take under examination the latter one. Fenton divides software entities in three different categories:

- Process: a collection of related software engineering activities.
- Product: an artifact produced as output of any software engineering activity.
- Resource: an artifact required as input of any software engineering activity.

These entities are connected to each other. Process activities use resources and creates the product. An artifact produced by a specific activity can feed another activity, thereby given a specific process an artifact can be considered both a resource or a product. Focusing on a specific process, product or resource for measurement, it is possible to characterize its attributes as:

- Internal: if they can be measured in terms of the sole process, product or resource. Thereby, internal attribute can be measured considering the process, product or resource on its own, not considering its behavior.

- External: if they cannot be measured in terms of the sole process, product or resource. Thereby, external attributes can only be measured considering the process, product or resource related with the external environment and context. In this case the behavior becomes as important as the entity itself.

External attributes are important for managers and customers who are interested in attributes like quality and productivity. However, external attributes are not directly measurable, thus they have to be decomposed into internal attributes which are instead measurable. For this reason, internal attributes play a key role in measuring external attributes. A common principle which characterizes the OO paradigm and design patterns is that good internal structure leads to good external quality. The next paragraph delves into OO metrics adopted for exhibiting the internal structure of a software system.

2.2.2 Object-Oriented Metrics

This section starts with a synthetic description of OO concepts, to achieve the application and the extension of software measurement principles for OO systems. After that the main OO metrics proposed in the software engineering literature will be presented.
OO is an optimal paradigm to successfully apply the *divide et impera* principle for managing the high complexity of large systems. OO systems are composed of "objects" which represent the abstraction of a real objects within an existing application domain.
OO systems are made of interacting objects at runtime, which reside in the computer memory and hold the data to be processed. Objects are characterized by state and behavior; the state represents the internal information structure, whereas the behavior represents the way they can interact with each other. Objects interact by sending messages, and when this occurs they are coupled. Objects are described by classes, the basic OO building components; classes specify how to instantiate objects.
People who create, maintain or improve OO software face problems such as handling software components. This mental burden can be described as cognitive complexity. High cognitive complexity can determine component with undesirable external qualities, like increased "bug-proneness" or reduced maintainability.
Cognitive complexity depends on the internal characteristics of the class and its inter-connections with others. OO metrics try to measure this complexity taking into account three elements:

- Coupling. It measures the static usage dependencies among class in an object-oriented system [19].
- Cohesion. It describes to which extent the methods and attributes of a class belong together [18].
- Inheritance. It evaluates to what extent classes reuse code.

It is worth noting that cognitive complexity has an impact on the external metrics but there are other mechanisms which can play this role as well. Some researches reported that developers exposed to high levels of physical and mental stress are more likely to introduce bugs [50].

2.2.3 Chidamber and Kemerer Metrics

Many OO metrics have been developed by researchers [25, 60, 63, 20]. By far, the most used and validated is the Chidamber and Kemerer (CK) suite [25]. Many studies reveal a correlation between some CK metrics and "bug-proneness" of a class. Others have found that some CK metrics are correlated with the effort required for maintaining classes.
Chidamber and Kemerer were inspired by ontological principles proposed by Bunge [22]. They defined six metrics computed at class level. Referring directly to the original paper, the six metrics are:

WMC - Weighted Method for a class.
Definition: consider a class C, with methods M_1 ,...,M_n that are defined in the class. Let c_1,...,c_n be the complexity of methods. Then:

$$WMC = \sum_{i=1}^{N} c_i \tag{2.1}$$

If all method complexities are considered to be unity, then WMC is the number of methods (NOM) for the class C:

$$WMC = NOM = n \tag{2.2}$$

Interpretation: this metric measures class complexity[4]. The number of methods and their complexity are predictors of the effort required to develop and maintain the class. The larger the number of methods, the greater the potential impact on children classes, as they inherit all methods of parent classes. Moreover, a class with a large number of methods is likely to be application specific, limiting the possibility of reuse.

LCOM - Lack of Cohesion in Methods
Definition: Given a class C, with methods M_1 ,...,M_n , let I_i with i =1, .., n, to be the set of instance variables accessed by M_i. We define the cohesive method set and the non-cohesive method set as:

$$CM = \{(M_i, M_j) | I_i \bigcap I_j \neq \varnothing, i \neq j\} \tag{2.3}$$
$$NCM = \{(M_i, M_j) | I_i \bigcap I_j = \varnothing, i \neq j\} \tag{2.4}$$

the LCOM metric is then given by:

$$LCOM = \begin{cases} |NCM| - |CM|, & \text{if } |NCM| > |CM| \\ 0, & \text{otherwise} \end{cases} \tag{2.5}$$

Interpretation: the LCOM metric refers to the notion of degree of similarity in methods. The degree of similarity is formally given by:

$$\sigma(M_i, M_j) I_i \bigcap = I_j \tag{2.6}$$

[4] Complexity is defined taking into account Bunge's ontology, where it is reported as the cardinality of properties of a substantial individual.

The larger the number of similar methods, the more cohesive the class. This approach is consistent with traditional notion of cohesiveness, which is intended to account for the interrelation of different parts of a program. Cohesiveness of methods in a class is a desirable attribute, since it promotes encapsulation. Lack of cohesion implies a class should probably split into two or more specialized subclasses. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

DIT - Depth of Inheritance Tree
Definition: the DIT metric is the number of ancestors of a given class, that is the number of nodes (classes) to be crossed to reach the root of the inheritance tree. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.
Interpretation: DIT is a measure of how many ancestors may potentially affect a class. The deeper a class is in the hierarchy tree, the greater can be the number of inherited methods making it more difficult to predict its behavior, and thus more effort is required to test this class. Deeper trees constitute greater design complexity, since more methods and classes are involved. On the other hand, the deeper a class is in the inheritance tree, the greater the potential level of reuse of inherited methods.

NOC - Number of Children
Definition: the NOC metric is the number of immediate subclasses subordinated to a class in the class hierarchy.
Interpretation: It is a measure of how many subclasses are able to inherit the methods of the parent class. The greater the number of children, the more the level of reuse. On the other hand, with greater NOC the likelihood of improper abstraction of parent class is greater and may warn of a misuse of inheritance mechanism. The NOC may also give an idea of the potential influence a class has on the design. If a class has a large NOC value, it may require more testing of the methods in that class.

CBO - Coupling Between Object Classes
Definition: the CBO metric for a given class is a count of the number of other classes to which it is coupled.
Interpretation: two classes are coupled when methods declared in one class uses methods or instance variables defined by other class. High coupling negatively affects modularity of the design. Moreover, classes with large number of methods are likely to be more application specific, limiting the possibility of reuse. The number of methods and its complexity is a predictor of how much time and effort is required to develop and maintain the class. In fact the larger the number of couplings, the higher is the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. In order to promote encapsulation and improve modularity, class coupling must be kept to the minimum.

RFC - Response For a Class
Definition: the RFC metric is the cardinality of the response set for a class. Given a class C, with methods M_1 ,...,M_n , let R_i with i = 1, .., n to be the set of methods called by M_i . Thus,

the response set for the class C is defined as:

$$RC = \bigcup_{i=1}^{N} c_i \tag{2.7}$$

and the RFC metric is then given by:

$$RFC = |RC| \tag{2.8}$$

Interpretation: the response set for a class is the number of methods that can be potentially executed in response to a message received by an object of that class. It measures the communication level among classes of the system. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester. For this reason the RFC, influencing testing time, is a indirect measure of the maintenance effort for a class. Increasing the number of methods potentially invoked by a class, the complexity of the system increases as well.

Although the CK metrics have been validated by a large number of studies, the literature presents also criticism mainly on a theoretical basis [27, 56]. Furthermore, the CK suite does not cover potential complexity that arises from certain other OO design factors such as encapsulation and polymorphism.

2.3 Social Network Measurement

This paragraph describes the social network analysis (SNA) and how it can be applied to software systems. The first part introduces the origin of social network analysis and then follows with the reason for its adoption in software engineering. Finally, typical SNA metrics are reported along with their descriptions.

2.3.1 Social Network origin

SNA started with social structure investigation, focusing its attention more on relationships among people than to individual properties. Reporting the definition of Wetherell et al.[106]:

> *Social network analysis (1) conceptualizes social structure as a network with ties connecting members and channeling resources, (2) focuses on the characteristics of ties rather than on the characteristics of the individual members, and (3) views communities as 'personal communities', that is, as networks of individual relations that people foster, maintain, and use in the course of their daily lives.*

Despite its special focus on relationships, SNA provides a rich framework of measures to describe as well the individuals of a network. Due to that, SNA have been used in different areas, and among them in computer science; there are in fact some points of contact between social and software structures.
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 these

units are classes. OO programming prescribes different roles and responsibilities 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 the source code level these interactions are identified by the dependencies among classes, and related to the software architecture.
SNA opens a new interesting perspective for analysing software systems as a network of nodes, or individuals, connected by ties due to relationships among them, representing “channeling resources” among objects. The SNA measures can be used, together with more traditional software metrics, like class LOCs, number of bugs, or the CK suite, to gain a deeper insight into the properties of software systems.
Using CK metrics is it possible to measure the degree of coupling and cohesion of classes in OO software contexts. However, studies using CK metrics do not consider the amount of “information” passing through a given module of the software network. 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 exploited to study software systems [113, 101].

2.3.2 Social Network Metrics

The first part of this section explains the most important definition in SNA, then relevant metrics for software systems study are introduced along with their meaning.
Each node of a network belongs to an Ego network and a Global network. The first is focused on one entity of the network during the analysis. The second instead takes into account all the participants of the network.
Social network measures of this dissertation are mainly focused on EGO network metrics, because they provide insights into the extent to which a node depends on others and *vice versa* for the flow of information. However this paragraph describes also other SNA metrics, such as structural hole and centrality metrics, relevant for describing a nodes importance inside the network.

Ego network
For every node in the network, there exists a sub-graph composed by the node itself, called “EGO”, and its immediate neighbors. Such sub-graph is called the EGO Network associated to the node. Figure 2.1 gives an example of Ego network where the red node represents the EGO, whereas black ones are its neighbors. There are many social network metrics, and the interested reader can get a more comprehensive overview by examining [77, 52]. Being uninteresting for the purpose of this dissertation, we describe only the most important metrics related to the neighborhoods of a node:

- Size: number of nodes one-step out of EGO-node, plus itself.
- Ties: number of edges of the EGO network related to the node. It measures how dense the connections in this local network are.
- Pairs : maximal number of directed ties. It is obtained as Size * (Size-1).
- Density: number of ties divided by the number of pairs. It shows the percentage of possible ties that are actually present.

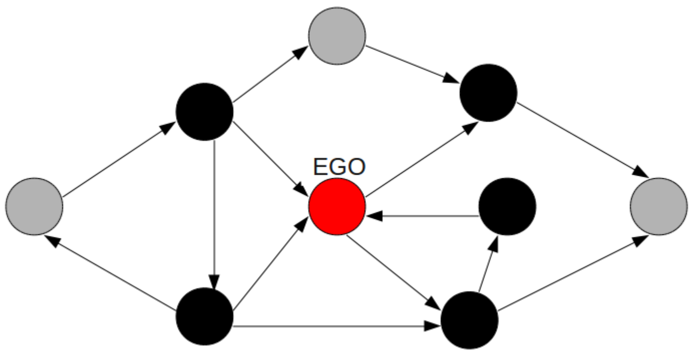


Figure 2.1: Ego Network

- Weak Components: the largest number of nodes which are connected, disregarding the direction of the ties. It measures the number of weak components in neighborhood.
- Normalized Number of Weak Components: number of disjoint sets of nodes in the EGO network without EGO node and the edges connected to it, divided by Size. This metric measures how much a node is needed to keep connected the other software units.
- Brokerage: number of pairs not directly connected in the EGO network, excluding the EGO node. This metric measures for how many pairs of nodes a given node acts like a broker, bridging the information flow among couples. High values of brokerage indicates an large number of paths which go through EGO.
- Brokerage normalized: norms the the Brokerage by the number of pairs . It describes for how many couples of node the given node acts like a broker, bridging the information flow among couples.
- Two Step Reach: the percentage of nodes that are two-step out of EGO node.
- Reach-Efficiency: normalization of Two Step Reach by size. The higher this number, the more primary contact of EGO are relevant in the network.

Structural Hole

The Structural Hole describes the positional advantage/disadvantage of a person which stemmed from their relation with neighborhoods. They were introduced by Ronald Burt [23] to explain how actor connections influence its behavior.
An example of structural hole is provided by comparing Figs. 2.2 and 2.3. As we can see in Fig 2.2, all nodes are linked together, there are no nodes in key position. The same layout is reported on 2.3, but this time node A is in an advantaged position with respect to B and C because they are not connected.

The relevant measures related to structural holes are:

- Effective size of the network: number of nodes connected to the EGO minus average number of Ties between those nodes. It measures the redundancy of the connections

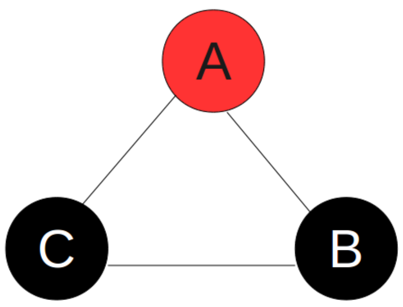


Figure 2.2: Absence of structural holes

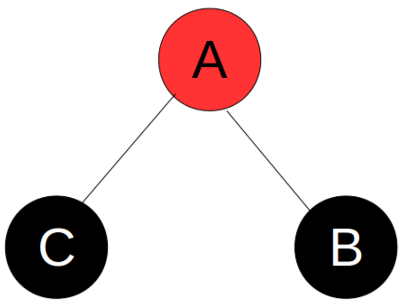


Figure 2.3: One structural hole between B and C

in the EGO network. If the average number of Ties is high, the local network has redundant channels available for the information flow, then the role of the EGO node in the information exchange is reduced.

- Efficiency: normalization of the Effective size of EGO's network by its Size.

Centrality Measures

Centrality metrics [92] can be used to measure, in the global software network, if the node plays a peripheral rather than a central role. In a software network these metrics are useful to identify a node with favored positions to get dependences. Centrality measure can be evaluated in different ways:

- Degree centrality: number of edges for a node.
- Closeness centrality: it takes into account all shortest paths from the considered node to all other nodes. It can be measured as:
  - Closeness: the sum of the lengths of the shortest paths from the node to all other nodes.
  - 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.
  - Information centrality: the harmonic mean of the length of paths starting from all nodes of the network and ending at the considered node. The node which is connected to others through many short paths has a small value of this metric.

Chapter 3

Complex Networks

This study deals with software systems described as a complex network. This chapter introduces the theory of complex networks and their characterization through the statistical analysis of key distributions of topological entities.

3.1 Complex Network theory

Complex Network (CN) theory gives a framework of general concepts useful for describing different research environments such as physics, computer science and even sociology. CN concepts can be used to describe empirical data and forecast the network topology evolution.
A Complex Network is composed of a set of entities, defined as vertices or nodes, and a set of link between them, called also edges. Thank to this simplicity, this model is flexible and able to describe a wide range of natural and artificial structures such as lattices, trees and lists.
Since some CN analysis require time consuming elaboration affordable for computers of last generations, only recently has their use been applied across lots of research fields.
Among the first who lead CN analysis were Flory [48], Rapoport [85, 86, 87] and Erdös and Rényi [40, 41, 42] who dealt with percolation and random graphs. Other relevant contributions were made by Watts and Strogatz [105] with their investigation of small-world[1] networks, and Barabási and Albert who described scale-free[2] models [11]. Then CN has also used to model the WWW [2, 11], power grids [3], biological systems [82, 10], business relations between companies, networks of citations between papers [88], software systems [30, 102, 83] and many others. The interested reader is encouraged to pursue surveys [2, 77], introductory papers [9, 54, 55, 4] and books [17, 8, 36] for an in-depth description.
Although software systems are among the most complex man made artifacts [21], they can be modeled as CNs as well. Software is decomposable in modules "connected" each other in order to cooperate and perform specific tasks. These modules are the nodes of a network

[1] The small world property was highlighted by Milgram et al. in 1967 [68]. Given a social network of people, all vertices can be reached from the others through a small number of paths. This effect has been found also in completely different networks.

[2] A function f(x) is defined as 'scale-free' if does not change when a rescaling of the independent variable x is performed

and their connections represent syntactical relationships between modules, subprograms, instructions, like the routine calls in C, or dependence in OO system.
Only recently complex networks theory have been applied to software engineering. In this perspective, OO systems are described as directed graphs whose nodes are classes and interfaces, and whose edges are the relationships between classes, namely inheritance, composition and dependence. In [104] complex software networks were analysed with nodes representing software entities at any level, and connections represented relationships among them. In [64] 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 [113] nodes are binaries, and edges are dependencies among binaries pieces of code. In [107] inter-class relationships were examined in three Java systems, and in [14] the same analysis was replicated on the source code of 56 Java applications. Current interest is focused on topological features of complex[3] network like degree distributions and hubs in software systems.
Object graphs were analysed in [83] in order to reveal scale-free geometry on OO programs, where the objects are the nodes and the links among objects are the network edges.
Potanin et al. [83] showed the existence Power Law distributions on run-time objects of OO systems, the same result were obtained on static class structures by [102, 103]. Similar results are obtained by Myers [75], Wheeldon and Counsell [107] and Marchesi [65].

The study of structural properties of software systems and its time evolution are commonly performed to investigate how the connections of the software modules change. Managers can take advantage of these analysis to keep control of software evolution complexity.
Complex network theory helps in this analysis because it gives a new perspective to characterize the mechanisms which rule software generation and evolution, or the interconnections among network modules. From this perspective, a new set of metrics can be introduced, or adapted and then correlated with other software metrics.
These network metrics are essential for accomplishing investigations, characterization, classification and modeling of software products. Application of such measurements can be the discrimination between different classes of structures, or to compare a theoretical model with a real networks. In the following paragraph we will introduce these CN concepts to bug data, studying how they change during the software evolution.
This dissertation, differently from previous studies, deals with bug metrics clarifying some issues related to:

- the distribution of bugs and its time evolution. Some key distributions, used in literature to model software properties, are compared and analysed.
- generative models able to produce such bug distributions. These models are applied to exploit the mechanism of bug introduction in software modules.

3.2 Software Metric Distribution

The distribution analysis of general properties of software graphs, giving a quantitative measure — or metric — of specific properties of complex network, is a hot topic in software en-

[3] The adjective "complex" of complex networks is related to the analysis of statistical properties of the whole network composed of millions of nodes.

gineering. This is supported by some recent work [30, 64] devoted to the investigation of the distribution of properties 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 [78].
Distribution analysis is an important step to develop a statistical model able to estimate the future value of software metrics. For instance, given n modules characterized by a metric distributed according to a power-law with exponent γ, the average maximum expected value for this metric in the module with highest metric value, $< x_{max} >$, is given by the formula [78]

$$< x_{max} >= n^{1/(\gamma-1)} \tag{3.1}$$

This formula provides a definite expectation of the maximum value taken by the metric, and hence allows us to flag specific modules with metric value of this order of magnitude. We can leverage such information to carefully select which parts of the software project are worth of more care and effort. Furthermore, distribution analysis gives some hint for the identification of a generative model responsible for a specific distribution.
Unfortunately, it is hard to identify a generative model able to describe each software metric. For this reason, when needed, we concentrate our effort on bug analysis. Bug analysis might help the management to organize pre- and post-release testing estimation of the future rate of bugs in software modules.
Usually, to mathematically synthesize the statistical behaviour of software distribution, a histogram of the software metric is computed from the empirical data, and then it is matched with some known probability distribution function (PDF). The same result can be also achieved using the cumulative distribution function (CDF) Eq. 3.3 and the complementary cumulative distribution function (CCDF) Eq. 3.4. Denoting by $p(x)$ the probability distribution function, by $P(x)$ the CDF, and by $G(x)$ the CCDF, we have:

$$G(x) = 1 - P(x) \tag{3.2}$$

$$P(x) = p(X \leq x) = \int_{-\infty}^{x} p(x')dx' \tag{3.3}$$

$$G(x) = p(X \geq x) = \int_{x}^{\infty} p(x')dx' \tag{3.4}$$

The CCDF represents the probability that the metric, X, is greater than or equal to a given value x.
Generally, software metric distributions used in literature exhibit a “fat-tail” distribution. It means that a small fraction of the population takes over a large portion of the measured resource. This means that, although lots of “entities” exhibit small values of a metric, there exist also a non-negligible probability of finding metric values of even orders of magnitude greater than the average. For this reason, when we deal with big samples, we can find easily very high metric values.
As matter of fact when the distribution exhibits a fat-tail, traditional statistics such as average or standard deviation may lose their meaning, and may be not characterizing statistics anymore [78]. In the next section we present typical key distributions used to model software metric distribution, highlighting which of them have a generative models able to produce such data in a bug context.

3.2.1 Power-Law

The Power Law (PL) function is mathematically formulated as $p(x) \simeq x^{-\alpha}$, where α is the power-law coefficient, known also as the exponent or scaling parameter. Alpha is the only parameter which characterizes the distribution, besides a normalization factor. The hallmark of a power-law distribution is a straight line exhibited when it is plotted on log-log scales.
Since for $\alpha \geq 1$ the function diverges in the origin, it cannot represent real data for its entire range of values. If $2 < \alpha \leq 3$, as typical in real-world situations, the mean of the distribution converges, but its standard deviation diverges as new measurements are added. If $\alpha > 3$, both the mean and the standard deviation of the distribution converge. Differently from exponential or normal distributions, in a PL distribution the probability of finding extreme values is not-negligible. This happens because this distribution exhibits a “fat-tail”. 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.

3.2.2 Double Pareto

The Double Pareto distribution is an extension of the standard power-law, in which two different power-law regimes exist. In literature there are different forms for the Double Pareto distribution [91, 89, 69, 107]. We use for the CCDF the form described in [97], because it provides a smoother transition across the two different power-law regimes:

$$G(x) = 1 - \left[\frac{1 + (m/t)^{-\alpha}}{1 + (x/t)^{-\alpha}} \right]^{\beta/\alpha} \tag{3.5}$$

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 software, providing the correct distribution of files size [90, 69] . 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 [69].

Large software systems can be divided into many subsystems that are worked on in parallel. From time to time, some of these subsystems can be declared stable, and thus “frozen”, with no further activity performed on them. This maintenance procedure is clearly more realistic than a single monolithic system whose modules are all likely to be updated in the same way, as in the generative process associated with the Log Normal distribution. Applying this vision on bug introduction, it can be assumed that new bugs are introduced into existing modules selected at random, in amounts proportional to the already inserted bugs (multiplicative factor). This model, which explicitly includes system growth, may be suitable for software systems, where bugs are inserted into existing modules and modules may be newly created as well as discarded.
The main drawback of this distribution is that it cannot include zeros, so it does not take

into account modules with no bugs. 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; models with less parameters are generally preferable.

3.2.3 Yule-Simon

The Yule-Simon distribution was introduced to explain the power-law in the tail of the distribution of genera and species in nature [110], and of the frequency of words in books [95]. 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, 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.

If we focus our attention on bug introduction, we will have as entities the modules and as properties the bugs. New modules are created and added to the system, and new bugs are introduced into existing modules. The preferential attachment may thus apply.
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's beta-function – in turn depending on the Euler Gamma function – depending on two parameters, α and c [78]:

$$p(x) = p_0 \frac{B(x+c,\alpha)}{B(h_0+c,\alpha)} \tag{3.6}$$

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \tag{3.7}$$

Since the beta-function has a power law behaviour in its tail, B(a,b) ~ a^{-b} , the Yule process creates a power-law distribution $p(x) \sim x^{-\alpha}$.

3.2.4 Log Normal

The Log Normal distribution has been proposed in the literature to explain different software properties ([14, 29, 64]). Mathematically it is expressed by:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-\frac{(ln(x)-\mu)^2}{2\sigma^2}} \tag{3.8}$$

The formula exhibits a quasi-power-law behaviour 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 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.
If we model a stochastic process in which new elements (i.e. bugs) 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 of being selected for the introduction of new elements [78].
This general scheme can be applied on bug introduction in 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 of developing 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.

3.2.5 Weibull

The Weibull distribution is generally used to explain the rate of faults in a technical system, due to the failure of single components. Its CCDF is given by eq. (3):

$$G(x) = e^{-(\frac{x}{\gamma})^{\beta}} \tag{3.9}$$

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 [108]. Weibull distribution is very flexible, and was found to fit very well the Alberg diagram of bugs [111], with shape parameter $\beta < 1$.

This sub-paragraph has shown the distribution functions used traditionally to fit software metrics; moreover it pointed out when possible how these distributions can be used to describe a generative model for bug.
From this point of view it is important to notice that:

- The Weibull model is related to a system made of components which fail with a given rate in time. It cannot account for the empirical evidence of bugs (failures) added to modules with specific statistical rules, typical of complex systems, whose presence is suggested by the fat-tail in the bug distribution.

- The Log Normal, Pareto and the Double Pareto models, however 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.

Chapter 4

Experimental Setup

This dissertation deals with the statistical properties of OO systems represented as complex networks. We interpret this analysis within the boundaries of software engineering, performing a correlation study of software network properties, and an analysis of the time evolution of software.
This chapter describes how we represented software as complex network. Once built this network, it reports which metrics are extracted for correlation analysis. Finally, it is explained which representation we have chosen for distribution analysis during time evolution of software. The studies of bug proneness and its time evolution [33, 32, 73, 74] are considered using Eclipse and Netbeans case studies because they offer open access to software code and public availability of bug reports.
The remainder of this chapter provides the following content:

- section 4.1 introduces the software configuration management system and the bug tracking (BTS) system used by software projects analysed for the bug proneness study. This paragraph illustrates also how we extracted software modules from project repositories and how we mapped them with software bugs hosted in BTS.
- section 4.2 explains which metrics are extracted from software graph and why.
- section 4.3 illustrates which perspective we used to highlight the hallmark of statistical distribution of extracted metrics.

4.1 Data mining from CVS and BTS

Our analysis are based on open source systems, which give free access to the source code repository and bug tracking system.

4.1.1 CVS

Software configuration management systems like CVS (Concurrent Version System) keep track of all maintenance operations on software source code. Unfortunately, these operations are recorded inside CVS in an unstructured way; it is not possible, for instance, to query CVS to know which operations were done to fix Bugs, or to introduce a new feature or enhancement.

Listing 4.1: File Log Structure

```
 <?xml version="1.0" ?>
- <changelog xmlns="http://www.red-bean.com/xmlns/cvs2cl/">
+ <tagdate>
+ <entry>
+ <entry>
+ <tagdate>
 ...
+ <tagdate>
+ <entry>
+ <entry>
+ <entry>
+ <tagdate>
</changelog>
```

Listing 4.2: Entry Node Structure

```
- <entry>
    <date>2007-01-27</date>
    <weekday>Saturday</weekday>
    <time>18:42</time>
    <isoDate>2007-01-27T17:42:42Z</isoDate>
    <author>darins</author>
    - <file>
        <name>src/org/eclipse/ant/core/AntCorePreferences.java</name>
        <cvsstate>Exp</cvsstate>
        <revision>1.99</revision>
        <tag>v20070128</tag>
    </file>
    <msg>Bug 167291 - Unable to create XML editor </msg>
  </entry>
```

Data is stored inside CVS make its repository, and each change on this data is recorded inside a log file. When a developer uploads —commits —their source code to the repository, it is introduced as an "entry". 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 files interested by the commit. All the system classes are contained in these Java source files, which are commonly called Compilation Units (CU). Besides the different reasons of a maintenance activity on source code, fixing-issue[1] activities are the most common. These operations usually refer to bug fixing, but they can describe a feature introduction or an enhancement. CVS does not keep track of this Bug resolution; for this reason, a parsing of commit messages is required. For the purpose of our study is in fact fundamentally a mapping between Bugs (and Issues) and Compilation Units (subsection 4.1.3 will better clarify

[1] A fixing-issue activity is a maintenance operation where an Issue is addressed

this aspect).
The structure of the original log file is not well organized to extract data. For this reason, we adopted the tool Cvs2cl[2] to obtain an easily manageable XML file.
Fig 4.1 shows the structure of XML log file, where the root node reports the software used Cvs2cl. The useful information for our research is inside the entry node. Fig 4.2 describes the 3 blocks hosted inside the entry node.

- block 1 (from line 2 to 6) reports the commit's author and the time of this upload
- block 2 (from line 7 to 12) lists the file(s) uploaded and report to which subproject they belong
- block 3 (line 13) clarifies the maintenance activity done on uploaded file(s)

The example in Fig 4.2 describes a fixing-issue activity on Bug 167291, which hits AntCorePreferences.java file belonging to ant.core subproject.
In block 3, the developer in charge of the code change reports the fixing-issue activity through a commit message, where the reported number is the Issue-ID. However, commit messages, including positive integer numbers, might instead refer to maintenance operations not related to Issue-ID resolution, such as branching, copyright updating, release building and so on. Unfortunately, these comments are optional, and not standardized. Thus, it is not simple to obtain a correct mapping between Issue(s) and the related CUs [96, 72].

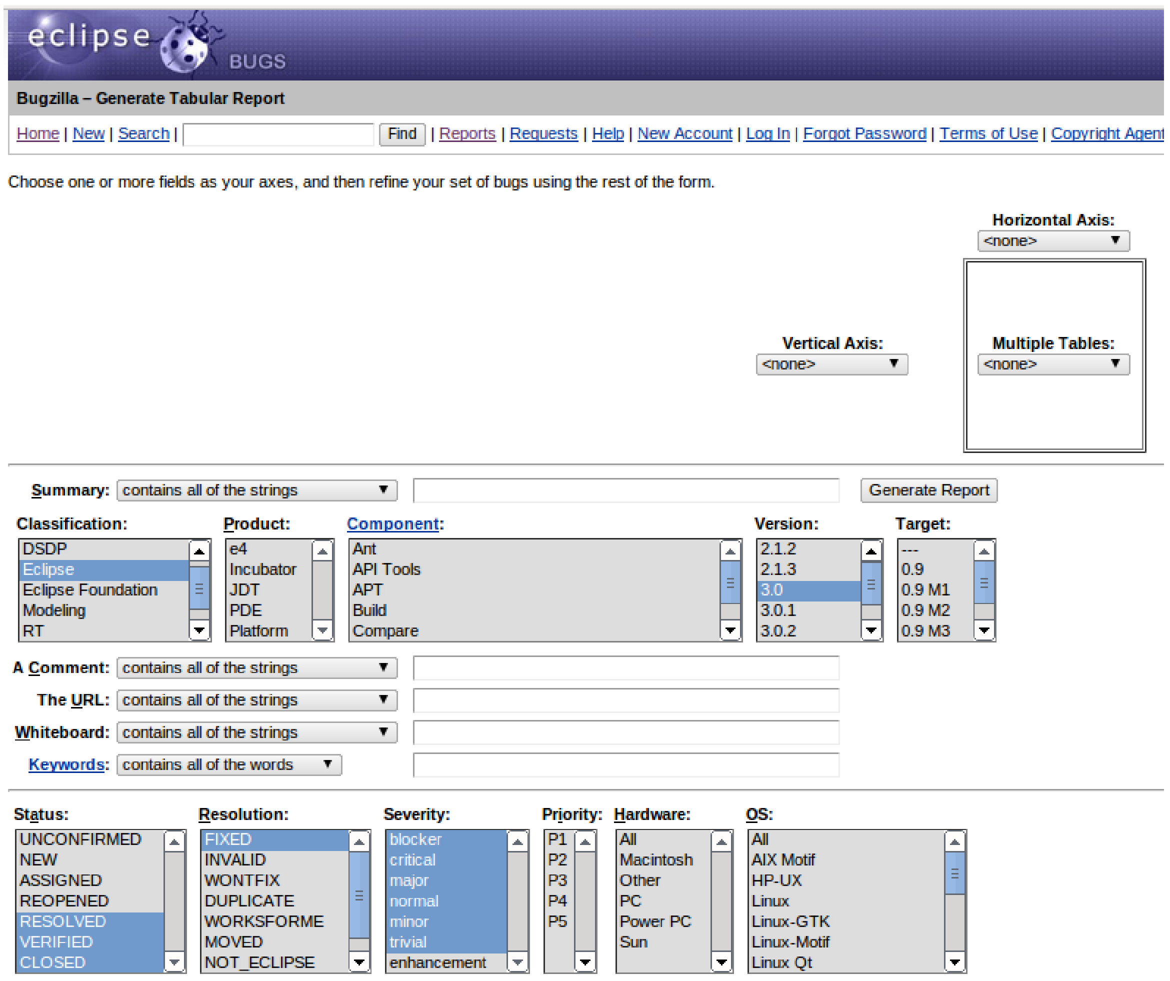
4.1.2 BTS

A Bug Tracking System (BTS) keeps track of Bugs, enhancements and features —all of which are called "Issues" —of software systems. The open source systems studied, Eclipse and Netbeans, make use of the BTS Bugzilla[3] (fig. 4.1) and Issuezilla[4], 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. [37]. 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, status as above, and with type *defect*.

4.1.3 Data Mapping between CVS and BTS

Software source code hosted in CVS, can be decomposed into modules with different granularity like functions, classes, files, collection of files; each one bears different information and allows system analysis at different levels. In this scenario, modules represent graph nodes, and edges describe relationships among nodes.
Is some of our experiments [31] modules refer to classes or interfaces, whereas in [73, 74, 33] we use CU-file as software modules. In the first case a directed graph is associated to OO software systems, where the nodes are the classes or the interfaces, and the directed edges

[2] http://www.red-bean.com/cvs2cl/
[3] https://bugs.eclipse.org/bugs/
[4] https://launchpad.net/bugs/bugtrackers/netbeans-issues



Bugzilla – Generate Tabular Report

Home | New | Search | Find | Reports | Requests | Help | New Account | Log In | Forgot Password | Terms of Use | Copyright Agent

Choose one or more fields as your axes, and then refine your set of bugs using the rest of the form.

Horizontal Axis: <none>
Vertical Axis: <none>
Multiple Tables: <none>

Summary: contains all of the strings — Generate Report

Classification:	Product:	Component:	Version:	Target:
DSDP	e4	Ant	2.1.2	---
[✓ Eclipse]	Incubator	API Tools	2.1.3	0.9
Eclipse Foundation	JDT	APT	[✓ 3.0]	0.9 M1
Modeling	PDE	Build	3.0.1	0.9 M2
RT	Platform	Compare	3.0.2	0.9 M3

A Comment: contains all of the strings
The URL: contains all of the strings
Whiteboard: contains all of the strings
Keywords: contains all of the words

Status:	Resolution:	Severity:	Priority:	Hardware:	OS:
UNCONFIRMED	[✓ FIXED]	[✓ blocker]	P1	All	All
NEW	INVALID	[✓ critical]	P2	Macintosh	AIX Motif
ASSIGNED	WONTFIX	[✓ major]	P3	Other	HP-UX
REOPENED	DUPLICATE	[✓ normal]	P4	PC	Linux
[✓ RESOLVED]	WORKSFORME	[✓ minor]	P5	Power PC	Linux-GTK
[✓ VERIFIED]	MOVED	[✓ trivial]		Sun	Linux-Motif
[✓ CLOSED]	NOT_ECLIPSE	enhancement			Linux Qt

Figure 4.1: Eclipse Bug Tracking System Bugzilla

are the various kinds of relationships between them, for instance inheritance, composition, dependence. In the second case, we adopted and extended to CU the concept of OO class graph, namely using CUs as nodes.
Once built, the software graph is required to identify nodes hit by Issues[5]. In order to do that, it is necessary to cross check CVS log file with issue-IDs contained in the BTS.
In our approach, we first analyse the CVS log, to locate commit messages associated to fixing-issue activities. Then, the extracted data is matched with information found in the BTS. The provided information allowed us to discriminate Bugs from other Issues.
Relying only on the analysis of fixing messages is insufficient; often programmers describe ambiguously the operations made, and which bugs were fixed. Each Issue is identified by a whole positive number (ID). In commit messages it can appear a string such as “Fixed 141181” or “bug #141181”, but sometimes only the ID is written. Clearly, every positive in-

[5] We consider a CU as affected by an Issue when it is modified for fixing-issue. We recall that an “Issue” reported in a BTS has a broad sense. It may denote an error in the code, but also an enhancement of the system, or a features request, or fixing a requirement error. Moreover, when many CUs are affected by a single Bug, it is possible that some of them are in fact modified not because they are defective, but as a side-effect of modifications made in other CUs

teger number is a potential Issue. Obviously, if we labeled each ID as an Issue, we would consider as bugs ID numbers bearing a complete different meaning. To cope with this problem, we applied the following strategies:

- we considered only positive integer numbers present in the BTS as valid Issue IDs related to the same release;
- we did not consider some numeric intervals particularly prone to be a false positive Issue ID.

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 Issues (Bugs) and CUs [7]. We checked manually:

- 10% of CU-Issue(s) associations (randomly chosen) for each release;
- each CU-Issue association for 6 subprojects (3 for Eclipse and 3 for Netbeans) without finding any error. A bias may still remain due to lack of information on CVS [7].

The total number of Issues affecting a CU in each release constitutes the Issue-metric[6] we consider in this study. The files stored in CVS, mainly written in Java, can be downloaded in a local machine through a command denominated "check out". All the system classes contained in these Compilation Units, generally contains just one class, but less frequently they may contain two or more classes[7].

4.1.4 Related work

A large amount of work addressed parsing problems of commit messages on CVS [34, 46, 47, 96, 112]. This is in fact the first step to realising mappings between data stored in CVS and BTS. Generally, these works identify fixing-issue commits searching for fixed regular expressions in the commit message text.
Cubranić and Murphy [34], Fischer, Pinzger, and Gall [46, 47], were among the first to use this characteristic to map issue-IDs on CVS and on BTS. Fischer at al. [47], analyzing the Mozilla project, presented a method to create a relational database for storing information found in CVS and BTS. The critical step in this activity is the identification, through pattern matching, of messages related to fixing-issue activity. A more sophisticated study was performed by Śliverski and Zimmerman [96, 112] who examined the Eclipse project. In their work, beside the pattern matching, they adopted a semantic analysis. This considers the developer who fixes the issue and the date when this issue becomes fixed on CVS and BTS.
In previous works, expressions like "Fixed 42233" or "bug #23444" are searched for. Each number can be an issue, but not all numbers are issues. Thus, to reduce false positives, they executed pattern matching looking for keywords like "fixed" or "bug" inside commit messages. Only in [96] regular expressions employed in pattern recognition are well defined.

[6] In certain analysis we will refer also to Bug-Metric, where the Bugs are a subset of Issues, as defined in 4.1.2

[7] In Eclipse 10% of CUs host more than one class, whereas in Netbeans this percentage is 30%

Previous approaches are based on a manual refinement of keywords of regular expressions to classify commits. Unfortunately, this tuning is tricky even for a domain expert. Moreover, it is subjective and therefore it can introduce a bias in commit classification. For this reason in our approach we did not use keywords for the identification of fixing-issue commits, avoiding the problem of bias introduction [72].

4.2 Metrics extraction

Once defined the software graph and identified the nodes hit by Issues, we can compute the relevant metrics for our static[8] analysis. In fact, one goal of this dissertation is to find, by means of the software graph framework, existing correlations among bugs and metrics. We focus our attention on CK metrics, because their relationship with bugs have been validated [25, 60, 12], and also on SNA metrics since recent studies have highlighted their relevance for bug study [113, 101].

CK computation
We associate to each node of the graph the values of the OO metrics computed on the class represented by it, and more specifically the four most relevant CK metrics:

- 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.

Furthermore, we computed the lines of code of the class (LOC), excluding blanks and comment lines. This is useful to keep track of class dimension because it is known that a “long” class is more difficult to manage than a short class.
Dealing with Issues-fixing there is the problem that Issues are associated to CUs and not to the single classes. To make consistent BTS data with source code, we decided to extend CK metrics from classes to CUs. For this reason in our study the Compilation Unit represents a central concept.
We defined a CU graph whose nodes are the CUs of the system. In this graph, two nodes are connected with a directed edge if at least one class inside the CU associated with the first node has a dependency relationship with one class inside the CU associated with the second node. We refer to this graph for computing in-links and out-links of a node. Taking into account this graph, we reinterpreted our metrics from classes to CUs as follows:

- CU LOCS is the sum of the LOCS of classes contained in the CU;

[8] Our analysis is static, since we do not consider run-time dependencies between classes.

- 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 6 metrics: In-links, Out-links, CU LOCS, CU LCOM, CU WMC, CU RFC and CU CBO[9].

SNA computation

On the previously defined CU software graph, we computed some metrics used in Social Network Analysis. We restricted ourselves to the subset of SNA metrics, mostly belonging to Ego network, that were found most correlated to software quality [113, 31, 100]. In the definition of the EGO network, we considered the graph links as un-directed links.
We take into account also social network global metrics to describe the role of given CU/edge is in the network. The used SNA metrics are operationally computed as follow:

- Size: size of the EGO network related to the considered CU.
- Ties: number of edges of the EGO network related to the CU.
- Normalized Number of Weak Components (NWeak-Comp): normalized Number of Weak Components. It is the number of weak components normalized by Size, i.e., WeakComp/Size
- Brokerage: the number of pairs not directly connected in the EGO network, excluding the EGO CU.
- Reach-Efficiency: the percentage of nodes within two-step distance from a node, divided by Size.
- Effective size (Eff-size): number of CUs in the EGO network minus one, minus the average number of Ties that each CU has to other CUs of the EGO network.
- Closeness: sum of the lengths of the shortest paths from the CU to all other CUs.
- Information Centrality (InfoCentrality): the harmonic mean of the length of paths starting from all CUs of the network and ending at the CU.
- DwReach: sum of all CUs of the network that can be reached from the CU, each weighted by the inverse of its geodesic distance.

All previous metrics are computed on the CU graph, and are among those studied in [113, 101].
Each CU is associated to a set of SNA, CK, outlink, inlinks and LOC metrics.

[9] We did not extend DIT and NOC reinterpretation to CU, because their meaning would be lost.

4.3 Distribution analysis

A statistically reliable forecast of the value assumed by some metrics in future system releases require a reasonable statistical description of the empirical data.
Software metrics are generally characterized by a “fat tail” which follows distributions like Log-Normal, Power Law, and so on. To appreciate their different slopes we need the right representation, because a simple plot of the histogram distribution in a logarithmic scales does not work [78]. The binning size of the histogram influences heavily how samples are grouped together. This number of samples in the tail of distributions is small, therefore the fractional fluctuations of each one of them can highly influence the bin count. The result of these fluctuations appears as a noisy curve on the plot. The tail of the distribution is noisy due to the presence of sampling errors which are determined by the reduced number of samples. Furthermore, because of the binning, the information relative to each single data is lost.
To avoid these problems, it is possible to plot the CDF (cumulative distribution function) 3.3 or CCDF (Complementary Cumulative Distribution Function) 3.4. The CCDF representation presents various advantages: 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, resulting in a major improvement of the quality of fit. An exhaustive discussion of all these problems may be found in [78].

4.3.1 Alberg diagram

Historically, software engineers who deal with bug data use the Alberg diagram [44, 5, 111]. The Alberg diagram [79] 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 $x(r)$ show the same information, and can be obtained from one another inverting 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” [1]. 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:

$$A(r) = \frac{\sum_{i=1}^{r} x(\frac{i}{N})}{\sum_{i=1}^{N} x(\frac{i}{N})} \tag{4.1}$$

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

Chapter 5

Results

This section presents the results obtained from analysis of the Eclipse and Netbeans case studies. From both projects we create the software graph and then extract the OO, CK, SNA and bug metrics across various releases. This chapter refines the general contributions reported in the introduction addressing the following research questions:

- Which trend has Issue occurrence during software development?
- Are software network metrics, such as SNA and CK metrics, significantly correlated with software bugs?
- Is Bug distribution persistent across all releases and in both analysed systems?
- Are there analytical distribution functions describing the empirical data?

We investigate the correlation between OO, SNA metrics and Bugs (Issues) considering their time evolution. This information may be used to understand, from the measure of the metric, which software properties is most related to bug proneness, and to devise possible strategies to apply during software development to control metrics values, with the goal of reducing Bug introduction. Finally, we compute the statistical distributions of software Bugs and Issues and we seek a theoretical model able to describe these distributions [33, 32]. There is in fact a lack of predictive theories about the bug distribution in large scale systems, or about how that distribution evolves. By publishing and comparing these data with other research, we can support or reject conventional wisdom related to bug metric distributions, software bug evolution and relationship between software attribute and bug proneness.
This chapter is organized as follows, section 5.1 describes the case studies of our analysis. Section 5.2.1 characterizes the Issue occurrence in Eclipse and Netbeans during their software development. Correlations among OO, SNA and Bug (and Issue) metrics are introduced in section 5.2.2. The core of this chapter is section 5.2.3 where the Bug distributions of Eclipse and Netbeans are investigated. The chapter ends reporting related work 5.3.

5.1 Case Studies

Eclipse and Netbeans are two large Java projects in the domain of integrated development environment (IDE). Both are mature and successful software tools derived from open source

Table 5.1: Details of case studies

Eclipse			Netbeans		
Release	Date	# Compilation Unit	Release	Date	# Compilation Unit
2.1	2003/03/28	8017	3.2	2001/04/30	3388
2.1.1	2003/06/03	8022	3.2.1	2001/08/22	3388
2.1.2	2003/11/04	8013	3.3	2001/12/12	4709
2.1.3	2004/03/17	8028	3.3.1	2002/02/01	4709
3.0	2004/06/25	10769	3.4	2002/08/26	6317
3.0.1	2004/09/17	10773	3.4.1	2003/01/20	6321
3.0.2	2005/03/18	10773	3.5	2003/06/09	7148
3.1	2005/06/26	12436	3.6	2004/03/13	8338
3.1.1	2005/09/29	12444	4.0	2004/12/15	9438
3.1.2	2006/01/26	12444	4.1	2005/05/11	10941
3.2	2006/06/28	14775	5.0	2006/01/31	13017
3.2.1	2006/09/29	14800	5.5	2006/10/30	16649
3.2.2	2007/02/14	12324	5.5.1	2007/05/24	17119
3.3	2007/06/24	16201	6.0	2007/12/03	38625
3.3.1	2007/09/28				

projects, with years of development. These projects give free access to their CVS repository, a key factor for our research. Another important property of these two projects are their similarities, which are useful to compare them.
The development of both analysed projects proceeds by main releases and patching releases. A Main Release (MR) is denoted by a two-digit decimal number - such as 3.2 or 4.0 - and represents a major step in functionalities. A Patching Release (PR) is denoted by a third number following the name of the MR they are updating - for instance 3.2.1 is a PR of MR 3.2. PRs do not add substantial features, but fix the Bugs and other Issues found in previous versions of the release. After a MR is released, its source code is used for fixing bugs through "patches", resulting in subsequent PRs. The work to produce a new MR, on the other hand, proceeds in parallel, and when another MR is released, its source code may be quite different from the code of the previous MR. Note that some MRs do not have a PR, but directly evolve into another MR. In both examined projects, there is approximately a MR every 8 - 12 months. Eclipse delivery process is more regular than Netbeans, because almost each MR is delivered after a year and each MR has two or three associated PRs. In Netbeans MRs follow a less regular delivery. We considered all the source code of Eclipse and Netbeans available in CVS repositories, that is the core system and the main add-ons. Table 5.1 shows the number of CUs of the considered MRs of both Eclipse and Netbeans projects. Both systems exhibit a steady growth in their number of CU during software evolution.

5.2 Data Analysis

5.2.1 Issue evolution

This section characterizes the Issue occurrence phenomenon in both projects, analysing how maintainers fix Issues during software development process. That description, giving a view of software behaviour over time, shows how Eclipse and Netbeans developers maintain their software system during its development.
It is known that software companies try to satisfy their audience offering a different mix of features and quality. Netbeans and Eclipse, having the same target audience, compete to satisfy the same customer requests and quality expectations. For this reason, dealing with software products with analogous objectives, our analysis suffer less from commercial variables and thus our comparison between both system maintenances is simplified.

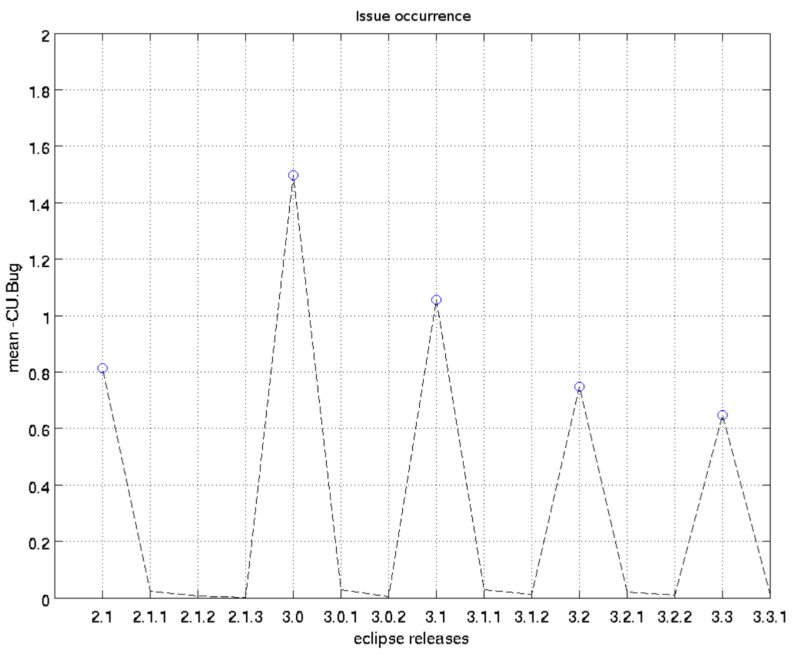


Figure 5.1: Evolution of mean number of Issues hitting a CU in Eclipse. MRs are indicated by circles.

Fig. 5.1 shows the regular trend of the mean value of number of Issues for CUs in Eclipse. For each MR, the number of introduced Issues is high, while in the following PRs this number decreases conspicuously. It is the first PR which follows an MR which reduces drastically the Issues, then the following PRs address almost all the remaining Issues. When a new MR is delivered, its new features are tested by the wide community of users. For this reason, a large number of Issues, generally Bugs, are signaled on it. In following PRs the developers tries to solve them as best as possible. Thus the number of Issues and Bugs discovered decreases quickly. More than one PR is needed because some Bugs are discovered also after the delivery of a PR. Figure 5.1 shows also a clear, steady reduction with time, of the mean number of Issues hitting a CU, starting from release 3.0.
Netbeans exhibits an Issue trend similar to Eclipse. In Figure 5.2, we can observe the pairs of

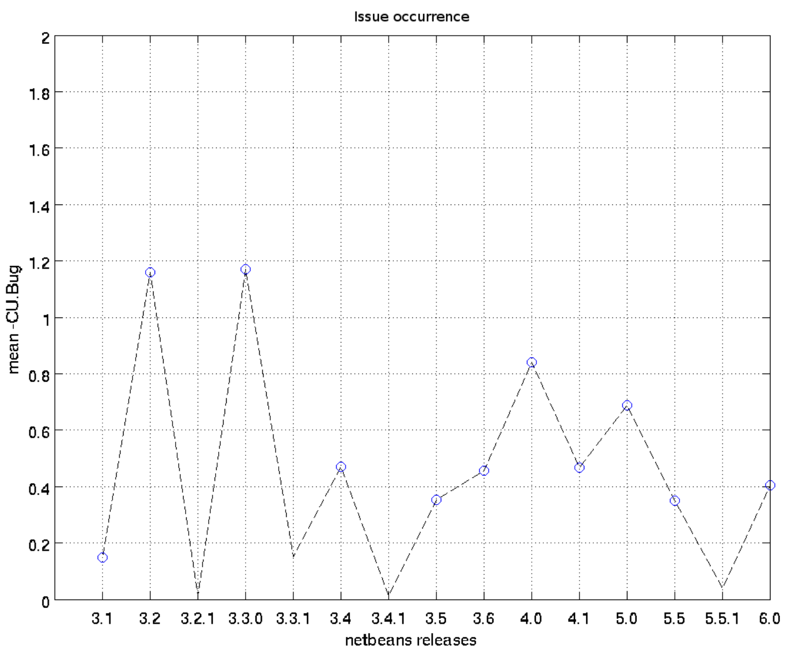


Figure 5.2: Evolution of mean number of Issues hitting a CU in Netbeans. MRs are indicated by circles.

MR and PRs 3.2-3.2.1, 3.3-3.3.1 and 3.4-3.4.1. For each pair, there is first an increment of the number of Issues in MR and then a decrement in PR. This behaviour is the same exhibited by MR-PR pairs in Eclipse. In Netbeans, however, there is a strong variability in the number of Issues during its evolution, and no apparent steady reduction of the mean number of Issue per CU even considering only MRs, as in Eclipse.

Research Question: *Which trend has Issue occurrence during software development?* Issue occurrence in Eclipse and Netbeans is not equal. All highlighted similarities and differences are here summarized:

Eclipse and Netbeans similarities

- Main Releases introducing changes to implement new features, determine also the introduction of a significant set of Bugs and enhancement request.
- Patching Releases exhibit a substantial lower number of Issues. This means that, in both considered systems, the Bug correction and enhancement implementation after a MR has been released, is easy to make.

Eclipse and Netbeans differences

- The Eclipse development process is characterized by a higher regularity in terms of MRs and PRs with respect to Netbeans.
- Netbeans project, while characterized by an average lower number of issues per CU with respect to Eclipse, does not exhibit a foreseeable behaviour in the number of Is-

Table 5.2: Spearman correlation among metrics

	Eclipse 2.1		Netbeans 3.2	
	numissue	numbug	numissue	numbug
numbug	95%**	-	98%**	-
LOCs	46%**	46%**	47%**	46%**
WMC	38%**	38%**	44%**	42%**
RFC	46%**	46%**	44%**	42%**
LCOM	34%**	34%**	41%**	39%**
CBO	45%**	45%**	33%**	32%**
Fanin	8%**	7%**	13%**	12%**
Fanout	44%**	44%**	45%**	44%**
Reach-Efficiency	16%**	16%**	17%**	16%**
Eff-Size	41%**	41%**	38%**	36%**
Closeness	38%**	39%**	38%**	36%**
DwReach	40%**	40%**	37%**	35%**
InfoCentrality	-33%**	-34%**	-26%**	-25%**
Size	43%**	42%**	39%**	38%**
Ties	42%**	42%**	39%**	37%**
NWeak-Comp	-28%**	-28%**	-26%**	-25%**
Brokerage	42%**	42%**	38%**	37%**
** Correlation is significant at the 0.01 level.				

sues per CU, as the system evolves. It seems that the developers are less is in control of system evolution.

The Issue occurrence depends on software development process. Having no specific information, it is not possible to assess any theory able to explain this trend[1], which is however out of scope of the Thesis.

5.2.2 Metrics Correlation

In this section, we discuss topological graph properties in terms of software quality. This result is achieved by a correlation analysis of the software network properties, measured as SNA and CK metrics, with the most important software quality indicators such as the number of Bug (Issue). Since the empirical distributions of all metrics are strongly non-normal, correlations will be computed using the Spearman coefficient.

In this chapter we report the correlations only for Eclipse 2.1 and Netbeans 3.2 (Tab 5.2). Correlation coefficients in the other releases are in fact substantially similar to those reported

[1] Our assumption, supported by communications with some members of both projects, is that Eclipse is a project that has been developed using consistently practices Agile, such as Test Driven Development (TDD), refactoring and feature-driven development throughout its life-cycle. Netbeans, on the other hand, applied just feature-driven development throughout its life-cycle, while automatic testing was introduced much more gradually. We believe that at least some of the differences highlighted in our study can be due to this different level adoption of Agile practices. Clearly, this evidence is just anecdotal, and more work is needed to quantitatively assess such hypothesis.

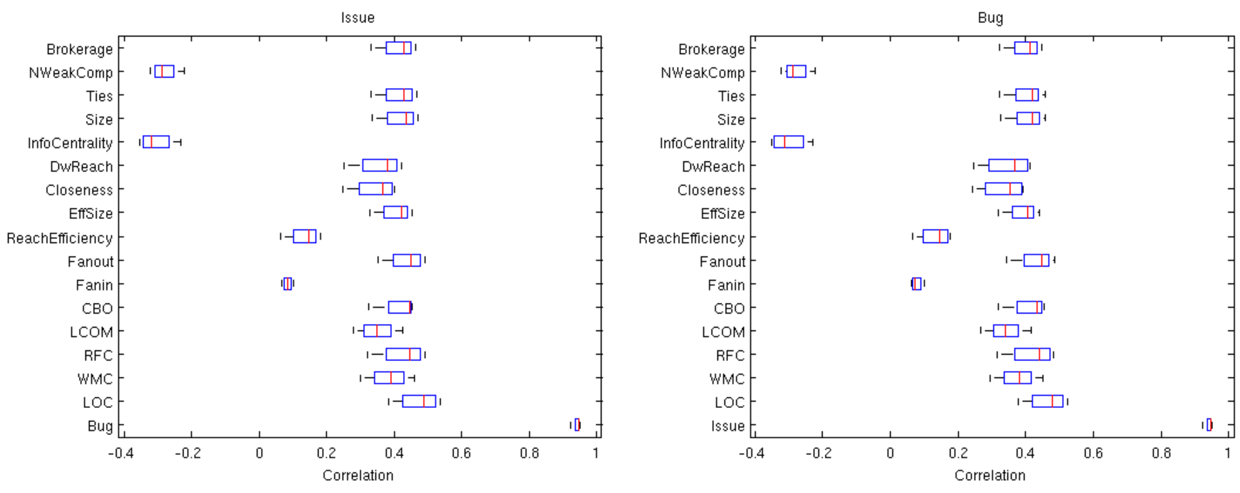


Figure 5.3: Boxplot of the correlations among Issue, Bug and the other metrics across Eclipse releases.

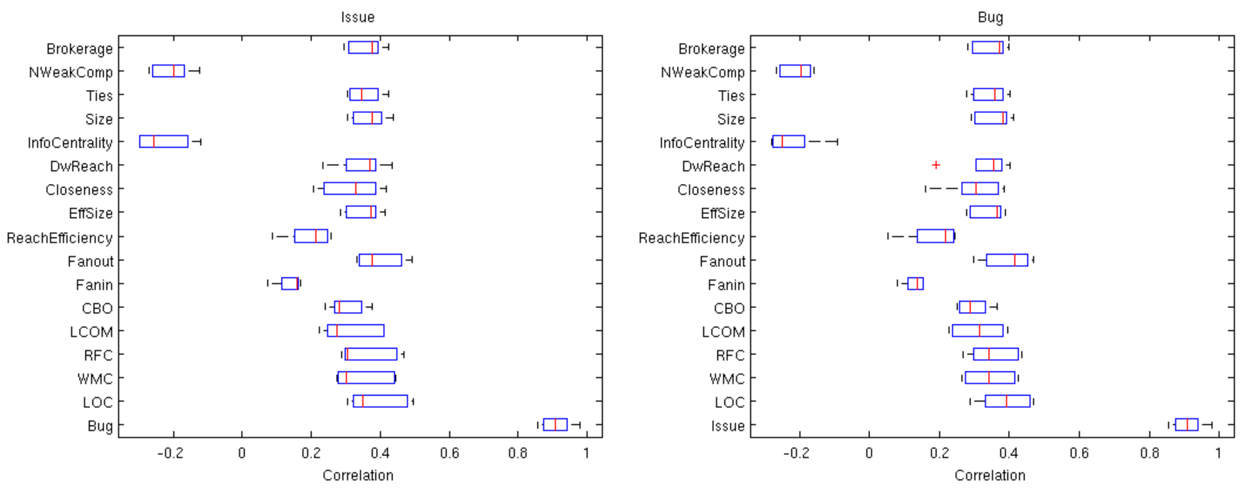


Figure 5.4: Boxplot of the correlations among Issue, Bug and the other metrics across Netbeans releases.

here[2], as highlighted by boxplots[3] in Fig 5.3 and 5.4. Other representative releases are reported in Appendix A where cross-correlation values are computed also among CK metrics, LOC, Fan-out, Fan-in and EGO-metrics.
The higher correlations are among Issues and Bugs, as 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, we will cite only one of them in the subsequent analysis.

[2] Across the different release, the values of cross-correlation in Eclipse exhibit a little spread around the median with fluctuations generally below 10%. In Netbeans this fluctuations are bigger, but generally below 15%.

[3] 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 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.

In Eclipse the CK metrics, LOCS, Fan-Out and EGO metrics generally show a moderate correlation with respect to Issues (Bugs). In Netbeans, we have similar correlations, though usually slightly smaller. In both systems, LOC metric is generally the most correlated with Issues. This would be expected, since bigger files have a larger chance to produce Issues and Bugs. However, other good predictors of Issues – comparable with LOC – are RFC, CBO, Fan-out.
On the SNA front, we observe (Tab 5.2) that many metrics are quite correlated with the number of Issues (and Bugs), confirming the importance of considering SNA for bug analysis. All show the same correlation with Issue and Bugs, for both Eclipse and Netbeans, even if they have slightly larger values in Eclipse. This suggests that the larger these metrics, the more the bugs in the corresponding modules. DwReach and Closeness are significantly correlated with Issues and Bugs. Since these two metrics are centrality metrics, namely they represent how much a module is closely connected to all others, the larger these metrics the more the module is connected to the rest of the software network. In contrast to previous metrics, the SNA suite shows some metrics, Info-Centrality and NweakComp, anti-correlated with the number of Issues. It suggests 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. These results highlight that the the social role of a node influences its Bug proneness. 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.
To understand the reason of high correlations between Bug and its good predictors, we have to recall some concepts of section 4.1.3. The number of Bugs in a module 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[4]. Thus, the larger the Size, the larger the probability of such events, and thus the number of bug corrections associated to the class. The same reasoning is applicable for RFC, CBO and Fan-out.
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. This analysis is confirmed by the Fan-in which shows only a small – though significant – correlation with Issues. The different correlation between Fan-in and Fan-out with respect to Bugs, indicates that to identify a bug-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 bugs from B to A, but not vice versa. This fact highlights the importance of an analysis of a software system as an oriented graph.
In general, these results show that the metrics related to the number of software modules directly connected to a given module are the most correlated to Issues and Bugs. Thus a module which plays the role of "star center" has more chance to be bug-prone or indirectly involved in bug fixing. We can finally answer our research questions:

Research Question: *Are software network metrics, such as SNA and CK metrics, significantly correlated with software bugs?*

[4] Another way to see this side effect is that large Fan-out (or RFC, CBO) implies an opportunity to make code adjustments in the class when the code of the called class is modified for bug fixing.

Data reported in Tab. 5.2 (and Appendix A), shows similar values in all considered releases of the studied system. On the basis of this data, we can assess that there are significant correlations between SNA, CK metrics and number of Bugs (and Issues).
Software nodes with high values of SNA metrics generally have a larger number of bugs than other nodes. The same holds for DwReach and Closeness, even if to a lesser extent. Results suggests exploration of InfoCentrality and NweakComp for software quality analysis. A high value for these SNA metrics can be in fact a hint of good software quality, in terms of the number of Bugs affecting the corresponding software modules. Significant correlation values were obtained between Bugs (Issues) and Fan-out and among Bugs (Issues) and CK metrics CBO and RFC. Thus all previous mentioned metrics can be used as rough indicators of software quality.

5.2.3 Distribution Fitting

We computed for Eclipse and Netbeans the statistical distributions of software Bugs[5]. In this analysis, we use four distribution functions - Weibull, Double Pareto[6], Log-Normal and Yule-Simon - which are suited to model sample data presenting leptokurtic behaviour. These functions are already used for software properties which exhibit a "fat tail" distribution, as is the case with many software metrics [64].
When we plot the Yule-Simon CCDFs we used all the system modules, including those with no Bugs. Whereas for Weibull, Log Normal and Double Pareto we discard these modules, because such distributions are not compatible with null values. For each distribution, we discuss how it might be linked to bug generation. This point is interesting from a software engineering perspective, since there is still not a general consensus on which type of shape has the bug distribution.
In the first part, we describe the Bugs distribution showing its persistence across Eclipse and Netbeans releases. In the second part, we compare the typical best-fit functions for Bug distribution; we then summarize pros and cons of these best-fitting functions.

5.2.3.1 Bug distribution across releases

We analyzed main releases of Eclipse, from 2.1 to 3.3, and of Netbeans, from 3.2 to 6.0. For each release, we computed the number of Bugs of each module[7]. Table 5.3 shows the basic statistics about Bug incidence, including the top hit modules by Bugs. Computed data exhibits that the 20% or even less, especially for some releases of Netbeans, of top hit modules host the 80% or more of software Bugs. More specially, a limited number of modules account for the high percentage of Bugs in the system.

Research question: *Is Bug distribution persistent across all releases and in both analysed systems?*
We found that the Bug metric has a very consistent statistical behaviour across releases of the

[5] 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 were found and fixed in a module, the more they are reduced.

[6] We do not deal with power-law - or Pareto - distribution, because it can be subsumed under the Double Pareto distribution.

[7] Software modules are files containing one or more classes.

Table 5.3: *Basic statistics of Eclipse and Netbeans.*

Project	Release	nr. of modules	total bugs	modules w. bugs	# of most hit modules	% bugs in most hit modules
Eclipse	2.1	8017	7024	2790	1376 - 17%	79.9%
	3.0	10769	16986	4989	1879 - 17%	75.7%
	3.1	12436	13836	5053	2518 - 20%	81.7%
	3.2	14775	15481	6293	3025 - 20%	78.9%
	3.3	16201	10451	4501	2025 - 13%	76.7%
Netbeans	3.2	3388	3888	1399	697 - 21%	81.9%
	3.3	4709	5206	1693	901 - 19%	84.8%
	3.4	6317	2951	1582	1582 - 25%	100%
	3.5	7148	2517	1480	1480 -20%	100%
	4.0	9438	7878	2521	1405-15%	85.8%
	4.1	10941	5475	2017	2017-18%	100%
	5.0	13017	8524	3044	3044-23%	100%
	5.5	16649	5656	2454	2454-15%	100%
	6.0	38625	17479	7410	7410-19%	100%

same system, even when these releases span over years, and have very different numbers of classes (and CUs). Figs. 5.5-5.12 shows in a log-log plot the CCDF for empirical Bug data, along with the best-fit function[8], referring to Eclipse. The same type of plots are reported for Netbeans from fig. 5.13 to 5.24. For both projects other plots are reported in appendix A. We plot the Yule-Simon distribution in separate plots since the Weibull, Log Normal and Double Pareto functions do not take into account modules without Bugs[9].
The empirical distributions of the Bug metric preserve the same shape, meaning that a single distribution function may account for the empirical data for all system releases. Moreover, this distribution also looks similar in Eclipse and Netbeans releases. Thus, since this distribution is known for Bug metric in one release, it is possible to infer the properties of that metric in other releases, whenever the number of CUs is provided. Figures 5.5 - 5.24 shows also a small cut-off in the extreme tail, which is typically due to the finite size of the sample.

5.2.3.2 Best-fit function for Bug distribution.

Figures 5.5 - 5.24 show the best-fit CCDFs for empirical Bug data referring to Eclipse and Netbeans. Table 5.4 evaluates the goodness of these fitting reporting the R^2 coefficients.

Research question: *Are there analytical distribution functions describing the empirical data?*
Watching figures from 5.5 to 5.24 we observe that:

- The Log Normal and Weibull distributions fit very well the bulk of the samples with a small number of Bugs, but in general they tend to zero too quickly with respect to empirical data.
- The Double Pareto function has a good fit meaning that the Recursive Forest model [69] can be suitable as a dynamic model for Bug generation in this object-oriented software, when discarding modules without Bugs.
- The Yule-Simon distribution outperforms the others, since it is able to obtain a good fit in the bulk and in the tail of the empirical data. Such visual qualitatively evalua-

[8] We performed best-fitting analysis using standard Matlab functions.
[9] Log-log plots do not report points corresponding to zero Bugs

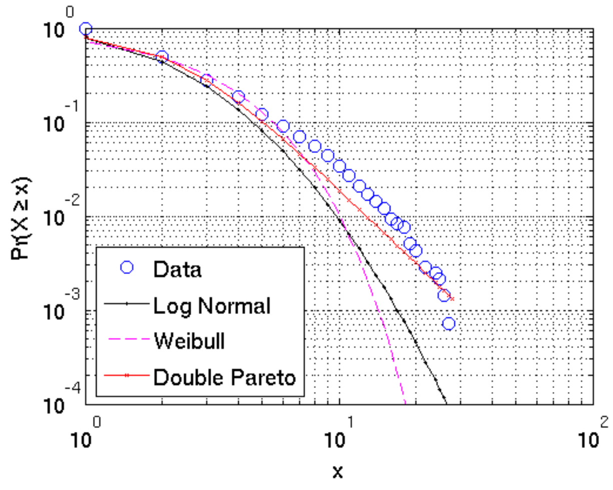


Figure 5.5: The CCDF of Bugs in Eclipse 2.1.

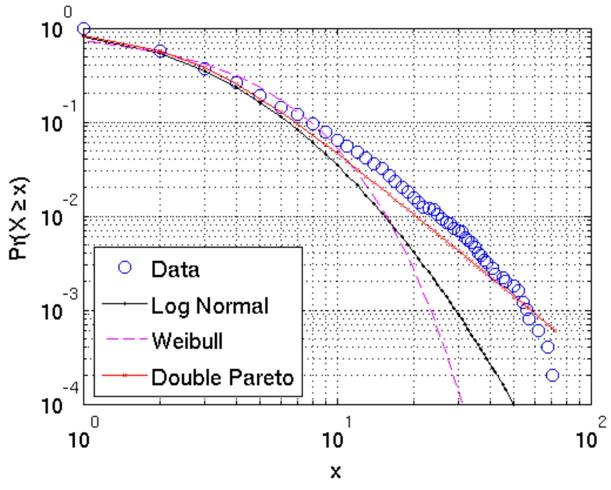


Figure 5.6: The CCDF of Bugs in Eclipse 3.0.

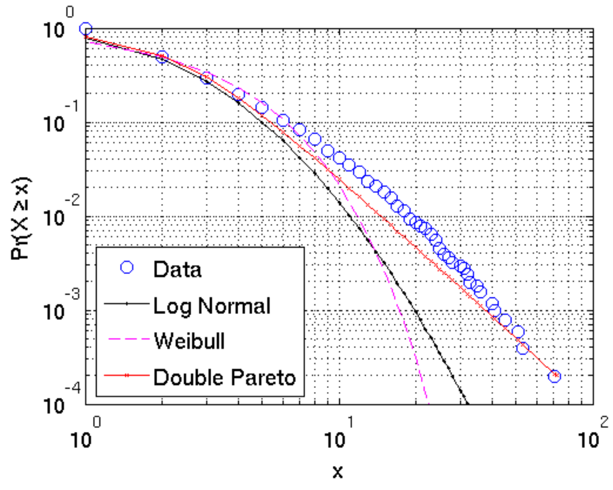


Figure 5.7: The CCDF of Bugs in Eclipse 3.1.

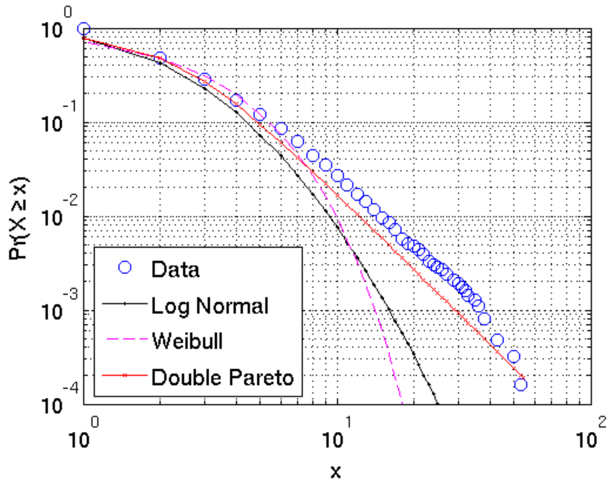


Figure 5.8: The CCDF of Bugs in Eclipse 3.2.

Table 5.4: R^2 *coefficient of determination.*

Project	Release	R^2 Weibull	R^2 Log-Normal	R^2 Double Pareto	R^2 Yule-Simon
Eclipse	2.1	0.929	0.949	0.961	0.99997
	3.0	0.947	0.967	0.975	0.99996
	3.1	0.939	0.953	0.964	0.99992
	3.2	0.931	0.951	0.963	0.99989
	3.3	0.922	0.945	0.957	0.99998
Netbeans	3.2	0.927	0.949	0.959	0.99980
	3.3	0.943	0.959	0.969	0.99966
	3.4	0.893	0.925	0.943	0.99996
	3.5	0.885	0.909	0.931	0.999992
	4.0	0.945	0.962	0.972	0.99938
	4.1	0.929	0.947	0.959	0.99986
	5.0	0.938	0.955	0.966	0.99975
	5.5	0.897	0.927	0.94	0.999978
	6.0	0.907	0.937	0.95	0.99998

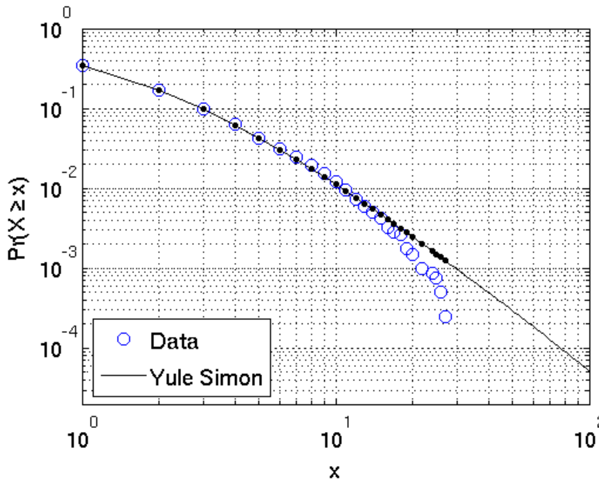


Figure 5.9: The CCDF of Bugs in Eclipse 2.1.

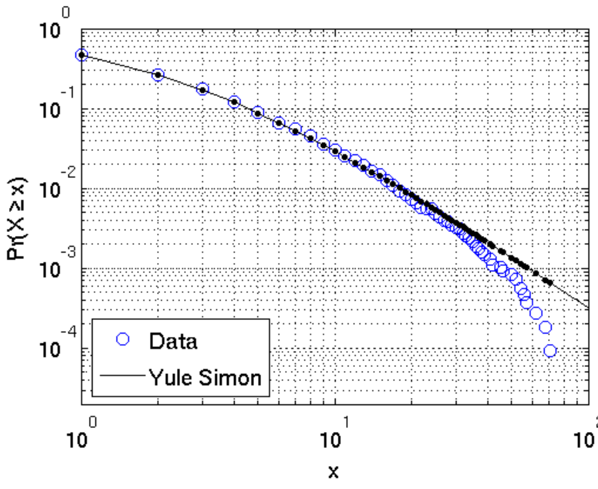


Figure 5.10: The CCDF of Bugs in Eclipse 3.0

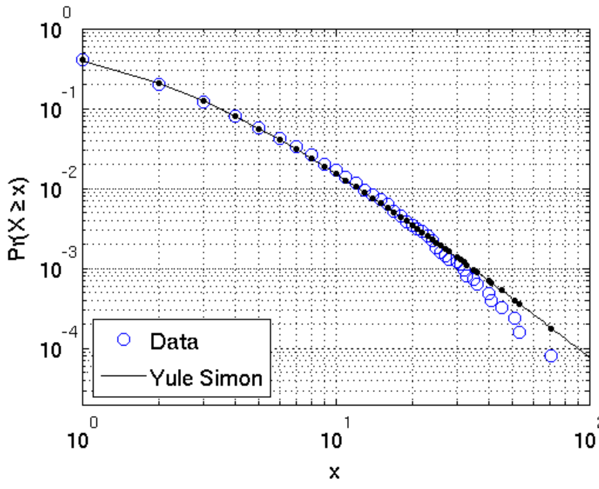


Figure 5.11: The CCDF of Bugs in Eclipse 3.1

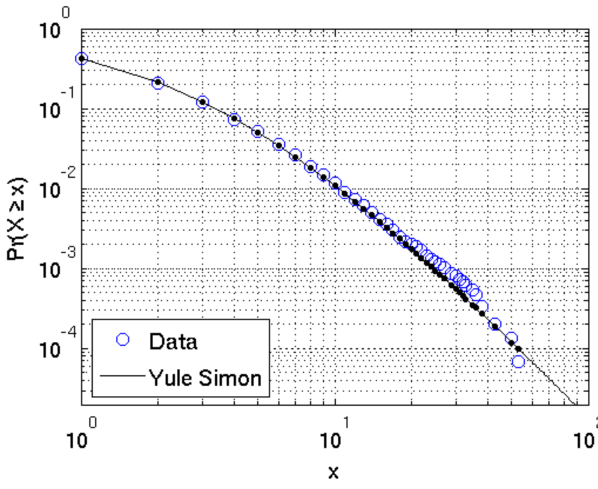


Figure 5.12: The CCDF of Bugs in Eclipse 3.2

tion is confirmed by the coefficient of determination reported in Table 5.4. Thus, this analytical distribution is the best one for describing the empirical data.

All fits have a very high R^2 determination coefficients (Table 5.4). It is worth noting 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[10] [28].
Our fitting procedure does not rely on any log-log representation of the data, we evaluate the determination coefficients on a linear scale. In that scale the fitting curves and the em-

[10] Our purpose is to provide a reasonable statistical description of the empirical data, and to find the analytical distribution function whit 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.

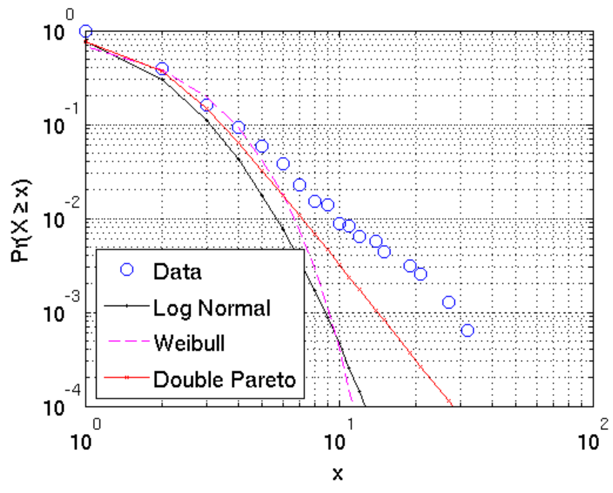


Figure 5.13: The CCDF of Bugs in Netbeans 3.4

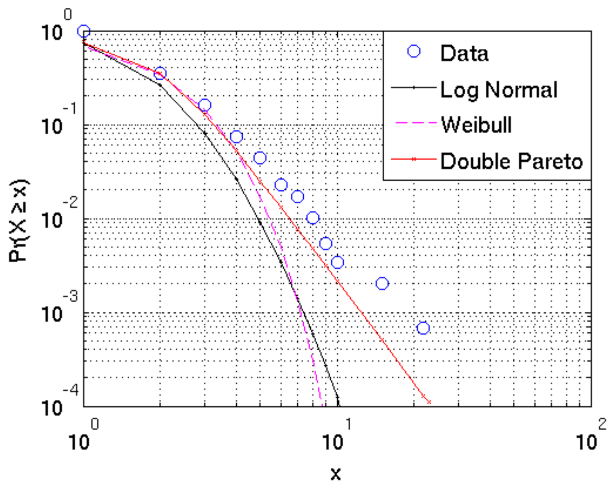


Figure 5.14: The CCDF of Bugs in Netbeans 3.5

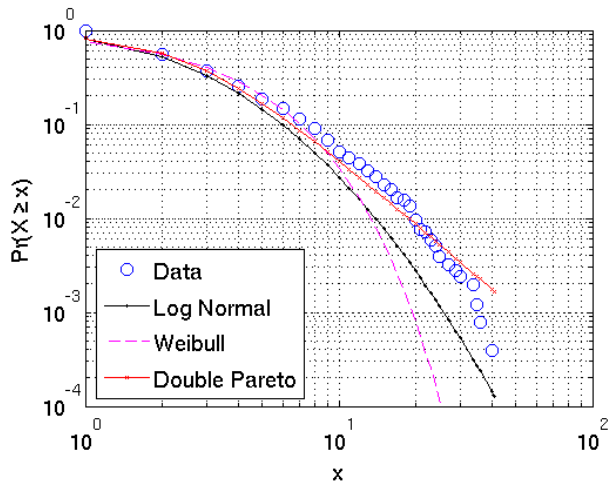


Figure 5.15: The CCDF of Bugs in Netbeans 4.0

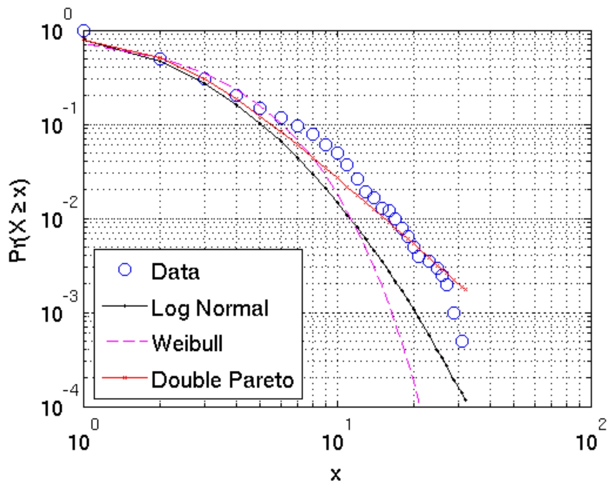


Figure 5.16: The CCDF of Bugs in Netbeans 4.1

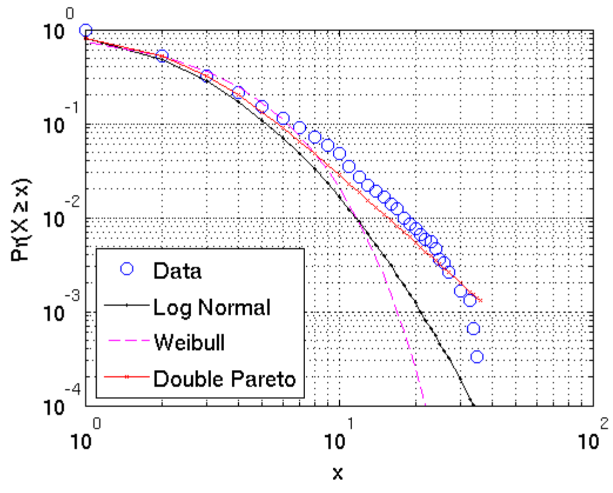


Figure 5.17: The CCDF of Bugs in Netbeans 5.0

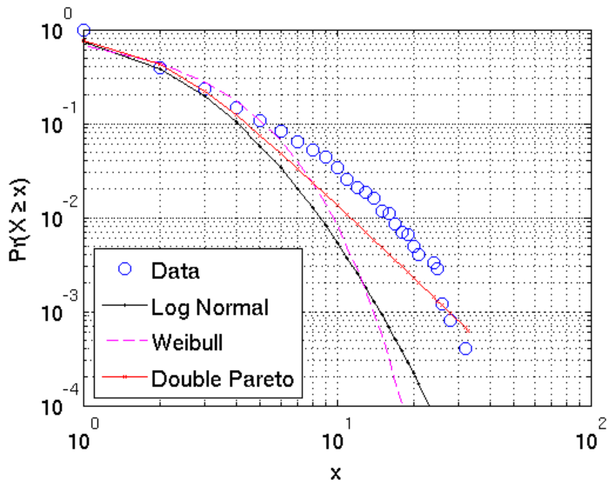


Figure 5.18: The CCDF of Bugs in Netbeans 5.5

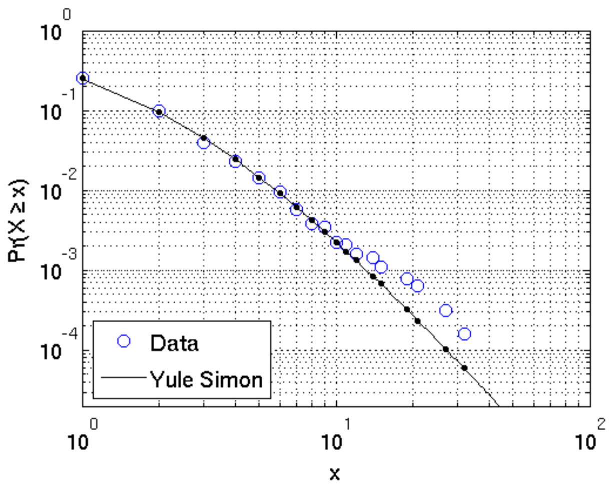


Figure 5.19: The CCDF of Bugs in Netbeans 3.4

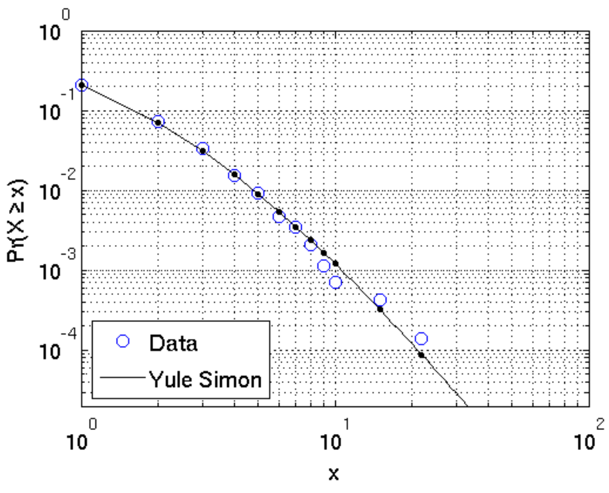


Figure 5.20: The CCDF of Bugs in Netbeans 3.5

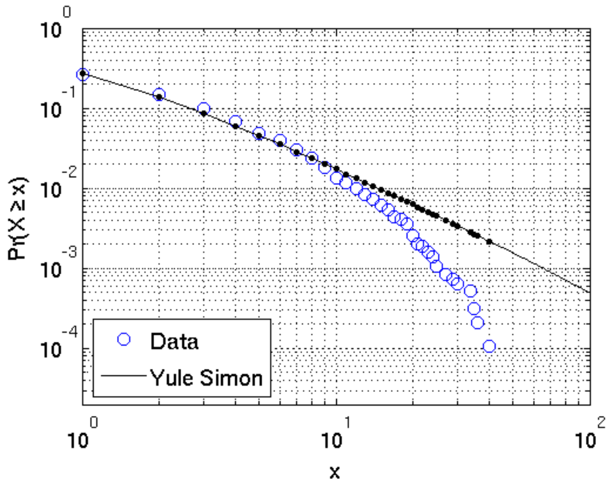


Figure 5.21: The CCDF of Bugs in Netbeans 4.1

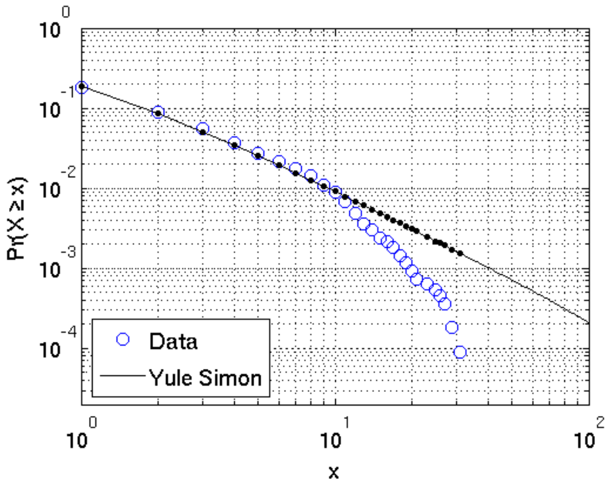


Figure 5.22: The CCDF of Bugs in Netbeans 4.1

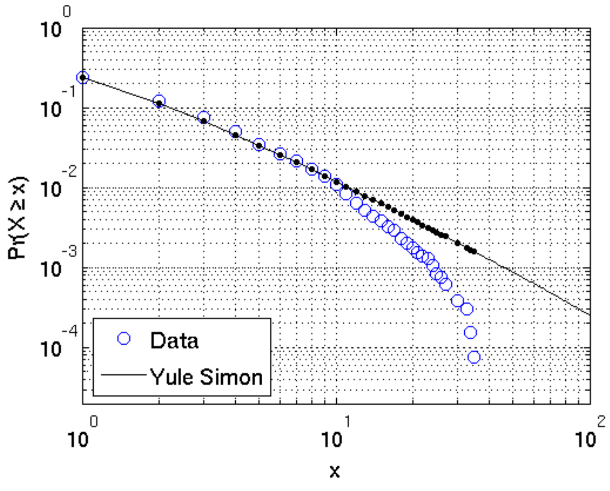


Figure 5.23: The CCDF of Bugs in Netbeans 5.0

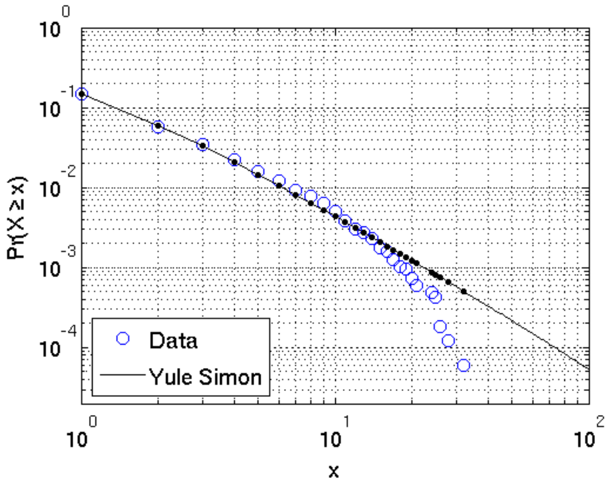


Figure 5.24: The CCDF of Bugs in Netbeans 5.5

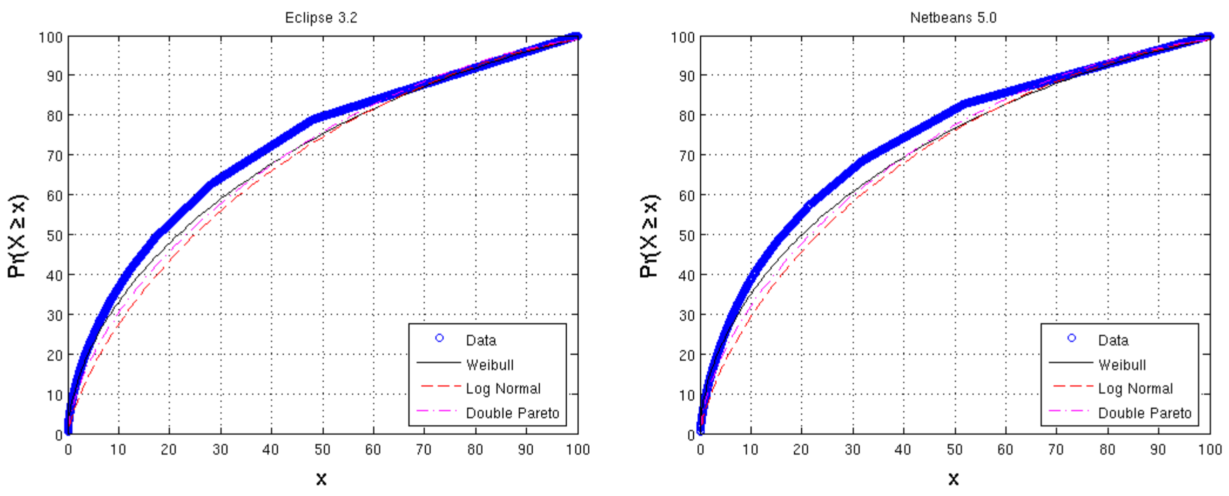


Figure 5.25: The Alberg diagram obtained from converting the fitting of the CCDF representation. Modules containing no Bugs are excluded.

pirical ones visually overlap. For this reason, we decide to plot all the figures in a log-log scale, where the discrepancy between best fitting curves and empirical curves are visually enhanced, especially in the tail.
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 [111], generating the corresponding ideal rank-frequency diagram, and then applying eq. (4.1), as described in Section 4.3.1. Fig. 5.25 reports the Alberg diagram referring to empirical data, for Eclipse 3.2 and Netbeans 5.0 (discarding zero values) and the corresponding Alberg diagrams generated from the Weibull, Log Normal and Double Pareto best-fit distributions.

In this case, from 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 R^2 coefficient, as in [111] to compare the fits, we found very similar results for the three distributions, with values of the order of 0.95. Double Pareto, Log-Normal and Weibull distributions are similar. Note that, comparing Fig. 5.25 with a similar plot reported by Zhang [111], in our case we first fit a statistical distribution using Bug data, and then generate an Alberg diagram using the theoretical distribution. In [111] 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. We believe, however, that our approach is more general, because under the hypothesis that the fitted distribution substantially holds for a significant time interval during development, our Alberg diagram will have a higher predictive value than just fitting a specific snapshot of the bugs hitting modules.

5.2.3.3 Pros and cons of best-fitting functions for modeling Bug distribution.

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 one main code quality indicator). Thus, our assumption that new Bugs are introduced into modules in amounts proportional to the already introduced Bugs (and not necessarily to the actually existing ones), is a reasonable first approximation.

On the downside, the Log-Normal model presents two significant problems. First, it does not explicitly consider system growth: new software modules are introduced during development while the Log-Normal growth occurs with regard to Bugs introduced into modules, in amounts proportional to the number that have already been discovered. The system size growth is not explicitly included, even if the same mechanism for Bug increment may always be applied after the addition of new modules.

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 for only 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 [69]. 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 [69].

In our software systems the files are the software modules, and Bugs play the role of file size. Assuming that new Bugs are introduced into existing modules selected at random, in amounts proportional to the already inserted Bugs (multiplicative factor), this model, which explicitly includes system growth, may be suitable for our software systems, where Bugs are inserted into existing modules and 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) which is larger than the other distributions, while models with less 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 the only growth 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 the Bug fixing activities that actually reduce the number of Bugs, are completely missing 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 quite naturally applies to the software development process and to the introduction of Bugs into modules. In fact, it accounts for the addition of new entities, and for the discrete increase with time of their properties – in our case the number of Bugs fixed. 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 existing entities whose property is incremented are chosen with 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, entities are software modules and properties are the Bugs introduced. As we already pointed-out, we count 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 5.5 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 then computed the average number of Bugs fixed for the same modules in the next release, 3.1. Table 5.5 shows how modules with more Bugs in a release, on average, exhibit more Bugs in the next release.

Modules with zero Bugs have a non zero chance of receiving Bugs, even if such event is more unlike than for other modules. This is again in agreement with the Yule-Simon model, where the constant c (eq. 3.6) takes this possibility into account.

The rule that can be inferred by data in Table 5.5, is that 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 3.5 and 7.6 new Bugs respectively on average, in the corresponding modules, and proportions are again 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. 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; it 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 mechanism 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

Table 5.5: 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	21	7.6
13	26	5.9
12	33	4.9
11	35	5.0
10	41	4.1
9	62	5.2
8	88	4.2
7	114	3.5
6	112	3.1
5	217	2.3
4	335	2.2
3	515	1.5
2	923	1.2
1	1953	0.6
0	5307	0.3

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 well, we believe that it is an optimal candidate for statistically modeling the introduction and the appearance of Bugs in the modules of a software system in continuous growth and evolution.

5.3 Related work

5.3.1 Metric Correlation

Product metrics extracted by analysing static code of software, have been used to build models that relate these metrics to failure-proneness [66, 98, 26, 12, 51]. Among these, the CK [24] suite is historically the most adopted and validated for analysing bug-proneness of software systems [98, 26, 12, 51]. The CK suite was adopted by practitioners [26] and is also incorporated into several industrial software development tools. It is for this reason that we decided to adopt CK metric in our analysis.
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 were correlated to defect density. Considering industry data from software developed in C++ and Java, Subramanyam and Krishnan [98] showed that CK metrics were significantly associated with defects. Among others, Gimothy et al. [51], in a study of an Open Source system, validated the usefulness of these metrics for fault-proneness prediction.
The purpose of CK suite is to measure proprieties like coupling and cohesion of classes in OO software contexts. Unfortunately this suite not take into account the amount of “infor-

mation" passing through a given module of the software network. To overcome this problem, the Social Network Analysis (SNA) can be adopted; these subject in fact offers 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 [113] showed that network measures derived from dependency graphs are able to identify critical binaries of a complex system missed by complexity metrics. However, their results are obtained considering only one industrial product (Windows Server 2003). Tosun et al. [101] reproduced the previous work [113] extending the network analysis 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 for small scale projects.

Both previous studies [113, 101] did not consider mutual relationships among SNA metrics and complexity metrics; therefore they did not show if SNA metrics carried new information with respect to the CK suite. Our work, instead, computes the correlation matrix among SNA Metrics and CK metrics, considering also mutual correlations with respect to Issue, Bug, Loc, Fan-out and Fan-in. In addition it must be noticed that [113], and barely [101], do not perform a study of the software system during its evolution. In our study we analyse how metrics correlations change along different releases during the software its life-cycle. Our research suggests which results can be generalized, or maybe project or release dependent. For all these reasons our work sheds new light on SNA for software systems.

5.3.2 Distribution fitting

Recent work 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[11] [44, 5, 111]. This is a sub-problem of the more general issue of a statistical description of different properties of large software systems. 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) [107, 29, 14]. These findings follow a more general Pareto 80-20 principle [78].
Fenton and Ohlsson [44] verified the Pareto principle for defect distribution in packages, and this suggests that the defect distribution may also satisfy a Power-Law, although there is no evidence for this in literature. Their finding was recently confirmed by Andersson and Runeson [5]. 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 [53]. In a recent paper, Zhang [111] pointed-out that fitting the Alberg diagram of bugs over modules[12] 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 [5]. He also 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.
This work clarifies some issues related to the distribution of bugs taking into account two representative Open Source Systems. In contrast to previous work, we compare key distri-

[11] 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.

[12] Note that the term "module" has a broad meaning and can be considered as a synonym of basic software components such as class, file a packages and so on.

butions used in literature to model software properties. We introduce the generative models able to produce such data, and apply these models to exploit the mechanism of bug introduction in software modules. We 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. To our knowledge, no research has performed such a deep analysis on bug distribution. Although our results have been obtained on Eclipse and Netbeans, we believe these models, in particular the Yule-Simon, can generalize to other software systems.

Chapter 6

Threats to validity

This work performs an empirical study on the statistical properties related to bug occurrence, in Object-Oriented (OO) systems. This chapter faces the most important threats to the validity related to our study, namely data mining of software bugs from CVS repository and generalization of our findings.

6.1 Threats to internal validity

Threats to internal validity concern each confounding factor that can influence obtained results. To posses Internal validity the experiment has to exhibit a causal relation between two variables, such as the treatment and the experiment's result [94]. Dealing with software Bugs, one of the most important concerns is the correct mapping between modules (i.e. files) and bugs. This assignment is strongly dependent on the information tracked on CVS and BTS, and then on the way in which developers report their maintenance activities [7, 15]. During bug mapping it is possible to have:

- a false positive when a bug should not have been mapped with a module
- a false negative when a bug should have been mapped with a module (but was not)

Both eventualities may perturb bug association. False positive and false negative can stem from these scenarios:

1. A developer marks an enhancements as a bug on BTS (false positive). This occurrence can happens sometimes[1] and it modifies the number of bugs detected. This problem is not solvable by any system which rely on BTS information.
2. A sequence number, such as for example a date or a release, is wrongly considered as a valid bug (false positive).
3. A valid sequence number which reports a bug, is not considered as such. This could happen in project where BTS and CVS are badly synchronized and a bug identification number is not reported in both system (false negative).

[1] We have never found this evidence, but we cannot exclude this possibility in systems which use BTS [6]

4. A bug fixing on a file may be done on files which are not related to the bug. For example a developer not able to bug fix a files belonging to third-party library, can overcame this problem through a "workaround" on other files accessible for them. As result the bug is mapped to a different file (false positive and negative[2]) .

Eclipse and Netbeans are projects where commit reports are disciplined and consistent; we believe that scenarios 1 and 3 are scarce. In addition, even if they could be exhibited, they are rare and their effect in bug mapping is mitigated by the size of the analysed OS projects. To overcame possible problems determined by scenario 2 we proceed in two ways:

- by validating all bug identification numbers against the BTS
- avoiding the use of some interval number particularly prone to this phenomena.

We assessed the goodness of this procedure, checking manually the 10% of module-bug associations[3] without finding any error.
To address problems raised by scenario 4, it is required more information than "is available or automatically inferable" as assessed by Purushothaman and Perry[84]. Even though is not always true that bug fixing occurs only on modified modules, this assumption is often adopted in the literature [44, 71, 76, 80].

6.2 Threats to external validity

Threats to external validity concern the degree to which we can draw general conclusions from our results. To be external valid the experiment's results have to be held across different procedures, participants and experimental settings. Having this attribute, the findings of study can be generalized to environment different from the considered experiment [94]. However - according to Basili et al. - software engineering processes depend on a large number of relevant context variables that complicate the extension of conclusions. Beyond the specific environment where the analysis is performed, it is impossible to consider the conclusion drawn for an empirical study still valid [13]; as El Emam states "Only when evidence is accumulated that a particular metric is valid across systems and across organizations can we draw general conclusions" [39].
This study is performed on Eclipse and Netbeans, both large Open Source projects with many similarities to projects developed in industry like the use of change management systems, extensive test cases, and descriptive commit messages. Moreover, some studies have suggested that there is little difference between open-source and closed-source software [67]. The commonality we have shown from both projects and across all releases allow us to consider that our results hold in industry Java systems of same dimension, or at last for other OS project. On the other hand, it also possible that our findings would not apply to other OS project, even if developed in Java. In fact, both OS projects analysed could not be representative of all development contexts, and hence our results may not be extendible. Commercial systems deal with different deadline pressures, personnel turnover patterns, different development processes and can have a different bug occurrence. Finally, it is a worth noticing that the considered software was developed in Java. Other languages without the Java static

[2] The bug should be detached from a module(s) and mapped to another one.
[3] Such mappings were randomly chosen.

type checking, i.e Smalltalk, may exhibit differences.
We will extend our research for other products. Replications of this work will provide greater clarity on these topic.

6.3 Threats to construct validity

Threats to construct validity are focused on how accurately the observations describe the phenomena of interest. This study deals with static analysis of software metrics such as the CK suite, SNA metrics, Bug and Issue. Since all of them are measured directly, no threats to the construct validity are introduced.

Chapter 7

Conclusions

In this dissertation we have adopted a graph, based on the Complex Network theory, to describe object oriented software systems. In this graph, modules represent nodes, and edges describe relationships among nodes. We extracted Chidamber and Kemerer (CK), Social Network, and other graph-related metrics. Using this representation we analysed, during software evolution, the Bugs and Issues occurrence of two case studies: Eclipse and Netbeans. The following are contributions of this dissertation:

1. We analysed how software Bugs were related to software network metrics, and how they evolved during the software life-cycle. We correlated Bugs with respect to traditional metrics, such us the CK suite, metrics derived from SNA and other related to software graph. We showed that these complex network metrics are significantly correlated with software Bugs. Correlations among SNA metrics and Bugs are general comparable with CK metrics. The SNA metrics highlight how the social role of a node influences its Bug proneness; the EGO metrics are in fact correlated most with Bugs. Central nodes of the software network were generally more Bug prone or indirectly involved in bug fixing with respect to peripheral nodes. All these reasons make SNA metrics worthy of study in deeper detail, not as an alternative to more traditional OO metrics, but in their conjunction.

2. We characterized the Issue occurrence phenomenon in Eclipse and Netbeans projects, analysing how maintainers fix Issues during the software development process. We perform a study of the software product release by releases. We found that Eclipse, had a foreseeable evolution of its average number of Issues, whereas Netbeans behaviour was less foreseeable.

3. We discussed the statistical distributions of Bugs affecting large software systems, roughly described as the 80-20 Pareto principle. Our work took into account conventional wisdom about bug distribution across software modules, showing the strengths and weaknesses of the proposed models.

    - We found that Bug distribution presented a persistent statistical behaviour across all the system releases, and showed a similar trend in both Eclipse and Netbeans systems.

- We went further by discussing the generative models able to justify the Bug distributions. This has been achieved by adopting statistical distributions, such as Weibull, Log Normal, Double Pareto and Yule-Simon, already successfully employed in the literature to describe general properties of large software systems. In addition for the Yule-Simon distribution we tested the applicability of the growth model.
  The goodness of fit of previous distributions were evaluated numerically comparing the R^2 determination coefficient and visually confronting the log-log plots, which enhanced the discrepancy in the tail of the distributions. We verified that all distributions had good fits with empirical data. However, among them Yule-Simon outperforms the other distributions, since it describes the shapes of Bug distributions taking into account also modules with no bugs.
  Our analysis showed that new Bugs were introduced, on average, in larger amounts in modules “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 model is instead related to a system made of components which fail with a given rate in time. It cannot account for the empirical evidence of Bugs (failures) added to modules with specific statistical rules, typical of complex systems, whose presence is suggested by the fat-tail in the bug distribution.
  We hope this work will favour replication on different case studies, to discover to what extent Yule-Simon model, or even other models, might be adopted for Bug distribution analysis in large object-oriented systems.

We contributed to the body of empirical and theoretical research on software fault behaviour in large complex systems. From the perspective of software practitioners, our results can be exploited to keep under control the quality of software systems. In fact, a statistical model able to estimate the future rate of Bugs in classes of modules could help management to organize pre- and post-release testing.

Bibliography

[1] L.A. Adamic, "Zipf, power-law, pareto - a ranking tutorial", 2000. [Online]. Available: http://www.hpl.hp.com/research/idl/papers/ranking/ [cited at p. 28]

[2] R. Albert, H. Jeong and A.-L. Barabási, "Diameter of the world wide web". Nature, 401:130-131, 1999. [cited at p. 15]

[3] L.A.N. Amaral, A. Scala, M. BarthÍmy and H.E. Stanley, "Classes of small-world networks", in Proc. Natl. Acad. Sci. U.S.A. 97 (21), pp. 11149-11152, 2000 [cited at p. 15]

[4] L.A.N. Amaral and J.M. Ottino, "Complex networks".European Physical Journal B, 38:147-162, 2004. [cited at p. 15]

[5] C. Andersson and P. Runeson, "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems", IEEE Transaction on Software Engineering, vol. 33, no. 5, pp. 273-286, May 2007 [cited at p. 28, 46]

[6] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests", in Proc. of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, October 27-30, 2008, Ontario, Canada [cited at p. 49]

[7] K. Ayari, P. Meshkinfam , G. Antoniol and M. Di Penta, "Threats on building models from CVS and Bugzilla repositories: the Mozilla case study", in Proc. of the 2007 conference of the center for advanced studies on Collaborative research, October 22-25, 2007, Richmond Hill, Ontario, Canada [cited at p. 25, 49]

[8] A.-L. Barabási, "Linked: How Everything Is Connected to Everything Else and What It Means". Plume, 2002. [cited at p. 15]

[9] A.-L. Barabási and E. Bonabeau, "Scale-free networks", Scientific American,288:60-69, 2003. [cited at p. 15]

[10] A.-L. Barabási and Z.N. Oltvai, "Network Biology: Understanding the cell's function organization", Nature Reviews - Genetics, 5:101-113, February 2004. [cited at p. 15]

[11] A.-L. Barabási and R. Albert. "Emergence of scaling in random networks". Science, 286:509-512, 1997. [cited at p. 15]

[12] V.R. Basili, L.C. Briand and W.L. Melo, "A validation of object-oriented design metrics as quality indicators", IEEE Transaction on Software Engineering, 22(10) pp. 751-61, 1996 [cited at p. 26, 45]

[13] V.R. Basili, F. Shull and F. Lanubile, "Building Knowledge through Families of Experiments", IEEE Transaction on Software Engineering, vol. 25, no. 4, pp. 456-473, July/Aug. 1999. [cited at p. 50]

[14] 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. 16, 19, 46]

[15] C. Bird, A. Bachmann, E Aune, J. Duffy, A. Bernstein, V. Filkov and P. Devanbu, "Fair and Balanced? Bias in bug-fix Datasets", in European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE),2009 [cited at p. 49]

[16] B. Boehm, "Software engineering" IEEE Transactions on Computers, C-25(12): 1226-1241, December 1976. [cited at p. 1]

[17] B. Bollobás, "Random graphs", Academic Press, Inc., 1985. [cited at p. 15]

[18] L. Briand, J. Daly and J.K. Wüst, (1999), "A Unified Framework for Coupling Measurement in Object-Oriented Systems", IEEE Transactions on Software Engineering, 25(1), 91-121. [cited at p. 7]

[19] L. Briand, L. Daly and J. Wuest, (1998). "A Unified Framework for Cohesion Measurement in Object-Oriented Systems", Empirical Software Engineering: An International Journal, 3, 65-117. [cited at p. 7]

[20] F. Brito e Abreu and W. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality", 3rd International Metric Symposium, pp. 90-99, 1996. [cited at p. 8]

[21] F.P. Brooks, "The Mythical Man-Month", Addison-Wesley, 1995 [cited at p. 15]

[22] M. Bunge, "Treatise on Basic Philosophy: Ontology II: the World of Systems", Riedel, Boston, MA, 1979 [cited at p. 8]

[23] R. Burt, "Structural Holes: The Social Structure of Competition", Cambridge, MA: Harvard University Press, 1995. [cited at p. 12]

[24] S.R. Chidamber and C.F. Kemerer, "Towards a Metrics Suite for Object Oriented Design", in Proc. Conf. Object Oriented Programming Systems, Languages, and Applications (OOPSLA 91), vol. 26, no. 11, pp. 197-211, 1991. [cited at p. 45]

[25] S.R. Chidamber, D.P. Darcy and C.F. Kemerer, "Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis, IEEE Transaction on Software Engineering, 24(8) pp. 629-39, 1994 [cited at p. 8, 26]

[26] S.R. Chidamber, D.P. Darcy and C.F. Kemerer Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis. IEEE Transaction on Software Engineering, 24 (1998) 629-639. [cited at p. 45]

[27] N.I. Churcher and M.J. Shepperd, Comment on - "A Metric Suite for Object Oriented Design", Correspondence of IEEE Transaction on Software Engineering, 21(3) pp. 263-65, 1995 [cited at p. 10]

[28] A. Clauset, C.R. Shalizi and M. Newman, "Power-Law Distributions in Empirical Data", SIAM Review, v.51 n.4, p.661-703, November 2009 [cited at p. 39]

[29] 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, 370:817-831, Oct. 2006. [cited at p. 19, 46]

[30] G. Concas, M. Marchesi, S. Pinna and N. Serra, "Power-laws in a large object-oriented software system", IEEE Transaction on Software Engineering, 33(10):687-708, Oct. 2007. [cited at p. 15, 17]

[31] G. Concas, M. Marchesi, A. Murgia, S. Pinna and R. Tonelli, "Assessing Traditional and New Metrics for Object-Oriented Systems", in Proc. Workshop on Emerging Trends in Software Metrics - WETSoM, ICSE 2010, 4 May 2010, Capetown (SA) (2010). [cited at p. 23, 27]

[32] G. Concas, M. Marchesi, A. Murgia and R. Tonelli, "An Empirical Study of Social Networks Metrics in Object Oriented Software", Advances in Software Engineering. 2010 [cited at p. 21, 29]

[33] G. Concas, M. Marchesi, A. Murgia, R. Tonelli and I. Turnu, "On the distribution of bugs in the Eclipse system", IEEE Transaction on Software Engineering, (in press) 2011. [cited at p. 21, 23, 29]

[34] D. Cubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts", in Proc. Conference on Software Engineering, pages 408-418, 2003. [cited at p. 25]

[35] T. DeMarco, "Controlling Software Projects", Yourdon Press Inc., New York, 1978. [cited at p. 6]

[36] S.N. Dorogovtsev and J.F.F. Mendes, "Evolution of Networks" - From Biological Nets to the Internet and WWW. Oxford University Press, 2003. [cited at p. 15]

[37] M. Eaddy, T. Zimmermann, K.D. Sherwood, V. Garg and G.C. Murphy, N. Nagappan and A. V. Aho, "Do Crosscutting Concerns Cause Defects?", IEEE Transaction on Software Engineering 26 (2008) 786-796. [cited at p. 6, 23]

[38] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron and A. Mockus, (2001). "Does code decay? Assessing the evidence from change management data", IEEE Transactions on Software Engineering, 27(1), 1-12. [cited at p. 1]

[39] K. El Emam, "A Methodology for Validating Software Product Metrics", Technical Report NRC 44142, National Research Council of Canada, 2000. [cited at p. 50]

[40] P. Erdös and A. Rényi, "On random graphs", Publicationes Mathematicae, 6:290-297, 1959. [cited at p. 15]

[41] P. Erdös and A. Rényi, "On the evolution of random graphs", Publication of the Mathematical Institute of the Hungarian Academy of Sciences, 5:17-61, 1960. [cited at p. 15]

[42] P. Erdös and A. Rényi, "On the strenght of connectedness of a random graph", Acta Mathematica Scientia Hungary, 12:261-267, 1961. [cited at p. 15]

[43] L. Erlikh, "Leveraging legacy system dollars for e-business", IT Professional, 2(3): 17-23, 2000. [cited at p. 1]

[44] N.E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system", IEEE Transactions on Software Engineering, vol. 26, pp. 797 - 814, 2000. [cited at p. 28, 46, 50]

[45] N.E. Fenton and S.L. Pfleeger, "Software Metrics (2nd ed.), a Rigorous and Practical Approach", PWS Publishing Company, 1997 [cited at p. 5, 6]

[46] M. Fischer, M. Pinzger and H. Gall, "Analyzing and relating bug report data for feature tracking", in Proc. Working Conference on Reverse Engineering, 2003. [cited at p. 25]

[47] M. Fischer, M. Pinzger and H. Gall, "Populating a release history database from version control and bug tracking systems", in Proc. International Conference on Software Maintenance, 2003. [cited at p. 25]

[48] P.J. Flory, "Molecular size distribution in three-dimensional polymers. i. gelation". Journal of the American Chemical Society, 63:3083-3090, 1941. [cited at p. 15]

[49] S. Focardi, M. Marchesi and G. Succi, "Extreme Programming Examined", Addison-Wesley, pp. 191-206, 2000 [cited at p. 2]

[50] T. Furuyama, Y. Arai and K. Iio (1997), "Analysis of Fault Generation Caused by Stress During Software Development", Journal of Systems and Software, 38, 13-25. [cited at p. 7]

[51] T. Gymóthy, R. Ferenc and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction", IEEE Transaction on Software Engineering 31 (2005) 897-910. [cited at p. 45]

[52] R. A. Hanneman and M. Riddle, "Introduction to social network methods. Riverside", CA: University of California, Riverside 2005. [cited at p. 11]

[53] L. Hatton, "Power-law distributions of component size in general software systems". IEEE Transactions on Software Engineering, 35(4) IEEE, July, pp. 566-572 (2009). [cited at p. 46]

[54] B. Hayes, "Graph Theory in Practice: Part I", American Scientist, 88(1):9-13,2000. [cited at p. 15]

[55] B. Hayes, "Graph theory in practice: Part II", American Scientist, 88(2):104-109, 2000. [cited at p. 15]

[56] B. Henderson-Sellers, L.L. Constantine and I.M. Graham, "Coupling and cohesion: towards a valid metrics suite for object-oriented analysis and design, Object Oriented Systems, 3(3), pp.143-58, 1996 [cited at p. 10]

[57] E. Hill, L. Pollock and K. Vijay-Shanker, "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Automated Software Engineering (ASE), Atlanta, Georgia, USA, pp. 14-23, November 5-9 2007. [cited at p. 1]

[58] M. Kajko-Mattsson, "Common concept apparatus within corrective software maintenance", in Proc. of 15th IEEE International Conference on Software Maintenance (ICSM'99), IEEE Press, pp 287-296. 1999. [cited at p. 5]

[59] M.M. Lehman (1980). "Programs, life cycles and laws of software evolution", in Proc. of the IEEE, 68(9), 1060-1076. [cited at p. 1]

[60] W. Li, S. Henry, "Object-Oriented Metrics that Predict Maintainability", Journal of Systems and Software, vol. 23, no. 2, pp. 111-122, 1993. [cited at p. 8, 26]

[61] W. Li, "Another metric suite for object-oriented programming", Journal of Systems and Software, 44:155.162, 1998. [cited at p. 6]

[62] B.P. Lientz and E.B. Swanson, "Software Maintenance Management", Reading, Massachusetts: Addison-Wesley, 1980. [cited at p. 1]

[63] M. Lorenz and J. Kidd, "Object-Oriented Software Metrics", Prentice Hall Inc., 1994. [cited at p. 6, 8]

[64] P. Louridas, D. Spinellis and V. Vlachos, "Power laws in software". ACM Transactions on Software Engineering and Methodology. 18, 1, Article 2 (September 2008), 26 pages. [cited at p. 16, 17, 19, 36]

[65] M. Marchesi, S. Pinna, N. Serra and S. Tuveri, "Power Laws in Smalltalk", Twelfth Smalltalk Joint Event, Koethen, Germany, 2004 [cited at p. 2, 16]

[66] T. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol.2, No.4, 308-320. 1976 [cited at p. 45]

[67] H. Melton and E. Tempero, "An empirical study of cycles among classes in Java", Technical Report UoA-SE-2006-1, Department of Computer Science, University of Auckland, 2006. [cited at p. 50]

[68] S. Milgran, "The small world problem", Psychology Today, 1(1):60-67, 1967. [cited at p. 15]

[69] M. Mitzenmacher, "Dynamic Models for File Sizes and Double Pareto Distributions", Internet Mathematics, Vol. 1, No. 3: 305-333, 2003. [cited at p. 18, 37, 43]

[70] P. Mohagheghi, R. Conradi and J.A. Borretzen, "Revisiting the problem of using problem reports for quality assessment", in Proc. of the International Workshop on Software Quality, pp. 45-50, 2006. [cited at p. 2, 5]

[71] K.-H. Möller and D. J. Paulish, "An empirical investigation of software fault distribution", in International Software Metrics Symposium, 1993, pp. 82-90. [cited at p. 50]

[72] A. Murgia, G. Concas, M. Marchesi and R. Tonelli, "A machine learning approach for text categorization of fixing-issue commits on CVS", in Empirical Software Engineering and Measurement 10, September 16-17 2010, Bolzano-Bozen, Italy [cited at p. 23, 26]

[73] A. Murgia, G. Concas, S. Pinna, R. Tonelli and I. Turnu, "Empirical study of software quality evolution in open source projects using agile practices", in Emerging Trends in Software Metrics, Pula, Italy, May 2009. [cited at p. 21, 23]

[74] A. Murgia, G. Concas, M. Marchesi, R. Tonelli and I. Turnu, "An Analysis of Bug Distribution in Object Oriented Systems", in Emerging Trends in Software Metrics, Pula, Italy, May 2009. [cited at p. 21, 23]

[75] C.R. Myers, "Software systems as complex networks: structure, function and evolvability of software collaboration graphs", Physical Review E 68, 046116, 2003 [cited at p. 2, 16]

[76] N. Nagappan, T. Ball and A. Zeller, "Mining metrics to predict component failures", in International Conference on Software Engineering, 2006, pp. 452-461. [cited at p. 50]

[77] M.E.J. Newman, "Structure and function of complex networks", SIAM Review, 45(2):167-256, 2003. [cited at p. 11, 15]

[78] M.E.J. Newman, "Power laws, Pareto, distributions and Zipf's law", Contemporary Physics, Vol. 46, pp. 323-351, 2005. [cited at p. 17, 19, 20, 28, 46]

[79] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches", IEEE Transactions on Software Engineering archive Volume 22 Issue 12, December 1996 [cited at p. 28]

[80] T. Ostrand, E. Weyuker and R. M. Bell, "Predicting the location and number of faults in large software systems", IEEE Transactions on Software Engineering, vol. 31, pp. 340 - 355 2005. [cited at p. 50]

[81] D.L. Parnas, (1994). "Software aging", in Proc. of the 16th International Conference on Software Engineering, pages 279-287. IEEE Computer Society. [cited at p. 1]

[82] R. Pastor-Satorras and A.Vespigani, "Epidemic Spreading in Scale-Free Network", Physical Review Letters, 86(14):3200-3203, April 2001. [cited at p. 15]

[83] A. Potanin, J. Noble, M. Frean and R. Biddle "Scale-Free Geometry in Object-Oriented Programming", Communications of the ACM, v.48 n.5, p.99-103, May 2005 [cited at p. 2, 15, 16]

[84] R. Purushothaman and D.E. Perry, "Toward Understanding the Rhetoric of Small Source Changes", IEEE Transaction on Software Engineering, vol. 31, no. 6, pp. 511-526, June 2005. [cited at p. 50]

[85] A. Rapoport, "Nets with distance bias", Bulletin of Mathematical Biophysics, 13:85-91, 1951. [cited at p. 15]

[86] A. Rapoport, "Spread of information through a population with sociostructural bias: I. assumption of transitivity", Bulletin of Mathematical Biophysics, 15:523-533, 1953. [cited at p. 15]

[87] A. Rapoport, "Contribution to the theory of random and biased nets", Bulletin of Mathematical Biophysics, 19:257-277, 1957. [cited at p. 15]

[88] S. Redner, "How popular is your paper? An empirical study of the citation distribution", The European Physical Journal B, 4(2):131-134, 1998. [cited at p. 15]

[89] 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 Nature", Physical Review E, vol. 66 (2002), 67-103. [cited at p. 18]

[90] W.J. Reed, "The Pareto Law of Incomes - An Explanation and an Extension", Physica A, Vol. 319, 469-485. 2003. [cited at p. 18]

[91] 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. 18]

[92] G. Sabidussi, "The centrality index of a graph", Psychometrika, vol. 31, pp. 581-603, 1966. [cited at p. 13]

[93] J.P. Scott, "Social Network Analysis Sociology", Vol. 22, No. 1, 109-127 (1988) [cited at p. 2]

[94] W. Shadish, T. Cook and D. Campbell, "Experimental and Quasi-Experimental Designs for Generalized Causal Inference", Houghton-Mifflin, 2002. [cited at p. 49, 50]

[95] H.A. Simon, "On a class of skew distribution functions", Biometrika 42 (1955) 425-440. [cited at p. 19]

[96] J. Śliwerski, T. Zimmermann and A. Zeller, "When do changes induce fixes?", in Proc. International Workshop on Mining Software Repositories, 2005. [cited at p. 23, 25]

[97] C.P. Stark and N. Hovius, "The characterization of landslide size distributions", Geophysical Research Letters, Vol. 28, No. 6, pp. 1091-1094, March 15, 2001. [cited at p. 18]

[98] R. Subramanyam and M. Krishan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects", IEEE Transactions on Software Engineering, 29 (2003) 297-310. [cited at p. 45]

[99] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing", National Institute of Standards and Technology 2002. [cited at p. 1]

[100] R. Tonelli, G. Concas, M. Marchesi and A. Murgia, "An Analysis of SNA metrics on the Java Qualitas Corpus", in Proc. of India Software Engineering Conference - ISEC 2011. [cited at p. 27]

[101] A. Tosun, B. Turhan and A. Bener, "Validation of Network Measures as Indicators of Defective Modules in Software Systems", in Proc. of the 1st International Conference on Predictor Models (PROMISE), 2009. [cited at p. 11, 26, 27, 46]

[102] S. Valverde, R.F. Cancho and R.V. Solé, "Scale-Free networks from optimal design", Eur. Phys. Letters 60, pp. 512-517, 2002 [cited at p. 2, 15, 16]

[103] S. Valverde and R.V. Solé, "Hierarchical small worlds in Software Architecture", URL: http://arxiv.org/abs/cond-mat/0307278 [cited at p. 2, 16]

[104] S. Valverde and R.V. Solé, "Network motifs in computational graphs: A case study in software architecture", Physical Review E 72 (2005) 026107. [cited at p. 16]

[105] D.J. Watts and S.H. Strogatz, "Collective dynamics of small-world networks", Nature, 393(6684):440-442, 1998. [cited at p. 15]

[106] C. Wetherell, A. Plakans and B. Wellman, "Social networks, kinship and community in Eastern Europe", Journal of Interdisciplinary History 24 (1994) 639-663. [cited at p. 10]

[107] R. Wheeldon and S. Counsell, "Power law distributions in class relationships", in Proc. Third IEEE Int. Workshop on Source Code Analysis & Manipulation, Amsterdam, The Netherland, 2003 [cited at p. 2, 16, 18, 46]

[108] 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 Transaction on Software Engineering, vol. 10, no. 4, pp. 376-383, 1984. [cited at p. 20]

[109] K. Wong, S.R. Tilley, H.A. Müller and M.-A.D. Storey, "Structural Redocumentation: A Case Study", IEEE Software, 12(1): 46-54, January 1995. [cited at p. 1]

[110] G.U. Yule, "A mathematical theory of evolution based on the conclusions of dr. J. C. Willis", Philosophical Transactions of the Royal Society of London B 213 (1925) 21-87. [cited at p. 19]

[111] H. Zhang, "On the Distribution of Software Faults", IEEE Transaction on Software Engineering, Vol. 34, No. 2, March/April 2008. [cited at p. 20, 28, 42, 46]

[112] T. Zimmermann, R. Premraj and A. Zeller, "Predicting defects for eclipse", in Proc. of the International Workshop on Predictor Models in Software Engineering, 2007. [cited at p. 25]

[113] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs", in Proc. of the 30th international conference on Software engineering, May 10-18, 2008, Leipzig, Germany. [cited at p. 11, 16, 26, 27, 46]

List of Publications Related to the Thesis

Published papers

Journal papers

- G. Concas, M. Marchesi, A. Murgia and R. Tonelli *An Empirical Study of Social Networks Metrics in Object Oriented Software* in *Advances in Software Engineering*, 2010.
- G. Concas, M. Marchesi, A. Murgia, R. Tonelli, and I. Turnu, *On the distribution of bugs in the Eclipse system* in *IEEE TRANS. SW. ENG.*, (in press) 2011.

Conference papers

- A. Murgia, G. Concas, S. Pinna, R. Tonelli, I. Turnu *Empirical study of software quality evolution in open source projects using agile practices* in *Emerging Trends in Software Metrics*, Pula, Italy, May 2009.
- A. Murgia, G. Concas, M. Marchesi, R. Tonelli and Ivana Turnu *An Analysis of Bug Distribution in Object Oriented Systems* in *Emerging Trends in Software Metrics*, Pula, Italy, May 2009.
- A. Murgia, Giulio Concas, Michele Marchesi and Roberto Tonelli *A machine learning approach for text categorization of fixing-issue commits on CVS* in *Empirical Software Engineering and Measurement 10*, September 16-17 2010, Bolzano-Bozen, Italy
- G. Concas, M. Marchesi, A. Murgia, S. Pinna, and R. Tonelli *Assessing Traditional and New Metrics for Object-Oriented Systems.* In Proc. *Workshop on Emerging Trends in Software Metrics - WETSoM*, ICSE 2010, 4 May 2010, Capetown (SA) (2010).
- R. Tonelli, G. Concas, M. Marchesi, A. Murgia *An Analysis of SNA metrics on the Java Qualitas Corpus.* In Proc. *India Software Engineering Conference - ISEC*, 23-27 Feb 2011, Thiruvananthapuram, Kerala (India).

Appendices

Appendix A

Extra Data

This section collects data not presented in result chapter. Table A.1 report Spearman correlations of Issue and Bug respect to LOC, CK metrics, Social Network metrics and other graph-related metrics. Figures A.1 - A.4 shows the CCDF of Bugs in several release of Netbeans and Eclipse 3.3. All of them report the original data along with the best fit functions.

Table A.1:

Spearman correlation among metrics

	Eclipse								Netbeans									
	2.1		3.0		3.1		3.3		3.2		3.3		3.4		4.0		6.0	
	Issue	Bug	Issue	Bug	Issue	Bug	Issue	Bug	Issue	Bug	Issue	Bug	Issue	Bug	Issue	Bug	Issue	Bug
Bug	95%	-	95%	-	95	-	92%	-	98%	-	93%	-	91%	-	86%	-	88%	-
LOCs	46%	46%	54%	52	51	50%	38	38	47%	46%	49%	47%	33%	29%	30%	39	35%	35%
WMC	38%	38%	46%	45%	40%	38%	30%	29%	44%	42%	44%	41%	30%	26%	28%	34	27%	28%
RFC	46%	46%	49%	48%	43%	42%	32%	32%	44%	42%	47%	44%	31%	27%	29%	34	30%	31%
LCOM	34%	34%	43%	41%	35%	34%	28%	27%	41%	39%	41%	38%	28%	24%	25%	31%	22%	23%
CBO	45%	45%	45%	44%	44%	43%	32%	32%	33%	32%	38%	36%	28%	26%	24%	25%	28%	29%
Fanin	8%	7%	10%	10%	9%	8%	7%	6%	13%	12%	16%	15%	17%	15%	16%	14%	7%	8%
Fanout	44%	44%	49%	48%	46%	45%	35%	34%	45%	44%	49%	47%	33%	30%	38%	42%	34%	35%
Reach-Efficiency	16%	16%	18%	18%	14%	13%	6%	7%	17%	16	24%	22%	9%	5%	21%	25%	26%	24%
Eff-Size	41%	41%	45%	44%	43%	40%	33%	32%	38%	36%	41%	39%	31%	28%	37%	37%	28%	29%
Closeness	38%	39%	40%	39%	34%	32%	25%	24%	38%	36%	42%	38%	21%	16%	25%	30%	33%	30%
DwReach	40%	40%	42%	41%	36%	34%	25%	25%	37%	35%	43%	40%	23%	19%	32%	37%	37%	34%
InfoCentrality	-33%	-34%	-35%	-35%	-31%	-28%	-23%	-23%	-26%	-25%	-30%	-28%	-12%	-9%	-17%	-22%	-30%	-27%
Size	43%	42%	47%	46%	44%	42%	33%	33%	39%	38%	44%	41%	33%	29%	38%	38%	30%	31%
Ties	42%	42%	47%	46%	44%	42%	33%	32%	39%	37%	42%	40%	31%	28%	35%	36%	31%	31%
NWeak-Comp	-28%	-28%	-32%	-32%	-30%	-29%	-22%	-22%	-26%	-25%	-27%	-27%	-18%	-16%	-13%	-17%	-20%	-20%
Brokerage	42%	42%	46%	45%	43%	41%	33%	32%	38%	37%	40%	40%	32%	28%	38%	38%	29%	30%

All correlations are significant at the 0.01 level.

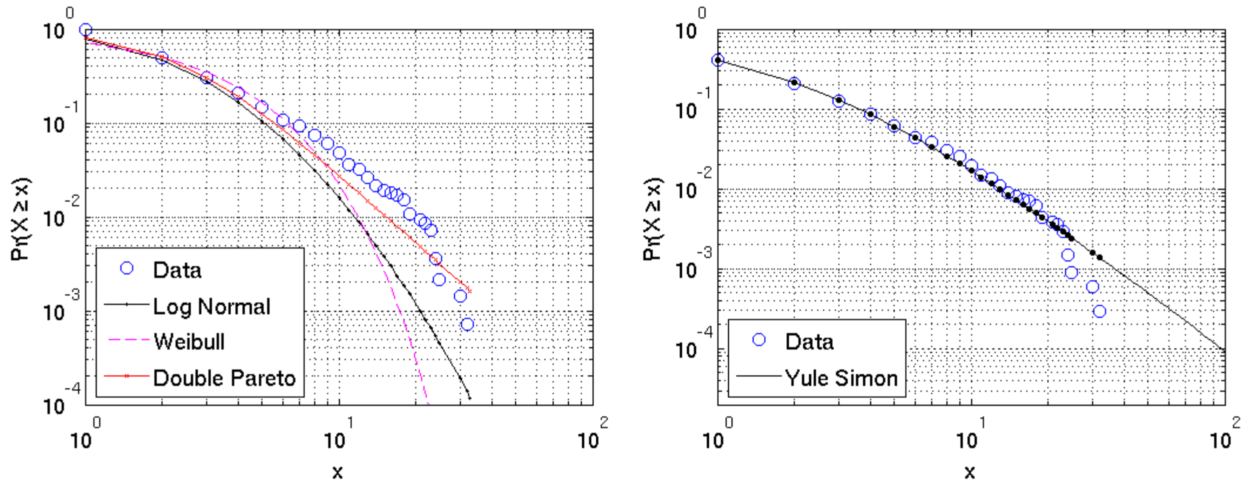


Figure A.1: The CCDF of Bugs in Netbeans 3.2.

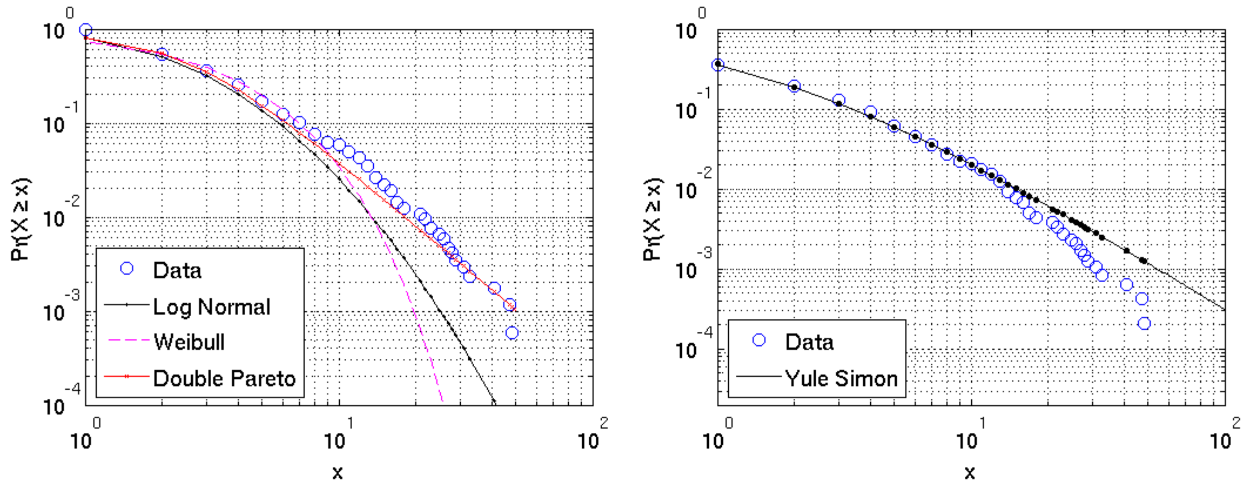


Figure A.2: The CCDF of Bugs in Netbeans 3.3.

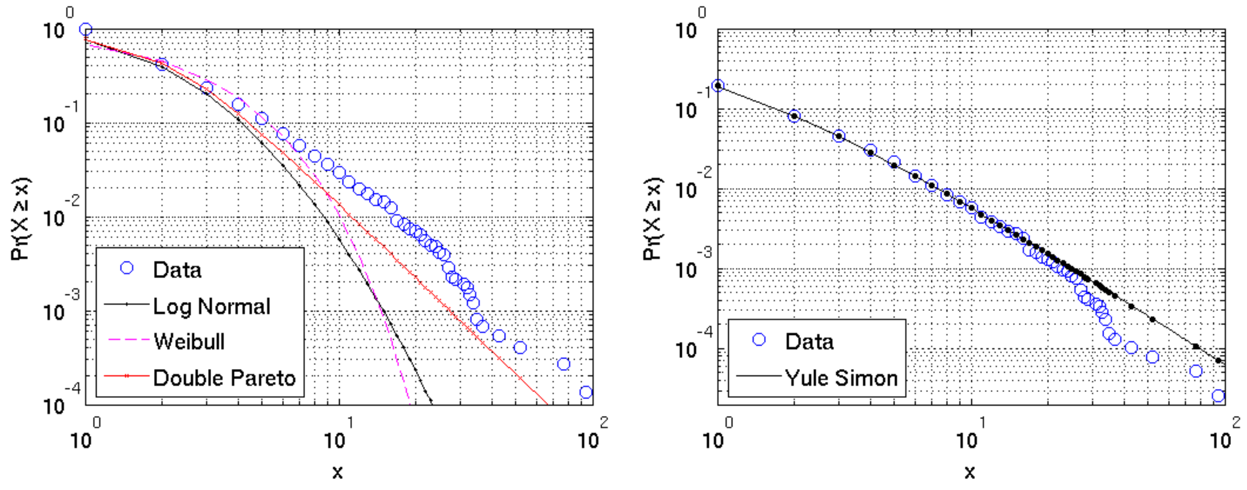


Figure A.3: The CCDF of Bugs in Netbeans 6.0.

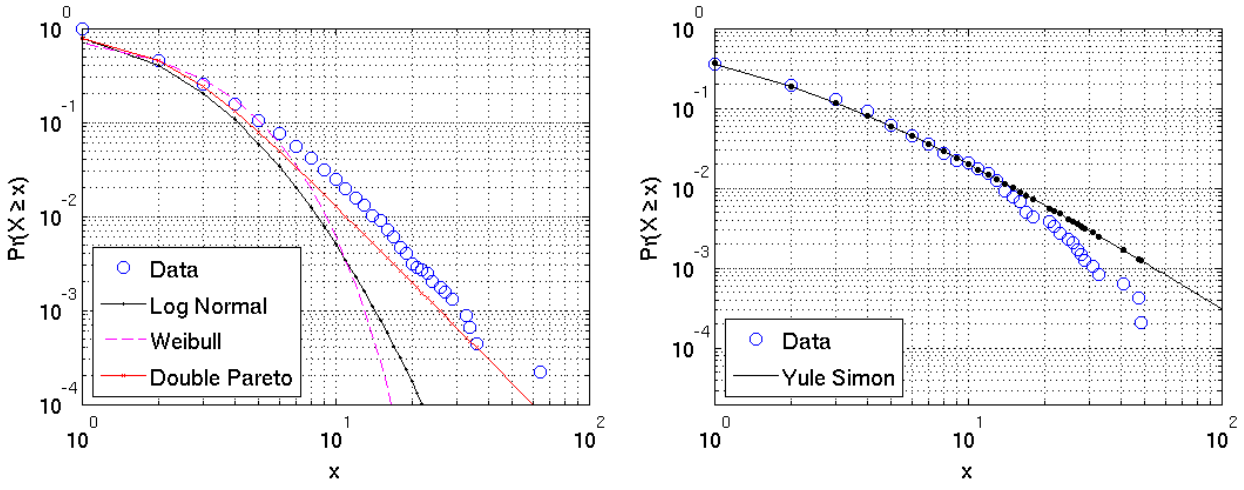


Figure A.4: The CCDF of Bugs in Eclipse 3.3.