Università degli Studi di Cagliari

## DOTTORATO DI RICERCA

Ingegneria Elettronica ed Informatica

Ciclo XXVIII

## TITOLO TESI

Study of Metrics and Practices for

Improving Object Oriented Software Quality

Settore scientifico disciplinari di afferenza

ING-INF/05

| | |
|---|---|
| Presentata da: | Matteo Orrù |
| Coordinatore Dottorato | Prof. Fabio Roli |
| Tutor | Prof. Michele Marchesi |

Esame finale anno accademico 2014 – 2015

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

# Study of Metrics and Practices for Improving Object Oriented Software Quality

Matteo Orrú

*Advisors*: Michele Marchesi and Roberto Tonelli
*Curriculum*: ING-INF/05 Informatica

Cycle XXVIII
2013-2015

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

# Study of Metrics and Practices for Improving Object Oriented Software Quality

Matteo Orrú

*Advisor*: Michele Marchesi and Roberto Tonelli
*Curriculum*: ING-INF/05 Informatica

Cycle XXVIII
2013-2015

*Dedicated to My Family*

*To the Memory of Giulio*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Software Engineering may be defined as the discipline that applies engineering principles, methods and practices to the production of software systems. Certainly one of its main goals is to produce software that meets the requirements with an advantageous trade-off between cost and benefits. Having a software system that does not present defects is certainly a must for both any software company and the final user.

However, pushed by an increasingly shortened time to market, companies usually are not able to completely test their programs to check for all the possible software defects. Software systems are then released when are reasonably fault-free but continuously maintained after the release. This process of continuous maintenance of software systems is needed not only to fix post-release bugs but also to adapt released programs to new requests coming from the users, adding new features and altering those already available to meet new requirements. In other words, software systems are ductile and adaptable artifacts, being explicitly designed to evolve according to the emerging needs of their users.

Modern software systems are usually large and complex, being composed by tens of thousands, or even millions of lines of code. Moreover, they are usually developed through a industrial process that involves dozens of people, with different roles (i.e. software engineers, team leaders, project or product managers) and it is influenced by several stakeholders, including final users. The inherent complexity of software systems, along with the need of providing new features without discarding the available product, imply that the size of software systems usually increases along the time, and the same happens with their complexity.

Additionally, software development is usually carried on in a distributed environment from developers that interact with each other using systems like Software Control Mangers (SCM) and Issue Tracking Systems (ITS). This emphasizes the complexity of the entire process.

In order to develop high quality software, software engineers need to take care of various aspects, spanning from development techniques to developers interactions. In order to control such kind of complexity it is required a system of measurements that can return useful information both on the product and on the process. Software metrics, came to help in this occasion, since it is practically impossible to control what you can not measure [33].Software metrics are used to measure many aspects of the software development: the product, the process, the resources, etc.

This thesis reports a collection of studies on software metrics that measure both product properties and the impact of some specific practices on software quality.To perform the reported research I applied a novel approach, based on the concept of complex network [11, 77].

As a matter of fact, software systems are composed by thousands of elementary modules, whose main goal is to execute simple tasks, that interact with each other exchanging data and sharing the computational burden, in order to carry on complex tasks.

Due to this inherent complexity, one of the best candidate to represent a software system is actually the complex network. In this model nodes are associated to the elementary modules of a software system, at different levels of granularity (i.e. functions, classes, files, etc.) whereas edges represent the various connections between the mentioned modules (i.e. function calls, inheritance relationships between classes, etc.)

The use of this model enables the researchers to investigate the topological properties of a software system, namely the reciprocal inter-relations between the single components, without neglecting other aspects related to how the single component is built. We called the metrics that exploits this topological representation *Network Metrics* The reported studies are focused mainly on some specific metrics, *Community Structure* and the related *Modularity* function, and *Clustering.* These metrics return an information about how much software modules (nodes) are connected to each other, representing a proxy measure for properties like *Coupling* and *Cohesion.* The former properties are related to the best practices and have been proven to be related to the quality of software systems [19].

This thesis is ideally divided in three parts. In the first chapters I present the related work and background information useful to better understand the remaining chapters (Chapters 2 and 3). In the second part, specifically in Chapters 4 and 5, I report the results of an analysis conducted on some popular software system, aimed at understanding the relationship between some network metrics and system's quality. In the third part I illustrate the results of a research aimed at quantify the impact of some development practice on software systems. Chapter 6 presents an analysis on the use of inheritance on Python software systems, along with the description of a curated collection of Python programs that I built in order to perform the mentioned research. Chapter 7 reports an investigation on the impact of Refactoring, a widely used practice of code restructuring, on the software topology. Finally, in Chapter 8, I draw some conclusions on the work carried on so far and I outline some ideas to improve the present research in the future.

## 1.1 Thesis Overview

The remaining of the present thesis is organized as follows:

- *Chapter 2* discusses the most significant research works in the field of complex networks, software networks and refactoring.

- *Chapter 3* presents the basic information useful to understand the rest of the thesis, including the approaches, algorithms and tools used.

- *Chapter 4* presents a case study performed on a release of a popular software system along with a longitudinal study on the evolution of the network topology of Eclipse, with specific regard to releases' defectiveness.

- *Chapter 5* reports and discusses the results of the analysis on the relationship between the clustering property of software networks and their defect proneness. The analysis has been performed on a number of releases of two popular open source software systems.

- *Chapter 6* contains an analysis of the use of inheritance in Python software systems, along with a description of the analyzed curated *corpus* of Python programs that I collected and publicly released to enable researchers to perform empirical research on Python systems.

- *Chapter 7* illustrates the results of a research on the impact of refactoring activities on the topology of a software system.

# Chapter 2

# Related Work

## 2.1 Complex Networks

Complex networks are complex systems consisting of a large number of elements connected together in a large number of different ways [81, 6]. The world around us is made in large extent by complex reticular systems, whether they belong to nature (i.e metabolic networks, food chains, etc.) or technology artifacts (i.e. computer networks, Internet, etc)[11].

A basic formal definition might be the following [42]. A complex network, likewise a graph, $G$ is a pair of sets $(V, E)$, where $V$ is a set of nodes (also called vertices in the proceeding) and E is a subset of $V^2$, the latter being the set of the unordered pairs of vertices. The elements of E are called edges. There are many properties of graphs, that may be present or not. For example, edges may be directed from a vertex to another. In this case we consider a directed graph (or network).

Complex networks are characterized by several properties that emerge along with the increase of their size. They usually present a power law distribution of the nodes' degree, meaning that the number of connections of the nodes can be described by the following law:

$$p_k \approx k^{-\alpha}$$

This kind of networks are called *scale-free*. Another common property of complex networks is the *small-world effect*. Discovered by Milgram [71] and popularize as “six degree of separation”, it determines that each pair of nodes is separated by a relative short path, namely a number of arcs that links the first and the second node, if compared to the size of the network.

Another meaningful property is the *community structure*. Inside a network, a community is a subgraph composed by nodes among which there are higher density of connections, if compared to nodes that are outside the community. The community structure is the specific way in which communities are arranged inside a network. According to the definition provided by Newman we can associate to a community structure a value of a benefit function called Modularity (Q) that represents how marked the community structure is.

Even software systems belong to the category of complex networks. The latter, in fact, are generally formed by elementary software modules, typically specialized and able to perform a specific task, which collaborate in different ways with other software modules to perform more complex tasks. Software engineering best practices prescribe to apply the *dividi et impera* (divide and conquer) principle and emphasizes the design of simple specialized modules to facilitate the reuse of code.

Software networks have been proven to be characterized by all the reported the properties, *small-world effect* [105], *scale-free* distribution of node degree [89] and *community structure* [104]. This thesis is partially devoted to the study of community structure in software networks.

## 2.2 Community Structure

The community structure is one of the properties that recently grabbed researchers' attention. Inside a network a community is a set of vertices between which there is a high density of connections. On the contrary between the communities connections are more sparse. The community structure of a network is its division in subgroups or communities [80]. Elements belonging to the same community are more likely to share the same behavior or properties, or represent a functional unit. This leads to many concrete applications of community detection in different fields (from marketing research to software development).

Community detection is traditionally addressed using techniques like hierarchical clustering and partitional clustering [42] and it faces different problems. One of the most important is the research of algorithms that allow network scalability up to different orders of magnitude (from thousands to millions of vertices) to be analyzed in a reasonable amount of time. Newman *et al.* proposed several algorithms for community detection [80, 79, 22] using different approaches and dealing with the problem of the computational burden. Moreover, in [80] Newman and Girvan introduced a quality function called *modularity*, to evaluate how good is a network partition in communities.

## 2.3 Software Networks and Metrics

Being software systems often large and complex, one of the best candidate to represent them is the complex network [41], [77], [101]. Many software networks, like class diagrams [89, 105], collaboration graph [77], package dependencies networks [30] have already been shown to present the typical properties of complex networks [11], such as fractal and self-similar features [102], scale free [18], and small world properties, and consequently power law distributions for the node degree, for the bugs [25] and for refactored classes [75].

Modeling a software system as a complex network has been shown to have many applications to the study of failures and defects. In [108] it has been shown that the knowledge of scale-free properties of software networks could be useful to reduce the time devoted to maintenance. Other methods have been applied to understand the relationship between number of defects and LOC, while in Ostrand et al. a negative binomial regression model is used to show that the majority of bugs is contained only in a fraction of the files (20%) [86].

So far many other methods have been tried for bug prediction [32, 55], especially using dependency graphs [82, 113], but only recently many researchers focused their attention on the community structure as defined in social network analysis, namely the division in subgroups of nodes among which there is a high density of connections if compared to nodes that are outside the community [80]. Being more connected, elements belonging to the same community might represent functional units or software modules, leading to practical applications of the community detection in the software engineering field. Community detection is usually performed with methods like hierarchical clustering and partitional clustering [42].

The issue of community structure and its application to software engineering has been recently addressed in a similar fashion by Šubelj and Bajek. The authors, after analyzing different Java softwares using a community detection algorithm, find that the software systems analyzed exhibit a significant community structure that doesn't match the packages structure imposed by the designer [104].

## 2.4 Refactoring

Refactoring was first formally described by William Opdyke in his Ph.D. dissertation [84] but it started gaining popularity in 1999, after Fowler defined his catalog containing information on when and how to do refactoring

[43]. After being introduced by Fowler, the code smells were also made recognizable in a book by Wake [106], while Simon et al. presented a generic approach for visualizing which classes need to be refactored [92]. Usually developers decide to apply refactoring by examining or changing the software code while they are performing other operations [76], such as bug fixing, addition of functionalities, or other code changes. For this reason, refactoring represents an important part of the software development cycle.

Previous studies claim that refactoring improves the quality of software [43], but they do not provide quantitative evidence. However, in the literature there are some works showing some effects of refactoring on external software quality attributes, such as changeability, maintainability and modifiability [46, 110] or on internal attributes, by exploiting the relationships between internal and external attributes. The relationship between refactorings and software metrics, such as internal quality metrics, has been studied in different works [19, 54, 15, 93].

Other works [15, 61, 73] propose coupling and cohesion metrics to evaluate and measure the effect of refactoring on maintainability or reusability. More recent works [74, 75] analyzed refactorings in the context of software networks, presenting a relationship between refactorings and node degree.

# Chapter 3

# Background

## 3.1 Bug extraction

We considered the number of defects (bugs) as the main indicator of software quality. Bug Tracking Systems (BTS) are commonly used to keep track of Bugs, enhancements and features of software systems. It is possible to collect data about the bugs of a software system by mining its associated BTS. In both of them defects are tagged with a unique ID number. An entry in BTS is called with the common term 'Issue', and it usually contains no information about classes associated to defects. We considered only issues that are labeled as fixed in the BTS.

To keep track of the development activities developers rely on Software Configuration Management systems (SCM) like Common Versioning Systems (CVS) [1], Subversion (SVN) and Git, just to name a few. These operations are recorded inside an SCM in an unstructured way; it is not possible, for instance, to query in a simple way a SCM to know which operations were done to fix Bugs, or to introduce a new feature, or an enhancement. All these operations are performed on files, called Compilation Units (CUs), which may contain one or more classes.

In order to identify Issues (Bugs) affecting given classes we had to match the data stored in the BTS with data stored in the corresponding SCM where all development history is recorded in the form of "commit operations" [1]. To obtain a correct mapping between Issue(s) and the related Java files, the compilation units, we analyzed the SCM log messages, to identify commits associated to maintenance operation where Issues are fixed.

Every positive integer number (including dates, release numbers, copyright updating, etc) might be a potential Issue-ID in BTS. In order to avoid wrong mappings between a file and the corresponding Issue, we filtered out any number which blatantly did not refer to bug fixes.

If a maintenance operation is done on a file to address a bug, we consider the CU as affected by this bug. We then assigned the bugs to classes in the corresponding CUs, but since there are a few CUs containing more than one class, we decided to assign all the bugs to the biggest class of those CUs. This method might not completely address the problems of the mapping between bugs and CUs [9]. In any case we checked manually 10% of CU-bug(s) associations (randomly chosen) for each release and each CU-bug association for 3 sub-projects without finding any error. A bias may still remain due to lack of information on SCM [9]. The subset of Issues satisfying the conditions as in Eaddy et al. is the Bug-metric [38]. Of course there are chances for wrong assignments to happen for some classes, but since the average bug number for class is very low, the number of wrong assignments in the entire system, considering also CUs with one class, is very limited.

## 3.2 Network Metrics

In this Section we reported the definitions of most meaningful network metrics adopted in this work.

### Clustering Coefficient

This metric quantifies the tendency to cluster: it is the mean probability that if vertex $A$ is connected to vertex $B$ and vertex $B$ to vertex $C$, then the vertex $A$ will also be connected to vertex $C$. It is defined by [81]

$$C = \frac{3 \times \text{ number of triangles in the network}}{\text{number of connected triples of vertices}}, \tag{3.1}$$

where a triangle is a set of three vertices all connected with each other, and the number of connected triples is the number of edges between the nearest neighbors of a node inside a clique [6]. High values of $C$ indicate a higher resilience to the removal of vertices, typical of real-world networks.

The clustering coefficient for the whole graph is the average of the $C_i$'s:

$$C = \frac{1}{n}\sum_{i=1}^{n} C_i, \tag{3.2}$$

where $n$ is the number of nodes in the network.

### 3.2.1 Modularity

Consider a network divided into $c$ communities and define a $c \times c$ symmetric matrix $\mathbf{e}$ whose element $e_{ij}$ is the fraction of all edges in the network that link vertices in community $i$ to vertices in community $j$. Given this matrix, we can get the fraction of edges in the network connecting vertices in the same community by computing its trace, $\mathrm{Tr}\,\mathbf{e} = \sum_i e_{ii}$. The latter would give an indication of a good community structure if it was close to 1, its maximum value, but this would be the most trivial case. To get a good measure of community structure, in [80] this fraction is compared with the expected value of the same fraction in a network with the same community divisions but random connections between the vertices. To do this, the authors define the column sums $a_i = \sum_j e_{ij}$, the fraction of edges that connect to vertices inside community $i$. Instead, in a network with no community structure, the edges fall between vertices uniformly at random and we would have $e_{ij} = a_i\, a_j$. Comparing these quantities, we obtain the modularity defined by the subtraction

$$Q = \sum_i (e_{ii} - a_i^2) = \mathrm{Tr}\, e_{ij} - \sum_{ij} (e_{ij}^2). \tag{3.3}$$

A good community structure is indicated by values of $Q$ close to 1. Typical values are found between 0.3 and 0.7 [80].

### 3.2.2 Community Structure Detection

The detection of the community structure is done by using the algorithm introduced by Clauset, Moore and Newman [22], (CMN). It is an agglomerative algorithm which is widely used; it performs the same optimization as the previous algorithm introduced in [79], giving the same community structure, but it is actually faster. CMN exploits some shortcuts which allow the computational cost to be nearly linear in time: for a sparse network with $n$ vertices and $m$ edges it is $O(n \log^2 n)$.

The algorithm computes the variations in modularity obtained by joining couples of communities until only one community is left. The modularity measure is computed using specific properties of the network: its adjacency matrix

$$A_{vw} = \begin{cases} 1 & \text{if } v \text{ and } w \text{ are joined} \\ 0 & \text{otherwise,} \end{cases} \tag{3.4}$$

where $v$ and $w$ are two vertices belonging to communities $c_v$ and $c_w$; the number of edges $m = \frac{1}{2} \sum_{vw} A_{vw}$, and the degree $k_v = \sum_w A_{vw}$. of a vertex, the number of its in-links. The modularity as defined in (3.3) can be expressed as a function of these properties: to show this, we note that the fraction of edges attached to vertices in community $i$ to vertices in community $j$ can be written as follows:

$$e_{ij} = \frac{1}{2m} \sum_{vw} A_{vw}\, \delta(c_v, i)\, \delta(c_w, j) \tag{3.5}$$

and so the vector $a_i$, the fraction of edges connecting vertices inside community $i$, is given by

$$a_i = \frac{1}{2m} \sum_{v} k_v\, \delta(c_v, i). \tag{3.6}$$

Using $\delta(c_v, c_w) = \sum_i \delta(c_v, i)\delta(c_w, i)$ and equations (3.5) and (3.6), the modularity $Q$ defined in (3.3) can be then written as a function of $A_{vw}$ and $m$:

$$Q = \frac{1}{2m} \sum_{vw} \left( A_{vw} - \frac{k_v k_w}{2m} \right) \delta(c_v, c_w). \tag{3.7}$$

The algorithm starts from considering every single vertex as a community, and at each step it joins two communities, it computes the change in modularity, it extracts the maximum value, and performs the corresponding joining, until only one community is left. So at first it initializes

$$e_{ij} = \begin{cases} \frac{1}{2m} & \text{if } i \text{ and } j \text{ are joined} \\ 0 & \text{otherwise,} \end{cases} \tag{3.8}$$

$$a_i = \frac{k_i}{2m}, \quad \text{and}$$

$$\Delta Q_{ij} = \begin{cases} \frac{1}{2m} - \frac{k_i k_j}{(2m)^2} & \text{if } i \text{ and } j \text{ are joined} \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

For each pair of communities (so for each $i$), the algorithm computes a sparse matrix containing $\Delta Q_{ij}$, and keeps the vector $a_i$ and a max-heap $H$ with the largest element of $\Delta Q_{ij}$ and the labels $i, j$.
Once the maximum value of the modularity is selected, the algorithm performs the corresponding joining of the two communities, incrementing $Q$ by $\Delta Q$, and it repeats the procedure until only one community is left.

The community structure was computed using the implementation of CMN algorithm given by the function *fastgreedy.community()* of the R package *igraph* [44].

## 3.3 Refactoring

Refactorings are code changes which do not modify the external system behaviour [43]. Usually developers decide to apply refactoring by examining or changing the software code while they are performing other operations [76], such as bug fixing, addition of functionalities, or other code changes. This process is widely used in Agile Development, where code is maintained and extended repeatedly in order to avoid code decay. Decay can be caused for example by unhealthy dependencies between classes or packages, bad allocation of class responsibilities, too many responsibilities per method or class, duplicate code, or simply confusion in the code. Changing code without refactoring can worsen the decay process, thus refactoring can spare a lot of time and costs in software development, by keeping the code easy to maintain and extend. To perform our analysis, we build the software networks associated to every release of our software projects and try to identify refactored classes and network connections among them.

The classes affected by refactoring have been retrieved with the use of RefFinder [3], the most commonly used automatic tool for the detection of refactoring operations. The 72 refactorings classified by Fowler [43] have been investigated also by other researchers with the purpose of finding other good techniques for automatic detection different from RefFinder [5], [8], [88]. Nevertheless RefFinder currently supports 65 of the 72 Fowler's refactorings, representing the most exhaustive coverage of all existing techniques. This tool compares two different software releases, analyzing the changes occurred from the first to the last, and identifies the refactoring operations according to Fowler's catalog. The output is the set of all refactored classes with the associated refactorings.

# Chapter 4

# Community Structure and Software Quality

## 4.1 Introduction

Modern software systems can be very large and can be made of tens of thousands, or even millions of lines of code. In the last decades, due to its simplicity,the use of Object Oriented (OO) programming paradigm largely increased and OO software systems are the majority in many applications. For any software system built according to the object oriented approach it is possible to easily define different kinds of networks, where the nodes represent specific software modules and connections among nodes represent relationships between software modules. Many software systems have been demonstrated to exhibit the structure and to possess the properties of a complex network [41], [77], [11], [18], [102], [104].

The investigation of complex networks received a large attention in the last decades, where many quantities for measuring and characterizing their complexity have been defined and used, and many algorithms and software programs have been developed for computing these quantities, with large impacts on the knowledge and understanding in very different fields and disciplines.

One of the important features of complex networks is the possibility of partitioning them into smaller sub-networks preserving complexity features. Using the language of social networks these sub-networks are named *communities*, meaning that there are strongly interconnected nodes inside a single community, while the connections with nodes of other communities are sparse or weak [80].

From this point of view software systems are excellent candidates for complex networks with an underlying community structure, since they are structured in a hierarchical manner, where small units cooperate with each other. Software engineering practices emphasize the decomposition of complex tasks in smaller ones, to encourage code reuse and agile development [26], and software systems are designed to be highly evolvable [77]. Nevertheless in the field of software, while the concept of software network is now largely used, there are very few researches investigating the community structure of software networks, finding that the software systems analyzed exhibit a significant community structure that doesn't match the packages structure imposed by the designer [66], [104].

Software modularization is acknowledged as a good programming practice [87, 10, 91] and a certain emphasis is put on the prescription that design software with low coupling and high cohesion would increase its quality [19]. We present a study on the relationships between software systems quality and their modular structure. To perform this study we used an approach based on the concept of complex network.

Due to the fact that that software systems are inherently complex, the best model to represent them is by retrieving their associated networks and the related topological properties [77, 104, 109, 94, 113]. In other words, in a software network nodes are associated to software modules (e.g. classes) and edges are associated to connection between software modules (e.g. inheritance, collaboration relationships).

We investigated the software modular structure - and its impact on software quality - by studying specific network properties: community structure, modularity and clustering coefficient. A community inside a network is a subnetwork of densely connected nodes when compared to nodes outside the community [48]. Modularity is a function that measures how marked is a community structure (namely the way the nodes are arranged in communities) [80]. The clustering coefficient is a measure of connectedness among the nodes of a network [81].

In this Chapter we are presenting two studies. The first one is a case study on a release of a popular software system written in Java, NetBeans. Our aims was to investigate if some metrics which have been proved useful for characterizing the community structure can be of help for characterizing the properties of the related software network.

In the second one, we studied several releases of a large software system, Eclipse, performing a longitudinal analysis on the relationship between community structure, clustering coefficient and software quality. Our aim is to figure out if the studied metrics can be used to better understand software quality evolution and to predict software quality of future releases.

In this work we use one of the algorithms proposed by Newman et al. to understand if such division can be related to software modularity as defined in software engineering and, eventually, if the community metrics may be useful to predict bugs in systems future releases.

In the first part of this Chapter we are presenting a case study on the relationship of the community structure and software defectiveness performed on a release of a popular IDE, NebBeans.

## 4.2 Experimental Settings

In this Section we present the structure of our research. For sake of clarity we we will report separately the experimental settings for the two studies. We will refer to the two studies with the name of the studied software systems.

### 4.2.1 Datasets

**NetBeans**

The reported case study regards a large software system, Netbeans, release 6.0. It is formed by 56 sub-projects, that are almost independent from each other and contain a total number of around 44000 classes. We retrieved the software network of each subproject by parsing the source code, looking for inter-class dependency like inheritance, composition, etc.

**Eclipse**

We analyzed 5 releases of Eclipse, whose main feature are presented in Table 4.1. Each release is structured in almost independent sub-projects. The total number of sub-projects analyzed amounts at 375, with more than 60000 nodes (classes) and more than 350000 defects.

| **Release** | **2.1** | **3.0** | **3.1** | **3.2** | **3.3** |
|---|---|---|---|---|---|
| Size | 8257 | 11406 | 13413 | 16013 | 17517 |
| Sub-Projects n. | 49 | 66 | 70 | 86 | 104 |
| N. of defects | 47788 | 59804 | 69900 | 80149 | 95337 |

Table 4.1: Main features of the analyzed releases of Eclipse: size (number of classes), number of sub-projects (sub-networks), and total number of defects.

Both the studied software systems are popular IDE (Integrated Development Environment) written in Java and widely used by the practitioners community. The data we used belong to a wider set of networks data collected by some of the authors and analyzed in previous works [27]

### 4.2.2 Metrics

**NetBeans**

Many of these metrics where already thoroughly defined in Section 3.2. For sake of clarity we are reporting also the definitions in this paragraph. We distinguish *System Metrics* from *Network Metrics*.
*System metrics*, computed both at system and at class level, are the following:

- *System Size*: the number of classes of the software system.

- *Number of Packages (NOP)*: the number of different subdirectories used by the developers to insert the classes.

- *Bug Number (BN)*: the number of bugs found in the class.

- *Average Bug Number (ABN)*: the number of bugs found into a system divided by the number of classes.

*Network Metrics* are the following:

- *Modularity (Q)*, which measures the quality of a community structure, defined as in Section 3.2.1.

- *Number of Communities (NOC)*: the number of disjoint communities in which the network is partitioned according to the CMN algorithm described in the next Section.

- *Mean Degree (MD)* of the complex network, the mean number of edges connected to a vertex.

- *Average Shortest Path (ASP)*, the mean geodesic distance among networks nodes.

- *Clustering Coefficient (CC)*, also known as *transitivity*, whose definition is reported in Section 3.2, for each of the communities detected.

#### Eclipse

We computed the following metrics:

- *System size*: the number of classes of the software system.
- *Average Bug Number (ABN)*: bug density, namely the number of defects found into a system divided by the number of classes.
- *Modularity*: a measure of the strength of the obtained community structure, as defined in Section 3.2.1.
- *Number of Communities (NOC)*: the number of disjoint communities in which the network is partitioned.
- *Clustering Coefficient (CC)*: as in the previous paragraph, it is defined in Section 3.2.

We detected the modularity and its associated community structure for each subproject of each release using the community detection algorithm devised by Clauset et al. [22]. This is an agglomerative clustering algorithm that performs a greedy optimization of the modularity. The community structure retrieved corresponds to the maximum value of the modularity. The community structure of the networks analyzed in this paper was computed using the implementation of Clauset-Moore-Newman (CMN) algorithm, with the implementation is given by the function *fastgreedy.community* of the R package *igraph* [44].

### 4.2.3 Analysis

In order to build the software networks we parsed the Java source code to find the dependencies between different classes. We considered classes as graph nodes, and relationships among them as links. The relationships, computed at the source code level, can be inheritance, composition and dependence. After having analyzed the source code for building the corresponding software graph, we computed different software metrics - reported in the previous paragraph. Then we extracted bugs affecting classes from bug repository and we associated them to the corresponding classes.

#### NetBeans

We applied the community analysis to the obtained software networks and computed four community metrics, in order to understand if they can provide indications about the fault proneness of the system. Using the CMN algorithm presented in Clauset et al. [22], we analyzed the community structure of the networks considered as independent systems and measured some of the most common community metrics, such as modularity, clustering coefficient, mean distance between nodes and mean node degree. We found that, among the community metrics, modularity and clustering coefficient show interesting correlations to software quality and defectiveness. We also performed an analysis at a global level, focusing on the relationship between the number of communities and the number of packages. We found that the number of packages is systematically larger than the number of communities, showing that the division given by developers is different from the community structure computed by the algorithm, a result which is probably related to some design choices. Finally we investigated the correlation between the number of communities and software faultness, finding that the mean bug number increases with the number of communities for medium size systems. To accomplish this comparison we use the research questions approach, formulating three questions:

- **RQ1**: *Are there correlations between the community structure and software defectiveness?*

- **RQ2**: *Are there correlations between the community metrics and software defectiveness?*

- **RQ3**: *Do the software networks analyzed present a community structure that matches the package structure devised by developers?*

The answers to these research questions will be discussed after the analysis of the results.

### Eclipse

We performed a correlation analysis among the network metrics and the software metrics considering each release on its own and the entire dataset, in order to have a better relevant statistical bases. To study the system evolution we used the following approach. We first carried out the analysis for each release, and then we assembled together different releases, according to its temporal evolution, taking into account the different releases. More precisely, for the 5 releases of our dataset, we studied the evolution of the system by cumulating the first and the second releases in a single set, then adding the third release to this first set to obtain a second set and so on. This way we were able to make predictions about the next release starting from those cumulated in the previous assembly.

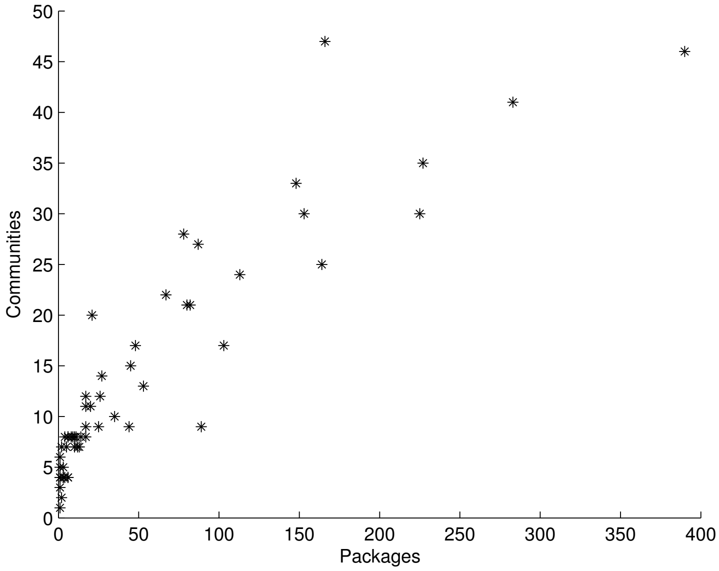


Figure 4.1: Number of communities vs. number of packages.

## 4.3 Results

### 4.3.1 NetBeans

In this section we analyze the results of our case study. Figure 4.1 reports the plot of the number of communities versus the number of packages for the 56 subprojects. The scatter plot shows a good correlation among the variables, and a sublinear dependence of the number of communities on the number of packages, excluding the case of very small projects. Their correlation and the level of significance are respectively 0.9018 and the level of significance (*p-value*) is almost zero.

Next we consider the average bug number of the systems, which is a size independent metric, but is a direct indicator of system quality. In general, the overall correlation of the mean bug number with the number of communities in each subproject is quite poor. But when we restrict the analysis to medium size systems, namely for systems in the range 250-1200 classes, there is a net increase of the correlation with the number of communities. Such correlation means, in terms of software quality, that the larger is the number of detected communities the poorer is the system quality.

Tab. 4.2 shows the correlations and p-values among mean bug number and number of communities for all the 56 systems and for the systems in the range 250-1200 classes.

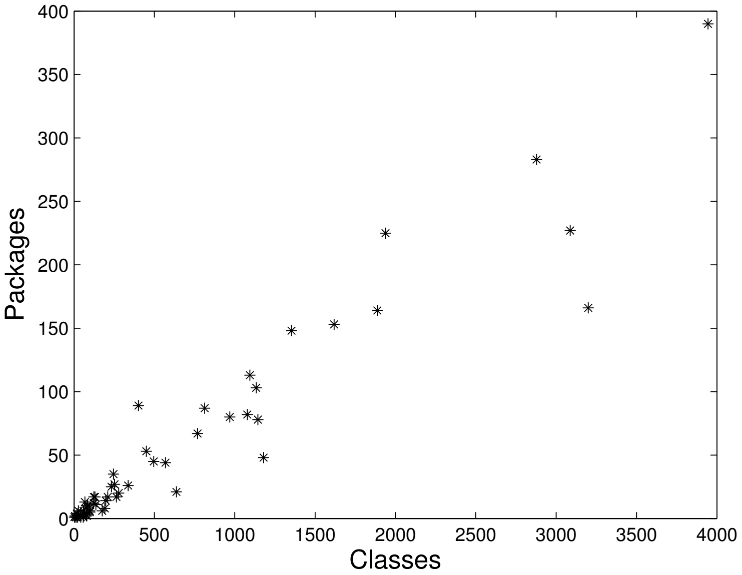


Figure 4.2: Number of packages vs. system size.

| | Correlation Coefficient | P-value |
|---|---|---|
| all systems | 0.1088 | 0.4200 |
| medium size systems | 0.6598 | 0.0040 |

Table 4.2: Pearson correlation and p-values among number of communities and mean bug number for all the systems and for systems in the range 250-1200 classes.

Next we consider variability inside the single systems. In this case the data variability is reduced, since there are small systems which are partitioned in too few communities to be analyzed with a statistical approach. Each system is divided into communities according to the CMN algorithm. We collected, for each community and for each system the bug number, the size, the clustering coefficient, the modularity, the average shortest path and the mean degree.

The results show a systematic high correlation among the number of bugs and the size of the communities, as we would expect. What is interesting to note is that, while the linear (Pearson) correlation is not high, there are systematic general relationships among the bug number and modularity, and among the bug number and the clustering coefficient. Namely, higher metric values correspond to higher bug numbers.

In order to measure and quantify these features we computed the Spearman correlation coefficients and p-values for the correlation among number of bugs inside a community and modularity, clustering coefficient, average shortest path and mean node degree. The Spearman correlation coefficient is able to identify threshold effects, since it accounts for a monotonic, while non linear, increase of one variable versus another variable. Tab. 4.3 reports the results. In the same Table we also computed the medians of the modularities for the 13 systems, and the percentage of bugs belonging to the communities above the median, in order to estimate the threshold effects.

### 4.3.2 Eclipse

The application of the CMN algorithm confirms that software networks present a meaningful community structure [94, 29]. Our results show a general tendency for certain metrics to converge to a narrow range of values when the number of classes increases. Figures 4.3, 4.4 and 4.5 show the relationship between systems' size (number of classes) and, respectively, modularity, defect density and clustering coefficient. All the metrics display more or less the same behavior. For relatively small systems, where the number of classes is roughly below 100, the metrics assume values in a large range. Specifically, the defect density ranges from 0 up to 25, the clustering coefficient and the modularity, whose maximum value may be 1, range from 0 to 0.6-0.7. For system's size between 100 and 500 roughly, all the oscillation ranges decrease: the defect density lays between 2 and 12, the clustering coefficient lays between 0.05 and 0.2, and the modularity between 0.3 and 0.6. Finally, for fairly large systems, where the number of classes is above 500 or more, the metrics stabilize, showing small oscillations and eventually converging asymptotically to precise values. On the contrary, the

| Network | CC | MD | APL | Mod | Mod med | % bug above |
|---|---|---|---|---|---|---|
| cnd | 0.513 | 0.744 | 0.645 | 0.566 | 0.311 | 99.3 |
| core | -0.420 | 0.804 | 0.634 | 0.628 | 0.453 | 74.3 |
| editor | 0.304 | 0.776 | 0.742 | 0.740 | 0.300 | 96.8 |
| enterprise | 0.072 | 0.708 | 0.603 | 0.541 | 0.303 | 97.2 |
| j2ee | 0.488 | 0.793 | 0.770 | 0.746 | 0.255 | 92.0 |
| mobility | -0.104 | 0.834 | 0.761 | 0.756 | 0.208 | 98.6 |
| ruby | 0.659 | 0.688 | 0.560 | 0.489 | 0.300 | 89.9 |
| serverplugins | 0.489 | 0.821 | 0.809 | 0.770 | 0.212 | 94.4 |
| uml | 0.548 | 0.724 | 0.678 | 0.663 | 0.178 | 94.0 |
| visualweb | 0.064 | 0.646 | 0.469 | 0.460 | 0.304 | 93.3 |
| web | 0.537 | 0.550 | 0.571 | 0.625 | 0.349 | 78.9 |
| websvc | 0.560 | 0.753 | 0.677 | 0.718 | 0.365 | 95.6 |
| xml | 0.611 | 0.726 | 0.633 | 0.640 | 0.317 | 93.8 |

Table 4.3: Correlation data between bugs and Clustering Coefficient (*CC*), Mean Degree (*MD*), Average Path Length (*APL*) and Modularity (*Mod*). The last two columns contain the modularity medians (*Mod med*) and the percentage of bugs above this median. (*% bugs above*).

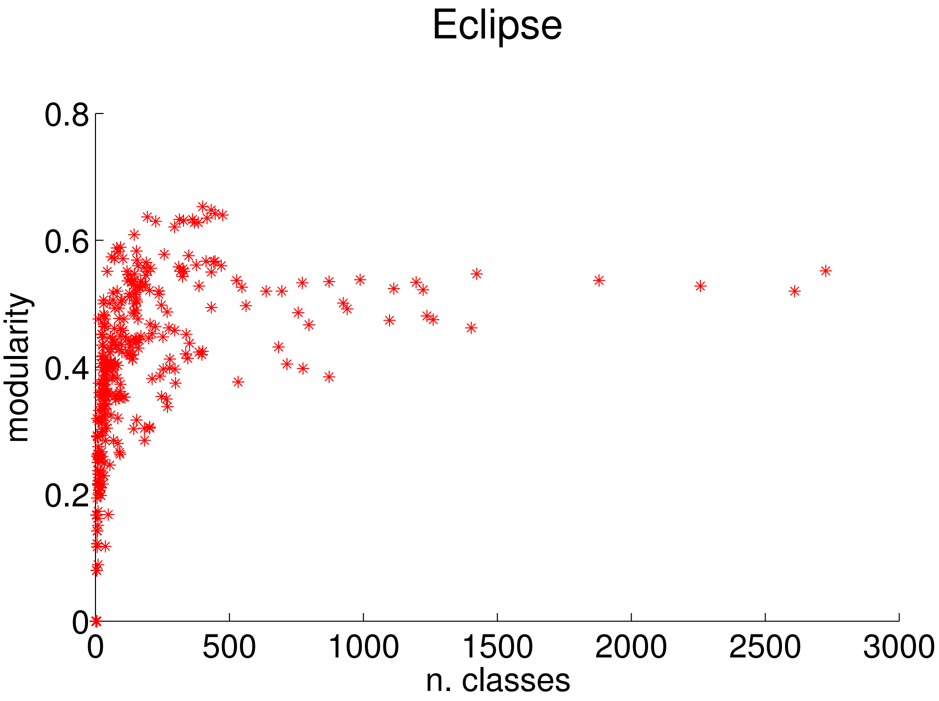


Figure 4.3: Scatterplot of the modularity vs system's size (n. of classes) for all subprojects

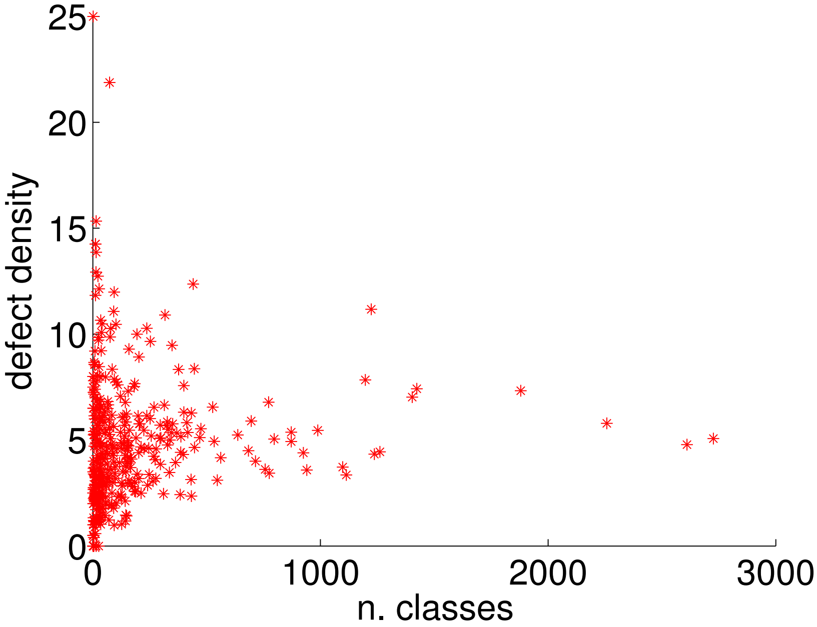


Figure 4.4: Scatterplot of the defect density vs system's size (n. of classes) for all subprojects.

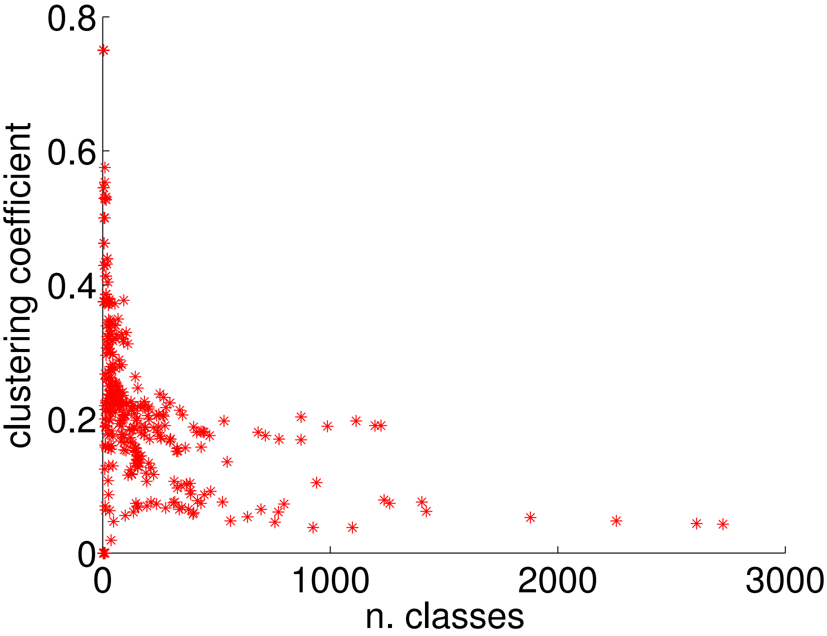


Figure 4.5: Scatterplot of the clustering coefficient vs system's size (n. of classes) for all subprojects.

| **Eclipse** | **max ADD vs NOC** | **max CC vs NOC** |
|---|---|---|
| dof | 13 | 13 |
| $\chi^2$ / dof | 0.361 | 1.005 |

Table 4.4: Fit data for the power laws between the maximum average defect density (max ADD) versus the number of communities and maximum clustering coefficient (max CC) versus the number of communities: correlation coefficient ($r$), normalized Chi squared ($\chi^2$ ), and number of degrees of freedom ($dof$).

NOC metric shows a monotonic increase with system's size, where after a first nonlinear behavior the curve aligns along a straight line, as reported in Figure 4.6. We also found a significant correlation between the number of communities (NOC) and both ABN and CC. It is worth to point out that for other network metrics such as mean degree or average path length, these correlations are not significant. Fig. 4.7 shows that the number of communities in a system increases, the maximum values of average bug number and clustering coefficient of the system tend to decay in a super-linear fashion. In particular, Figures 4.7a and 4.7b show the distributions of ABN (Figure 4.7a) and of CC (Figure 4.7b) with respect to the number of communities (NOC) for all the subprojects of all the releases. Starting from this result, we found that there is a power law relating the maximum values of defect density and the maximum values of clustering coefficient in systems having the same number of communities to the number of communities itself. This led us to hypothesize that there might be a linear relationship between NOC and ABN and we investigated if we can exploit this relationship in order to predict the defectiveness of a future release of a software, knowing the history of the previous releases. Figures 4.8 and 4.9 show, in a log-log scatterplot, the relationships between NOC and ABN for the former and between NOC and CC respectively for the latter. Each color correspond to each set composed by a number of releases, collected according to the chronological order of the releases. For each distribution we reported the corresponding fitting line. Tab. 4.5 and Tab. 4.6 report, among the others, the values of the exponent for the power-law, namely the coefficient of the fitting line. They refer to different "cumulated" releases for the relationship between NOC and CC and ABN respectively. The latter Figures confirm that the power-law like relationship appears in every analyzed release and is a regular behavior throughout software evolution.

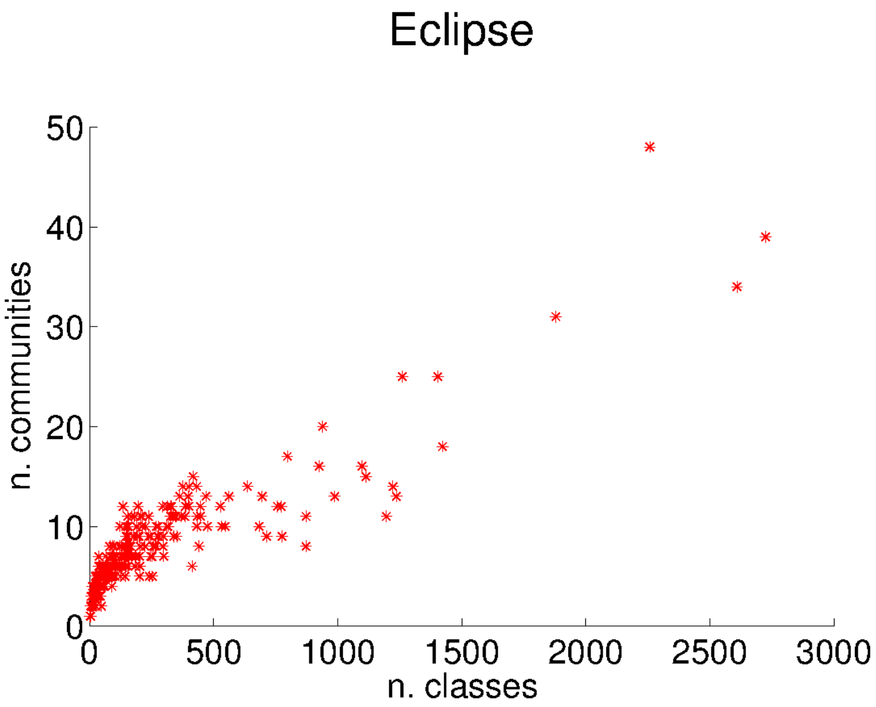
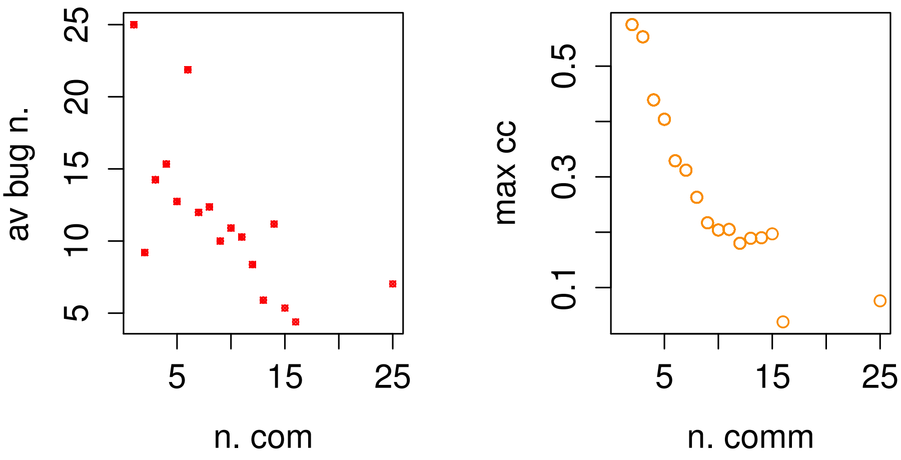


Figure 4.6: Scatterplots reporting the number of communities (vertical axis) and the sizes of the subprojects (in number of classes, on the horizontal axis). It clearly shows a linear correlation between the two measurements.

| **Releases** | $\alpha$ | $r$ | $\chi^2$ | $dof$ |
|---|---|---|---|---|
| 2.1 - 3-0 | -1.010 | -0.654 | 0.075 | 16 |
| 2.1 - 3.1 | - 0.917 | -0.667 | 0.057 | 17 |
| 2.1 - 3.2 | -0.977 | -0.715 | 0.087 | 20 |
| 2.1 - 3.3 | -0.986 | -0.712 | 0.119 | 21 |

Table 4.5: Results on the power law between maximum Clustering Coefficient vs Number of communities for Eclipse: exponent $\alpha$, correlation coefficient $(r)$, value of Chi Squared $(\chi^2\ )$ and number of degrees of freedom $(dof)$.



(a) Average Bug Number vs. Number of Communities

(b) Clustering Coefficient vs. Number of Communities

Figure 4.7: Scatterplot of the relationships between the studied metrics.

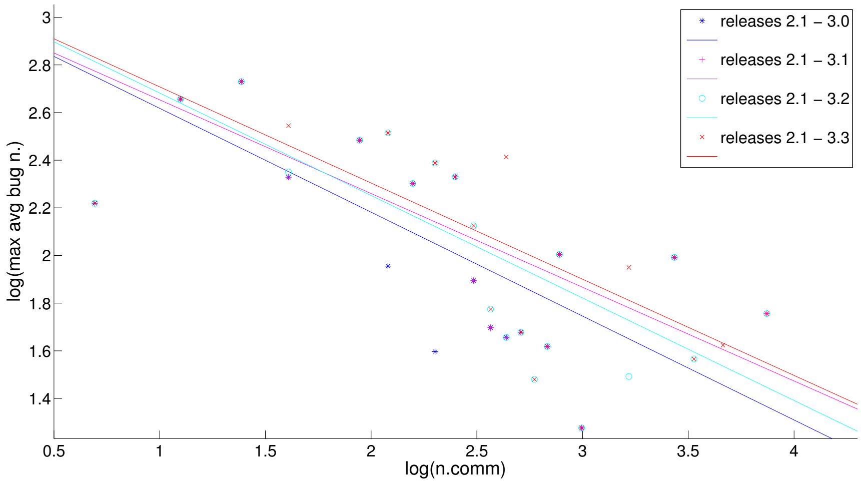


Figure 4.8: Eclipse: cumulated log-log plots and best fitting lines of maximum defect density vs number of communities

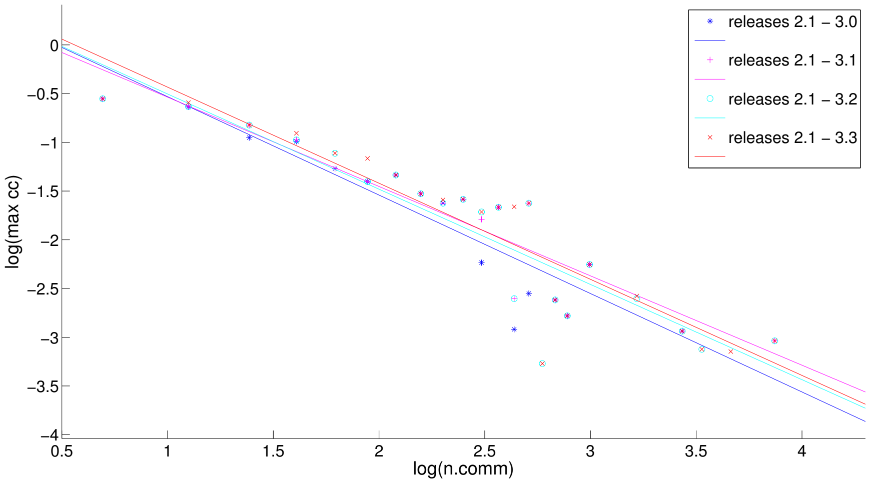


Figure 4.9: Eclipse: cumulated log-log plots and best fitting lines of maximum clustering coefficient vs number of communities

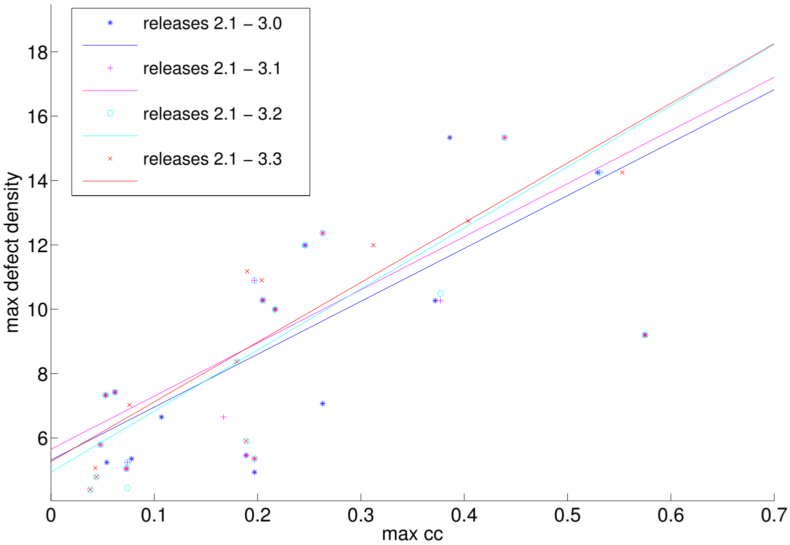


Figure 4.10: Cumulated plots and fitting lines for the maximum defect density vs maximum clustering coefficient.

| **Releases** | $\alpha$ | $r$ | $\chi^2$ | $dof$ |
|---|---|---|---|---|
| 2.1 - 3-0 | -0.436 | -0.920 | 0.065 | 16 |
| 2.1 - 3.1 | -0.393 | -0.934 | 0.059 | 17 |
| 2.1 - 3.2 | -0.4302 | -0.911 | 0.056 | 20 |
| 2.1 - 3.3 | -0.404 | -0.921 | 0.052 | 21 |

Table 4.6: Results on the power law between maximum defect density vs number of communities for Eclipse: exponent $\alpha$, correlation coefficient $(r)$ , value of chi squared $(\chi^2)$ and number of degrees of freedom $(dof)$.

| **Releases** | $r$ | $\chi^2$ | $dof$ |
|---|---|---|---|
| 2.1 - 3-0 | 0.565 | 0.633 | 16 |
| 2.1 - 3.1 | 0.576 | 0.651 | 17 |
| 2.1 - 3.2 | 0.677 | 0.523 | 20 |
| 2.1 - 3.3 | 0.687 | 0.547 | 21 |

Table 4.7: Fit data for the maximum defect density vs maximum clustering coefficient: correlation coefficient $(r)$, normalized Chi squared $(\chi^2\ )$, and number of degrees of freedom $(dof)$.

## 4.4 Discussion

### 4.4.1 NetBeans

As reported, looking at Figure 4.1 we found a linear correlation between the number of communities and the number of packages. Correlation is 0.9018 while p-value is almost zero

This correlation value is what we would expect, since both variables are related to system size, but the sublinear dependence shows that the partition in packages established by developers is finer than the partition in communities obtained with the CMN algorithm, which depends on the links among nodes. Namely, while developers insert classes with the same purposes and functionalities in Netbeans packages, the community detection suggests instead that there are many interdependencies among such packages, since they are not seen as separated communities on the base of their connections. This result is not true for relatively small systems (with less than 100 classes), where instead the number of packages can be larger than the number of communities, meaning that single packages are partitioned in communities.

It is interesting to compare our results with the dependence of the number of packages on the number of classes, which is almost perfectly linear (Fig. 4.2) with linear (Pearson) correlation $\rho = 0.9549$ and *p-value* almost zero.

This means that developers tend to create packages with roughly the same number of classes, since the mean number of classes per package remains constant (around ten, in this case). This is not true for the number of communities, where the mean number of classes per community increases with system size.

These empirical data show that the larger the system, the more dense are the connections among nodes. From the software architecture perspective, this means that the larger the system, the larger are the groups of classes directly linked or dependent from each other.

Tab. 4.2 reports the correlations and p-values between mean bug number and number of communities for both all and for the systems in the range 250-1200 classes. These different situations should find an explanation from the point of view of the software development. In fact, the difference appears when dealing with medium size systems, and is not present for small and large systems. In the case of small systems the statistics are too low. In fact systems with less than a few hundreds of classes are hardly divided into many communities. Thus any relationship existing for the number of communities is biased by a lower cut-off.

In the case of large systems instead, the statistics are large enough, and some other mechanism may apply. One possible explanation is the following. The partition in many communities implies that classes are arranged in many relatively small networks, where they are much more coupled than with the remaining classes of the system. Thus, to obtain a large number of communities, coupling among classes must be locally high and globally low. This means that on average classes will tend to have a larger value of CBO, and this has already been demonstrated to be a bad and fault prone programming practice [54]. On the other hand, when the system is large, the complexity increases and developers tend to loose the plain control of the software graph structure. The global coupling increases and the number of communities increases as well in a random fashion. Thus the correlation among mean bug number and number of communities disappears.

The reported results show that, apart from the clustering coefficient, the Spearman correlation among bug number and community metrics is between 0.5 and 0.8, with p-values below $10^{-2}$ (not reported in Tab. 4.3). Furthermore, for almost all systems the percentage of bugs belonging to the communities with modularity above the median is more than 80%. The medians of modularity of the communities are very close for all the systems, ranging from about 0.2 to 0.3, with one system presenting 0.45. Thus if one chooses 0.3 as threshold value for all the systems, the percentage of bugs into communities above this threshold is close to 80-90%.

Since the bug number in each community increases with community size, one should ask if the high values for the correlations among bug number and metrics are due to a trivial correlation between community size and modularity, average path length and mean degree. To check for a possible size influence on these correlations, we performed some tests on different networks. In particular we checked if modularity increases according to the network size. We first computed the modularity for some networks we built from scratch whose sizes differ by some orders of magnitude, but sharing the same community structure.

Next we computed the modularity for a set of networks generated according to the Girvan-Newman benchmark [80], using the implementation provided by Lancichinetti, Fortunato and Radicchi described in [65]. We found that the modularity computed for these networks does not change significantly as the network size increases. This is a counterexample showing that modularity and the other community metrics are not in general correlated with the size of a network.

In software networks, bugs and size are usually correlated, but our tests show that community metrics are not always bigger for the communities containing more classes. For our case study we can conjecture that the high correlations between bugs and community metrics are not an effect of the size, but they could depend on the topology of the communities detected by the CMN algorithm for the given software networks, namely by the way classes depend on each other. For such networks the CMN algorithm aggregates classes into clusters which are bigger and with more bugs whenever the three discussed metrics are larger. Now we can answer to the research questions.

- **RQ1**: *Are there correlations between the community structure and software defectiveness?*
  The answer to this research question is partially positive. In fact, considering all the 56 systems analyzed there is not net correlation among the mean bug number and the number of communities. But if one restricts the analysis to medium size systems, a net correlation appears. We tried to provide a possible explanation for this behavior in the text, but our hypotheses need to be verified carefully on different systems.

- **RQ2**: *Are there correlations between the community metrics and software defectiveness?*
  The answer is positive for three out of four of the metrics analyzed. We found very high Spearman correlations for all the 13 systems analyzed for the modularity, the mean degree and the average path length. We did not find a systematic high correlation for the clustering coefficient. We also checked that these correlations were not trivially determined

by a size effect. We found that there is a threshold effect so that most of the bugs, roughly 80%, belong to the communities with modularity values above the median, which is roughly 0.3 for every subproject.

- **RQ3**: *Do the software networks analyzed present a community structure that matches the package structure devised by developers?*
  The answer is in general negative. The CMN algorithm aggregates classes in communities which do not respect the design devised by developers, and do not match with the packages. In general the number of packages is much larger than the number of communities detected. Furthermore, while the number of classes in packages is roughly the same, the number of classes in communities increases with system size. This means that the larger the software systems, the larger are the groups of classes directly linked or dependent on each other.

### 4.4.2 Eclipse

We analyzed a large software project using complex network theory with the aim of achieving a better understanding of software properties by mean of the associated software network. The results show the existence of meaningful relationships between software quality, represented by the average bug number (ABN), and community metrics, in particular the number of communities (NOC) and clustering coefficient (CC).

The presence of a strong community structure in a software system reflects a strong organization of classes in groups where the number of dependencies among classes belonging to the same community (inter-dependences) is higher with respect to the number of dependences among classes belonging to different communities (external-dependences). From a software engineering perspective this goal might be achieved by adopting good programming practices, where class responsabilities are well defined, classes are strongly interconnected in groups, and coupling among groups is kept low. Within this perspective the network modularity can be seen as a proxy for software modularity.

Figure 4.3 shows that, with the exception of sub-projects with less than 500 classes, the modularity does not increase along with the size, converging to values that range from 0.6 to 0.7. As reported in Section 3.2.1, these values indicate that the community structure is significant and well defined. At the same time Figure 4.6 shows that there is a linear relationship between the number of communities and the number of classes. Such relationship is not trivial: the modularity and the number of communities are theoretically independent by the size [49] and, in general, the number of communities does not increase with network's size. Moreover, by and large, there may be large networks divided in a small number of communities, depending on the network's topology. As a consequence our findings suggest that, in the examined case, it is possible to partition the software networks into a set of communities, where the number of communities is correlated with system's size.

Figures 4.4 and 4.5 report, respectively, the relationship of ABN and CC with the number of communities. Both metrics have a similar trend, with values converging to a range between 4 and 12 for ABN and between 0.2 and 0.6 for CC. This means that when the system's sizes increases the number of defects stabilizes and the same happens to the clustering coefficient. We already mentioned the significant increment of the number of communities with system's size. Since the increment of NOC is not trivial, this led us to assume that there might be a relationship among the topology of software networks, that determines the number of communities, and the other metrics.

In order to investigate this relationship we collected the projects with the same number of communities into various sets. Any set includes all the sub-projects that have the same number of communities, which we denote by $k$, being $k$ a number in the range from 1 to the maximum number of communities detected, denoted by $k_{max}$. We computed the maximum defect density and the maximum clustering coefficient of all the projects in the same set. The results of this analysis are shown in Figures 4.7a and 4.7b that show the occurrence of the same kind of relationship of CC and ABN with the number of communities. Moreover, this relationship seems to follow a power-law trend in the maximum values of both the average number of bugs (ABN) and the CC associated to each community.

The power law relating the NOC and the maximum values of ABN indicates that the community metrics, specifically the number of communities, can be exploited in order to evaluate the evolution of the defectiveness of a software system. In other words, once the relationship between NOC and the maximum values for ABN is known one can evaluate approximately the maximum ABN in a future release of the same system, by computing the number of communities for that release. This way, we might assume that systems with the same number of communities should have a number of defects per class lower than a given value.

The same argument applies to the clustering coefficient of systems having the same number of communities. The relationship between CC and NOC is againg a power law. This implies that if the NOC of an initial release (or of a set of releases) is known, one can in principle predict that in the following releases the clustering coefficient will not be greater than a certain value. These results might help developers to estimate the expected maximum ABN for software systems with a known community partition.

Since a power laws relates the maximum values of both CC and ABN to NOC there must be a relationship between the first two metrics. In order to investigate the predictive power of the mentioned network metrics, we performed the following analysis. Specifically, we evaluated if, with a starting dataset of $N$ releases, the best fitting curve for the cumulated $N-1$ releases could also be a good fit for the $Nth$ release. Figures 4.7a and 4.7b show, in a log-log scale, the relationship of NOC with ABN and of NOC with CC, respectively. Tables 4.5 and 4.6 report the power law exponents, the correlation coefficients, the $\chi^2$s and the degrees of freedom ($dof$) for the best fitting in log-log scale. These tables show that the power laws parameters do not change significantly from one cumulated release to another. This suggests the existence of a progressively more stable behavior during software evolution, where the fitting with a power law becomes more accurate and tends to a fixed value as new releases are added in the cumulated dataset.

The scatterplots portraied in Fig. 4.7 show the relationship between the maximum defect density and the maximum clustering coefficient, for all the cumulated releases, along with the best fitting straight line. Table 4.7 reports the results for the best fitting for the relationship between CC and ABN showing that the reported linear correlation is not very high. However, the $\chi^2$ test returns an high level of significance. Table 4.4 reports the results of the analysis on the forecast for software quality. We computed the ratio between the $\chi^2$ and the degrees of freedom. According to the results reported on Table 4.5, on the right, the $\chi^2$ values are close to 1, meaning that for the given degrees of freedom the fits are good.

These results can be explained by noting that the larger the clustering coefficient, the higher is the number of classes linked to each other and the higher the probability of diffusion of defects among them. The topology of a software network is characterized by hubs, and the clustering coefficient in the area of the graph around any hub is higher by definition. If one hub is affected by one or more defects, it is more likely that these defects will spread among the classes connected to the hub, increasing the defect density in the area around it. This would explain the correlation among defect density and clustering coefficient.

## 4.5 Conclusion

In this Chapter we reported two studies on the relationship between network metrics and software quality. The first research is an analysis of a release of NetBeans, specifically Netbeans 6.0, which is an OO software system written in Java, consisting of more than 44000 classes and of 56 subprojects. The second study presented is a longitudinal analysis on the evolution of a large software system, Eclipse, with a focus on software defectiveness. In both the studies we built the software networks corresponding to all the subprojects, and analyzed their structure as complex networks.

We computed several network metrics, including, mean degree, average shortest path and clustering coefficient, not to mention modularity and number of communities of the network community structure, using the CMN algorithm. We extracted the number of bugs from the Issuezilla (for NetBeans) and BugZilla (for Eclipse) bug repository, and associate the number of bugs to the networks nodes representing software classes. In the following we separately report the results.

### NetBeans

Our findings show that, at least for the analyzed release of NetBeans, medium size sub-projects hold a different community structure, which appears related to the mean bug number for class. Thus, for these systems, the partition in a large number of strongly aggregate clusters of classes can carry an enhanced defectiveness.

We also found that the clustering coefficient and the modularity, when computed for the communities inside a single system, appear to be good indicators for software bugs. We chose the median of the specified networks metrics (modularity or clustering coefficient) as an appropriate threshold value that discriminates between highly defective communities and the others. We believe that this could be a method to select communities when looking for fault proneness modules, for example during test or maintenance activities.

### Eclipse

As well as in the first study, after having retrieved the number of defects and associated them to the software network classes, we performed a topological analysis of the system defectiveness. We found a power law relationship between the maximum values of the clustering coefficient, the average bug number and the division in communities of the software network. This led to a linear relationship between the maximum values of the clustering coefficient and of the average bug number. We showed that such relationship can in principle be used as a predictor for the maximum value of the average bug number in future releases.

# Chapter 5

# Clustering and defects

## 5.1 Introduction

Software systems are subject to evolution and changes during their development and maintenance, but also to code decay. From one release to the next the system evolves and changes, because new functionalities may be implemented, defects need to be fixed, software needs to be maintained, software quality must be improved, and so on. In particular, software quality needs to be monitored during maintenance and development, in order to keep times and costs as low as possible. This can be done for example by fixing defects, or addressing in general the Issues reported in Bug Tracking Systems (BTS). It is well known in fact that software quality is closely related to bugs and minor fixes, and that it can be kept high by controlling the density of defects and issues.

The study illustrated in the present Chapter aims at understanding whether we can determine, starting from a class or a file that has been subject to a maintenance activity, which other classes or files are good candidates for corrective or maintenance activities, such as bug fixing or similar. The purpose of this investigation is extremely practical, since this information is very valuable to developers who start bug fixing on a file, and must then proceed finding and examining which other classes or files need to be modified as well, according to the performed activity.

This problem can be easily tackled when the software system is interpreted as a network, where classes or files are linked to each other through software relationships, like dependencies, inheritance and so on. Many software systems have been demonstrated to exhibit the structure and to possess the properties of a complex network [77], [11], [18], [102], [104], [66] and several researchers has recently produced meaningful results by applying a complex network approach to software systems [107, 12, 48, 68, 89, 77, 64, 104].

From the statistical analysis of software defectiveness it is possible to develop useful models of bug prediction. Many authors have faced this problem [51, 58], and recently it was shown that the number of bugs in a software system follows the Pareto law [40, 7, 112]. In particular, in [112] is shown that the distribution of bugs in software modules can be described by an Alberg diagram [83], while in [25] these distributions are fitted very well with a Yule-Simon distribution, which can also be a valid generative theoretical model to explain the recurrence of bugs over time in a software system.

In this Chapter we present some results showing that classes or files which have been subject to a fixing procedure form connected subnetworks or clusters, thus displaying higher coupling inside software network systems. We compared the number of clusters formed by files affected by different issues, with the number of clusters formed by a random selection of files. Our results show that any intervention on the code which can be associated to an issue “id” inside software repositories is done on classes or files which are interconnected and form clusters, rather than being disconnected and independent. The analysis reveals a significant tendency of files hit by issues to be linked to each other. More precisely, files affected by issues present, on average, a higher density of connections, i.e, higher coupling. This information can be used to better understand and investigate the quality of a software system by looking at the topology of its corresponding network, since defects and issues are more likely to spread among connected classes.

## 5.2 Experimental setting

We analyzed a total of 7 releases of two large Open Source (OS) Object Oriented (OO) software system written in Java. These are two Integrated Development Environments (IDE), Eclipse and Netbeans, quite popular among Java practitioners. They are composed of source files with .java extension, named Compilation Units (CU): the information about bugs and issues, extracted from the commit logs, is associated to these files. In Table 5.1 we review the main features of the analyzed systems.

We built the software network associated to every system, where the nodes may be classes or files and the edges are relationships among classes, like dependencies, inheritance, composition and the like. We recovered the classes and files associated to each code change, namely to each "id" inside software repositories, by crosschecking information from Bug Tracking Systems (BTS) and Software Configuration Managers (SCM). Then we analyzed the classes or files associated to every issue "id" by identifying their positions

| Systems | n. CUs | n. Issues |
|---|---|---|
| Eclipse 2.1 | 7545 | 55537 |
| Eclipse 3.0 | 10288 | 75824 |
| Eclipse 3.3 | 15439 | 97093 |
| NetBeans 3.2 | 3346 | 25098 |
| NetBeans 3.3 | 4383 | 30995 |
| NetBeans 3.4 | 6264 | 37080 |
| NetBeans 4.0 | 9317 | 47128 |

Table 5.1: Main features of the analyzed releases of Eclipse and NetBeans: number of CUs and number of Issues.

and connections inside the software network.

Since the present analysis is aimed at understanding how and to which extent software defects can spread inside a software systems, and whether we can find relationships among defective modules, where by modules we refer to the Java CUs, we checked the tendency of these modules to form clusters. To verify if there is a net tendency of CUs to form clusters, we carried out a comparison between the number of clusters formed by CUs reporting any Issue and the average number of clusters formed by selecting randomly a number of CUs equivalent to the number of defective ones. We built the software network of each subsystem by associating the nodes to each CU and the links to relationships among them.

Then we detected the clusters formed by CUs affected by the same Issue. We define a cluster of nodes at distance $d$, inside the software network, as a set of nodes such that there is at least one path of length $d$ between each pair of nodes in the set. In this preliminary analysis, we considered the case when $d = 1$, which means that the cluster is a connected subgraph, i.e, every node inside the cluster is connected with at least another node in the same cluster. Therefore, given a set of $n$ nodes, the number of clusters formed inside this set can be univocally determined. This number varies from 1, when all the nodes are connected, to $n$ when all the nodes are isolated. It has to be pointed out that the links between CUs are considered as undirected, since clusters do not depend on the direction of edges.

We computed the number of clusters formed by sets of CUs affected by the same Issue, then we computed the average number of clusters for all the obtained cases. We then compared this number with the number of clusters obtained by a random selection of CUs. We selected random sets of $n$ CUs, with $n$ going from 2 to the total number of CUs in the system, and for each of these sets we computed the number of clusters at distance 1. We repeated this process 1000 times for each $n$ in order to get relevant statistics, and computed the average number of clusters for each random sampling. In this first step all system CUs were involved, regardless if they were affected by some Issue or not.

## 5.3 Results

In Figures 5.1 and 5.2 we report the scatterplots with the results of this analysis. For each system the plots at the top are relative to the issues, whereas those at the bottom are relative to the bugs. The blue dotted line represents the mean number of CUs for the random case and the red dotted line refers to the real case, namely it refers to the CUs that are affected by the same Issues. The random selection always results in a linear growth of the average number of clusters, since the latter is proportional to the number of selected CUs. The average number of clusters formed by CUs affected by the same Issue is systematically lower than the random case and it does not increase considerably as the number of CUs increases. This implies that CUs affected by the same Issue tend to be connected to each other.

In order to better explain this result we will now focus on two specific releases of the two systems, NetBeans 4.0 and Eclipse 3.3, considering both the cases of Issues and Bugs. The plot of the average number of clusters vs the number of CUs for Eclipse 3.3 is reported in Fig. 5.3, both for randomly selected CUs and for CUs affected by Issues. The plot for randomly selected CU can be split in three parts as $n$ increases: the initial linear growth suggests that for relatively low values of $n$, around 4% of system size, the CUs are mainly disconnected since the number of clusters and the number of CUs are nearly the same. When $n$ increases, CUs start to be more connected and the plot bends, then the number of clusters decreases after the maximum. This is due to the fact that after each random selection of CUs, we pick some of them which are already connected with CUs belonging to clusters extracted in the previous step. The tail of the distribution can also give some interesting information on the type of network. The number of clusters corresponding to the total number of CUs in the system is different from 1, meaning that there is not one single giant component, namely there are some isolated CUs or groups of CUs.

In the plot of Fig. 5.3 we report the behavior for CUs affected by Issues in comparison with the random plot, but since the scale of the former phenomenon is lower than that of the latter, we need to zoom the area around the origin of the axes. In the case of CUs affected by issues, the number of clusters is always below a threshold, which is very low compared to the linear trend of randomly selected CUs. This threshold is around 5 clusters, showing that also large sets presenting the same Issues tend to form a small number of clusters. In Fig. 5.4 we report the same plot for the system Netbeans 4.0: the behaviour is similar to the previous case, but for the randomly chosen CUs the plot does not decrease linearly after reaching the maximum. For NetBeans the number of clusters formed by all the CUs in the system is close to the maximum, around 800. This shows that Netbeans has more isolated or disconnected CUs than Eclipse. In the zoom panel we again note that while the random plot is linear, the clusters formed by defective CUs are below a threshold which is around 5. Thus, for both systems, CUs affected by Issues tend to be strongly connected, forming clusters inside the global software network. Since software defectiveness is related to bugs, we also performed the same analysis on a subset of the Issues which can be classified as bugs, as explained in 5.2. The behavior is similar to that of the issues. We report our results on bugs in plots 5.6 and 5.5.

## 5.4 Conclusions

We have presented an original approach to understand if Java CUs affected by Issues are more connected with each other compared to randomly chosen classes or files inside the systems. This approach relies on extracting Issue Ids, containing information about bug fixes, enhancements and the like, from repositories such as SCM and BTS, where developers record all changes to the code. We analyzed several releases of two popular OO software systems, NetBeans and Eclipse. We performed a statistical analysis for both the software systems by comparing clusters formed by randomly selected CUs with those formed by CUs affected by the same Issue. We found that the latter CUs typically form a small number of clusters if compared to the number of clusters formed by randomly selected CUs, meaning that they are strongly linked to each other. The same result has been obtained also for CUs affected by bugs.

We believe that our results can help developers to distinguish and decide which other classes or files are good candidates for corrective or maintenance operations, once a class or a file has been chosen for the same purpose. Moreover, since bug fixes generally improve software quality, the clustering properties of buggy or defective files could be also related to software quality. Additionally, it is worth to point out that software defects of different systems have different tendency to form clusters. This tendency could be an helpful indicator while investigating the defects capability of a software system.

Our analysis is preliminary and could be extended to detect more pre-

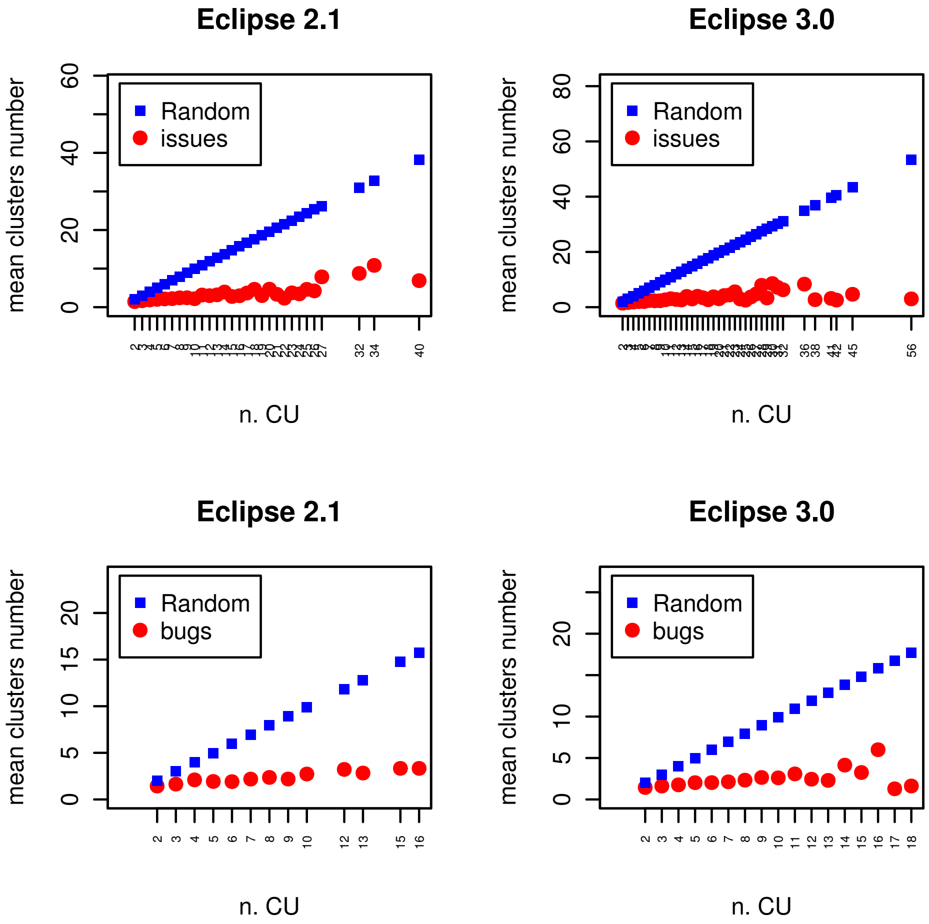


Figure 5.1: Plots of the mean number of clusters of CUs affected by Issues (in red two scatterplots at the top) and Bugs (in red two scatterplot at the bottom) vs the number of randomly selected CUs (in blue in both scatterplots) for Eclipse 2.1 and 3.0.

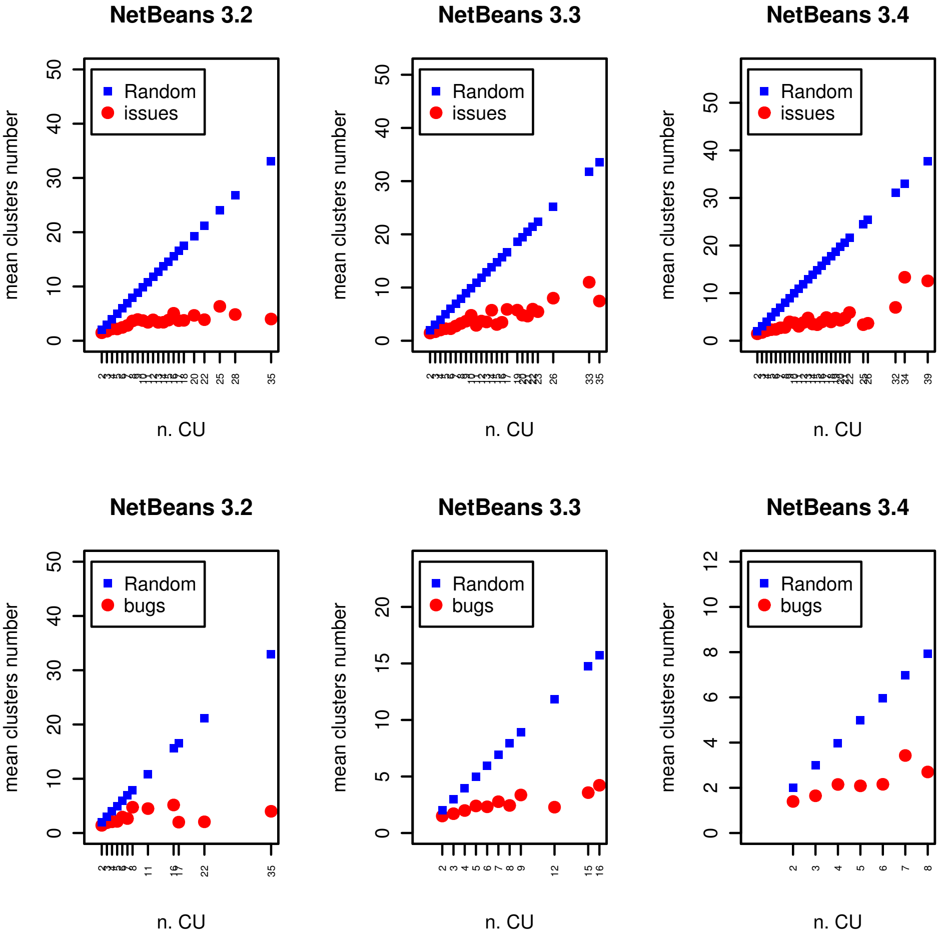


Figure 5.2: Plots of the mean number of clusters of CUs affected by Issues (in red, the two scatterplots at the top) and Bugs (in red, the two scatterplots at the bottom) vs the number of randomly selected CUs (in blue in both scatterplots) for Eclipse 2.1 and 3.0.

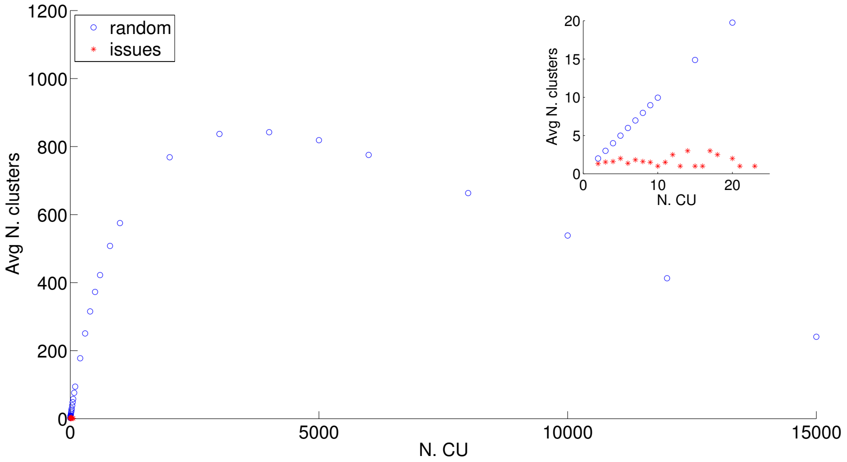


Figure 5.3: Plot of the mean number of clusters vs the number of randomly selected CUs (in blue) for Eclipse 3.3. The red dot at the origin of the axes represents the distribution of the mean number of clusters vs the number of CUs affected by the same Issue. The panel shows a zooming of the area close to the origin of the axes, where the comparison between the two statistics is more evident.

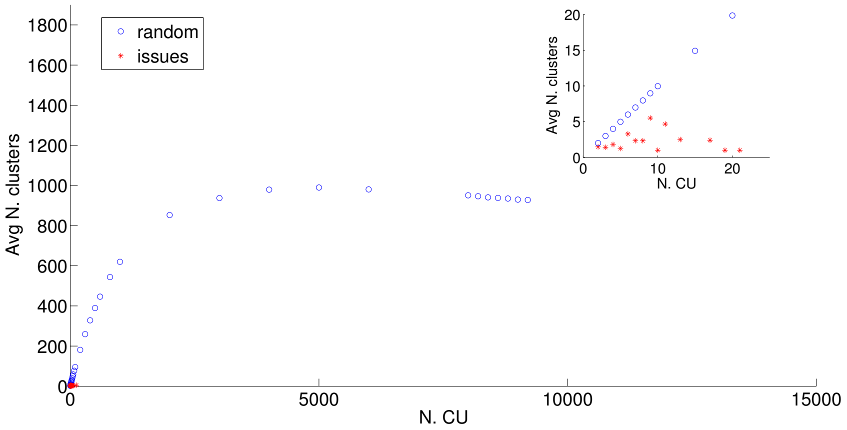


Figure 5.4: Plot of the mean number of clusters of CUs affected by issues vs the number of randomly selected CUs (in blue) for NetBeans 4.0. The red dot at the origin of the axes represents the distribution of the mean number of clusters vs the number of CUs affected by the same Issue. The panel shows a zooming of the area close to the origin of the axes, where the comparison between the two statistics is more evident.

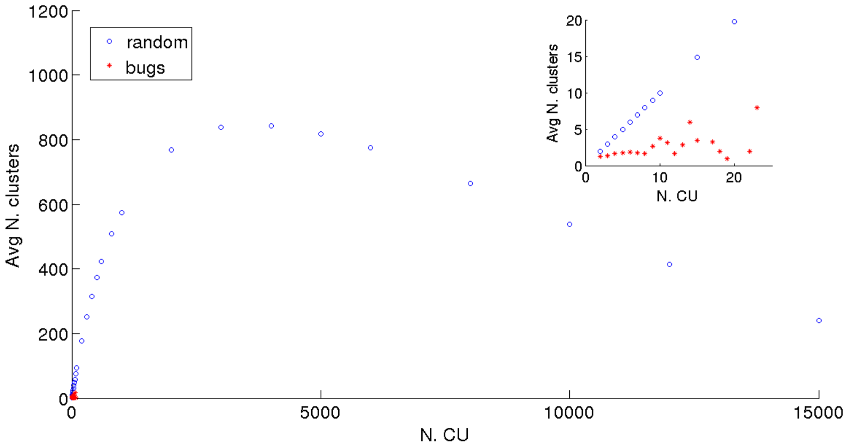


Figure 5.5: Plot of the mean number of clusters of CUs affected by bugs vs the number of randomly selected CUs (in blue) for Eclipse 3.3. The red dot at the origin of the axes represents the distribution of the mean number of clusters vs the number of CUs affected by the same bug. The panel shows a zooming of the area close to the origin of the axes, where the comparison between the two statistics is more evident.

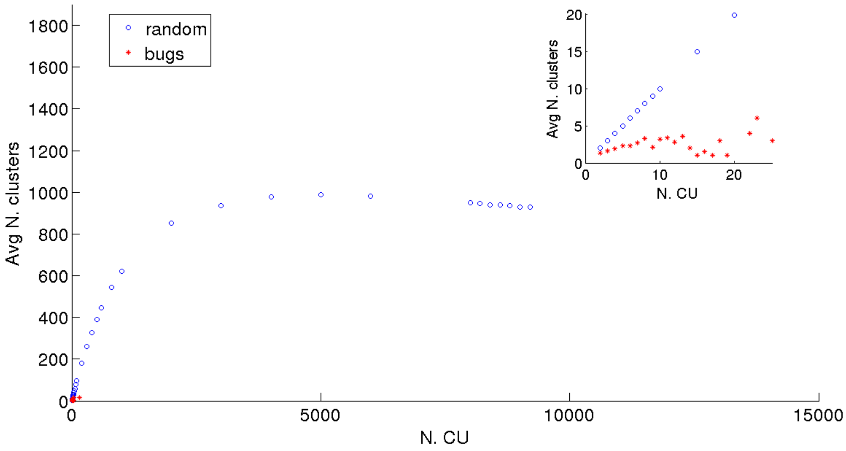


Figure 5.6: Plot of the mean number of clusters of CUs affected by bugs vs the number of randomly selected CUs (in blue) for NetBeans 4.0. The red dot at the origin of the axes represents the distribution of the mean number of clusters vs the number of CUs affected by the same bug. The panel shows a zooming of the area close to the origin of the axes, where the comparison between the two statistics is more evident.

cisely the defective or buggy CUs among those affected by Issues in order to distinguish which files among those affected by a specific Issue are more inclined to form clusters. We could also verify whether corrective activities change the cluster structure of a software network. If this hypothesis could be verified, it could be possible to make a prediction about which classes need bug fixing operations, or understand if some classes were fixed, by looking at the cluster structure in the proximity of the involved classes.

# Chapter 6

# Inheritance in Python Software Systems

## 6.1 Introduction

Inheritance is one of the major features offered by Object Oriented (OO) programming with respect to other programming paradigms. While it is visible in many ways, for example appearing in many design patterns [45], there are also warnings on its misuse or overuse. Being so widely employed it seems likely that software design is strongly influenced by the extent and the ways inheritance is adopted by software developers. Different programming languages provide inheritance in different ways and so any investigation about the use of inheritance must consider that the results can be strongly influenced by which programming language is involved.

So far there have been several studies on the effects of inheritance on software maintenance and defectiveness [31, 17, 57, 95, 24, 21, 13, 16, 63]. In this paper, we are interested in the patterns of use of inheritance — what decisions programmers make with respect to whether they use inheritance, and how they use it. This question has already been answered to some extent for Java by Tempero et al. [98], but not for other languages.

In particular, there has been little study of Python, despite the fact that it is becoming more and more popular among software developers and it is largely adopted both in the academia and in industry [34, 52, 67]. For example, TIOBE index[1] and Popularity of Language Index (PYPL)[2] provide

[1] http://www.tiobe.com

[2] http://pypl.github.io/PYPL.html

a measure of Python's large popularity. StackOverflow[3] and GitHub[4] shows how developers largely use Python. OpenHub[5], a popular public directory of Free and Open Source Software, at the moment reports around 68.000 Python projects, confirming the interest toward this language, from the open source community. In the academic field, Scopus[6], one of the most important bibliographic databases, shows an increasing number of scientific works using Python language for performing scientific computing or presenting software developed in Python.

The present Chapter illustrates a research aimed at answering the following question: "How do Python programs use inheritance?". The answer naturally depends on how Python supports inheritance, but also on how effectively developers make use of inheritance in practice. Our study is descriptive, meaning that we are interested in the way Python programs are structured in the real world.
One major difficulty in performing such kind of empirical research, including an investigation on the practical use of inheritance, and on the interpretation of results is due to inaccuracies, ambiguities and uncertainties on the data sets retrieved by researchers in empirical software engineering which especially affect reproducibility of the studies. By and large, empirical research in software engineering often lacks of reproducibility and presents ambiguity of results due mainly to inaccuracies and uncertainties on the empirical data set used for the analysis. As a consequence, the usefulness of empirical research conducted on such data, both for scholars and practitioners, is largely reduced because it is difficult to compare the findings based on different datasets. The research efforts can be significantly improved by using a representative collection of software systems taken from real world as a benchmark, in order to enable reproducibility and comparison of the results. Presently, there exists a curated collection of software, the Qualitas Corpus [96], but it is limited to Java software systems. With regards to Python systems, there is nothing to support the research this way, albeit Python is a programming language of wide adoption both in academia and industry [34].

In order to perform such analysis and in order to guarantee reproducibility of results, we created and collected a *curated corpus* of well-known widely-used Python programs following the guidelines of the Java Qualitas Corpus (JQC) [96]. As with the JQC, such a curated collection of Python software systems taken from real world can be used as a benchmark in order to enable

[3] http://stackoverflow.com

[4] https://github.com

[5] https://www.openhub.net

[6] http://www.scopus.com

reproducibility and comparison of the results. It provides a representative sample of 51 popular Python software systems. After having downloaded all the systems, we computed several metrics. Additionally we investigated the internal structure of each system and collected meta-data in order to provide additional information to allow reproducibility of the results. Our final goal is to create a curated corpus of Python software similar to the Java Qualitas Corpus (JQC) [96]. In the meantime we are releasing the dataset of metrics associated to the software collected so far and we are confident that this dataset might be useful for researchers interested in conducting empirical studies on Python systems. This dataset and its documentation is available on the Promise Repository [4] at the following url:

`http://openscience.us/repo/code-analysis/python.html`

as are details for acquiring the corpus.

We then performed a complete analysis of the practical use of inheritance in Python systems, using the metrics suite proposed in [98], investigating some important features of the accepted practice about inheritance in Python and the related metrics.

## 6.2 Related Work

The first work reporting measurements on inheritance is that of Chidamber and Kemerer [20] using their metrics Number of Children (NOC) and Depth of Inheritance (DIT). The measurements came from two system written in C++ and Smalltalk respectively. Several authors tried to assess the validity of the Chidamber and Kemerer's study. Daly et al. [31] and Harrison [57] came to the conclusion that deep inheritance may have a negative impact on maintenance activities. Two large studies were conducted on different languages. Succi et al. [95] conducted a study on two large datasets of 100 Java and 100 C++ software systems and Collberg at al. worked on 1132 Java systems [24]. Several studies have been also conducted on Python code [35, 78]. Some researchers focused their attention on the relationship between inheritance metrics and system defectiveness (Basili et al. [13] and Briand et. al [16]). Tempero et al. performed studies on the JQC in order to understand how much inheritance is used in practice [98] and with which purpose [99]. The work presented here is a replication of the former, but on Python. In the earlier work, a number of different metrics were presented. Many were variations of Chidamber and Kemerer's original NOC and DIT metrics. They noted that the NOC and DIT metrics (and variants) provide measurements only for single types, and so provide only a local view of how inheritance is used. They proposed metrics that measure some aspect of how much inheritance is used in the full system, and in doing so provide a global view of the use of inheritance. They applied the different metrics to 93 open source Java applications and presented some of the results. A key finding was that around three quarters of Java types use some form of inheritance.

An issue with some studies involving analysis of software is the difficulty in replicating it due to insufficient information being provided on what exactly was analyzed. Crucial information, such as the version of the system that was analyzed, or which source files were included in the analysis, are often not available. On the other hand, reproducibility, reliability and applicability of results or findings can be significantly improved by the use of datasets of software systems. For example, some of the works already cited were performed on dataset of software written in Java (Succi et al. [95], Collberg at al.) or C++ (Succi et al. [95]). However, the datasets used for these studies are often not publicly available, and for this reason it is difficult to reproduce the results. A solution to this is to provide a *curated* corpus of software and the associated datasets, including meta-data.

One example of this is the Java Qualitas Corpus (JQC) [96]. JQC's main aim is to enable reliable and reproducible empirical studies of Java software. It provides, as well as the code, metadata describing the contents of the corpus and the criteria for inclusion. A similar project is the Qualitas.class [100] whose main goal is to enable the research on compiled Java of systems hosted on JQC. Another repository of Java Software systems for empirical studies is the Helix Data set [90]. Presently it includes more than 1000 releases of 40 systems and includes also meta-data and data about software defectiveness. In general, there are actually many available datasets. The Mining Software Repositories conference (MSR), since 2013 [114], hosts a data-showcase session where researchers are called to illustrate and share their datasets. Additionally, many datasets of the Mining Challenge presented in the MSR conference are available, for example in the Promise Repository website [4, 70] where a number of software engineering research datasets are hosted. Moreover, there are also some infrastructures for empirical studies on software engineering. The DaCapo benchmark [14] and the New Zealand Digital Library project [111] collect open source Java software. The Software-artifact Infrastructure Repository (SIR) [37] contains software systems written in different languages (Java, C, C++, and C#).

However, with the exclusion of the first three reported cases, the cited datasets can not be described as curated, neither they report metrics for the collected systems. Additionally, excluding few exceptions such as the work of Farah et al. [39] and the GHTorrent dataset [50], there is little data available for Python systems.

## 6.3 A Python Corpus

This section reports a description of the Python Corpus we built in order to extend the JQC by including a curated collection of Python software systems.

### 6.3.1 Dataset Construction

The main goal of this work is to provide a dataset of metrics computed on a curated corpus of software written in Python, in order to enable researchers to perform empirical studies on Python systems. To collect the systems we followed the guidelines adopted for the JQC [96] relying also on our knowledge of the Python community. In general, we considered systems written in Python that are publicly available in order to allow researchers the access to their code. Moreover, we considered systems whose repositories are hosted on GitHub, to exploit some GitHub features for further studies. For each system of the corpus, we downloaded the latest available release from its official repository. In order to have a representative base of code, we considered systems with more than 50 Python files. Moreover, these files had to represent more than 50% of the system files. We faced two main issues when retrieving data for our corpus. The first one is related to the fact that the source code retrieved from repositories usually contains "infrastructure code", i.e., code that is related to the development, management, installation and test. Some of this code is sometimes included in the official release (e.g, examples, demo, etc.) whereas some other usually is not (e.g. test code, etc). This code is not a part of the system, but it is provided to help the user to get the most out of it. Consequently, taking into account this code while computing the metrics could bias the results. For this reason, we decided not to consider test code, examples, code associated to documentation and the like. The second issue is related to the fact that some systems were not written only in Python, but presented a significant part of the code written in another language. In order to deal with this not-Python code we made a distinction between not executable (Xml, Html, Css, etc.) and executable code (e.g. C, Javascript, etc.).

- When we found that a project had a significant part of its code that is not executable, unless this amounts to the majority of the available code, we included it in the corpus.

- In case we found that a significant part of the executable code was written in languages different from Python, we discarded the system.

There is a third case, where we found that a significant part of the executable code is written in a language other than Python, but that code is somehow mandatory for the domain of the application. It is the case of scientific libraries like `matplotlib` that delegate some computations to most efficient C language functions. In this case we included the system in our corpus, unless the not-Python code was the vast majority of the available code.

### 6.3.2 Dataset Description

The present corpus is composed by 51 popular Python software systems, whose names [7] are reported in the frame on Fig. 6.1. Table 6.1 illustrates the representativeness of the dataset, reporting the different domains along with the number of systems belonging to each domain. The corpus includes open-source public available systems, in order to allow researchers to access to the code. We focus mostly on systems hosted on GitHub to exploit some GitHub features in future work.

The corpus includes the latest available release of each system downloaded from its official repository. We analyzed the source code in order to remove the "infrastructure code", namely the code used during the development and for management, test and installation purposes, that sometimes is included in source code available on the repositories. Including this code would have biased the metrics' computation, since this code is not actually a part of the system, but it is included to support the user to better and easier use it.

Additionally, while building the corpus, we found that a significant part of some systems is written in languages other than Python. In these cases, we included only systems that contain a significant part of non executable code (XML, HTML, CSS, etc.), considering those that contain a meaningful part of executable code (e.g. C, javascript, etc.) just when it was mandatory for the Python system to properly work. The latter is the case of some scientific libraries like `matplotlib` that rely on code written in C to perform some computations more efficiently. The corpus includes also several code metrics computed with Understand [103], and it is available with its documentation on the Promise Repository [4] as are details for acquiring it[8].

To provide an understanding of the representativeness of the corpus, we provide some summary statistics. Systems' size spans a large range, from 48 to 5587 classes and from 2626 to 687058 lines of code, as it is shown in Tab. 6.2. Figure 6.2 displays also the boxplot of the number of classes per system, showing a non uniform distribution with a large fraction of small systems and fewer large systems, and five outliers (two overlap each other)

[7] Names include the systems' release number or the release tag taken from their respective repositories.

[8] http://openscience.us/repo/code-analysis/python.html

Table 6.1: Domain representation of Corpus

| **Domain** | **No** |
|---|---|
| Additional Development Package | 5 |
| Applications | 21 |
| Graphic framework | 2 |
| Math library | 4 |
| Scientific package | 9 |
| UI framework | 2 |
| Web Application | 1 |
| Web Framework | 7 |

“Astropy v1.0rc1” “Biopython biopython-165” “BuildBot v0.9.0-pre” “Calibre v2.23.0” “CherryPy 3.5.0” “Cinder 2015.1.0b3” “Django 1.7a2” “Emesene v2.12.9” “EventGhost v0.4.1.r1640” “Exaile 3.4.4” “GlobaLeaks v2.60.63” “Gramps gramps-2.90.0-beta” “Getting Things Gnome! 0.3.1” “OpenStack - heat 2015.1.0rc1” “IPython rel-3.0.0” “Kivy 1.9.0” “OpenStack - magnum 2015.1.0b2” “GNU Mailman 0.6c9” “OpenStack - manila 2015.1.0b3” “Matplotlib v1.4.3” “Miro v6.0” “NetworkX networkx-1.9.1” “OpenStack - neutron 2015.1.0b3” “Nova 2015.1.0b3” “NumPy 1346-g3c5409e” “Pathomx v3.0.0a” “Python Imaging Library 2.8.1” “Pip 6.1.1” “Plotly 1.6.12” “Portage v2.2.18” “Pygame 1.9.1” “PyObjC 3.0.4” “Pyramid 1.5” “PYthon Remote Objects 4.35” “Quod Libet 3.0.1” “SABnzbd 0.7.11” “Sage 6.5” “scikit-image v0.11.0” “scikit-learn 0.16-branching” “SymPy sympy-0.7.6” “TurboGears2 tg2.3.4” “Tornado v4.1.0” “Trac 1.0“ “Tryton 3.4.0” “Twisted 15.0.0“ “Veusz veusz-1.22” “VisTrails v2.2-pre” “VPython 1.5” “web2py R-2.10.3” “wxPython 1.5” “Zope 2.13.22”

Figure 6.1: List of the analyzed system

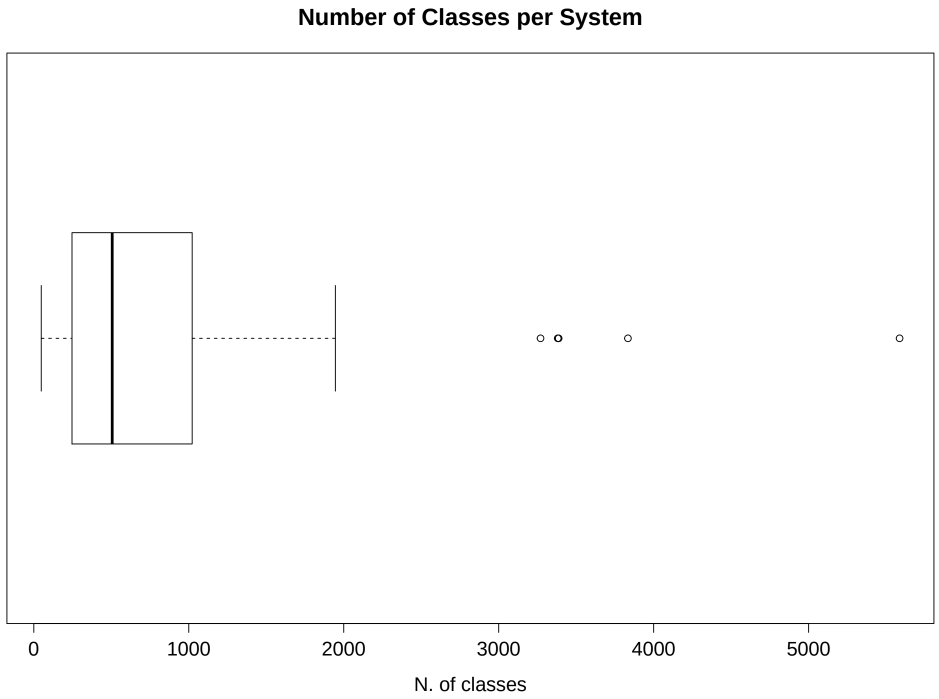


Figure 6.2: Boxplot of number of classes per system

with more than 3000 classes, namely Calibre, EventGhost, Sage, VisTrails and WxPython.

The largest systems are `Sage` (maximum number of files) and `WxPython` (maximum number of classes), which are respectively a library for scientific computing and a cross-platform GUI library. The smallest systems are `Pyro4` (minimum number of files), a library for remote object management and `plot.ly` (minimum number of classes), a library for creating browser-based graphs. `Sage` and `Pyro4` are respectively the largest and the smallest system even if one considers the Lines of Code (LOC) and Non-Comment Lines of Code (NCLOC) metrics.

The presented dataset contains metrics associated to a corpus of 51 Python systems, belonging to different application domains as reported in Table 6.1. Along with several metrics, for each system we report meta-data to provide information about the systems. The meta-data are the following:

**System:** a unique identifier for the system.

**Description:** a description of the system.

**Sysver:** a unique identifier for the system version

| Metric | Min | Max | Mean | Median |
|---|---|---|---|---|
| N. Files | 22.0 | 1590.0 | 357.4 | 249.0 |
| N. Classes | 48.0 | 5587.0 | 904.0 | 506.0 |
| LOC | 2626 | 687058 | 67838 | 32471 |
| NCLOC | 2425 | 657523 | 63816 | 30717 |

Table 6.2: System size values

**Fullname:** the full name of the system.

**Domain:** the application domain for the system.

**Python Software Library Version (PSLv):** which version of the Python Standard Library the system is compliant.

**License:** license under which the system was released.

**LOC:** number of line of code for the entire system.

**NCLOC:** number of line of code (excluding comments).

**N_Files:** number of files.

**N_Classes:** number of classes.

**Url:** url of the official site.

**Repo_url:** url of the official repository.

**Download_date:** date of the download from the repository.

All this data are reported in a comma separated values (CSV) file. We considered three different kind of metrics:

- Size/Volume metrics;
- Complexity metrics;
- Object-Oriented metrics.

Depending on which kind of metrics is considered, it is possible to obtain information about the dimension of a software system, its complexity (e.g. McCabe Cyclomatic Complexity) and about object oriented properties (e.g. Chidamber and Kemerer metrics). For, example, we computed 13 size metrics - including Line of Code (LOC) and Comment Line of Code (CLOC). To calculate these metrics we used Understand 3.1 (build 766) [103]. For each system, we provided metrics for 3 different levels of granularity: system, file and class level. Metrics are contained in files named following the pattern: `<system_name>-<entity_name>-<metrics_kind>_metrics.csv`. `<system_name>` is self explanatory whereas `<entity_name>` corresponds to the level of granularity analyzed and could be *File* or *Class*. On the other hand, `<metrics_kind>` corresponds to a category of metrics (volume, complexity and object oriented). For example, the object oriented metrics associated to the files of Zope are reported in a file called `Zope_file_oo_metrics.csv`. Global metrics at class and file level are reported in two files called respectively `class.metrics.global.selected.csv` and `file.metrics.global.selected.csv`. System meta data are reported all at once in a file named `system.meta.data.csv`.

### 6.3.3 Limitations

In this paper we presented a dataset of metrics associated to the first release of a curated collection of Python systems that could be used to perform empirical research. Being the first release of an ongoing work (that is meant to be maintained along the years in the future as it happens for the Java Qualitas Corpus [96]), there are several limitations. The most important is that at the moment the dataset contains only the last version of the collected systems. A second limitation is that the systems have been retrieved from repositories, and we have not yet performed a thorough analysis of the internal dependencies with third party libraries hosted in repositories or released along with the source code. Thus, there is the possibility that the code base might differ from that available from other sources (i.e. Python Package Index [2]). At the moment, there are some application domains that are under-represented (e.g. Web Application): as future work we plan to put effort in order to increase the number of systems in these categories and, in general, enhance the corpus representativeness. Despite the mentioned limitations, we are confident that the provided data can be an interesting source of information for researchers interested in investigating Python systems properties and development.

### 6.3.4 Research Opportunities

Even if Python is a widespread programming language both in academia and in industry, there are few empirical studies on Python software compared to other languages, i.e., Java. For this reason there are many research opportunities for scholars, starting from replication studies that can be performed on Python systems, in order to assess if the already found results apply also to Python. The kind of studies that might be supported by this dataset are mainly those that involve a static analysis of Python code. In this perspective there are many specific features of Python language that are not available for Java software (e.g. dynamic binding, multiple inheritance, etc.) that could be investigated to understand how they are used by developers. A comparison of the usage of some Python features in a specific application domain with their counterpart on Java could lead to interesting results. The provided metrics can be compared with other measurements of different nature [28] taken from the associated repositories (e.g., social metrics) or the corresponding Issue Tracking Systems (e.g. number of issues, etc.), or for patterns detection [36, 47]. Since we provided a categorization for the software systems in the corpus, software metrics can be also used to investigate specific differences among systems, depending on the application domain. Additionally, it could be interesting to investigate if and how metrics change among different components of the same system. It could also be interesting to study different properties of systems that share the same application domain but are written in different languages.

## 6.4 Methodology

### 6.4.1 Modelling Inheritance

We model inheritance in Python by adapting the model proposed by Tempero et al. [98]. They used a Directed Acyclic Graph (DAG) where types are vertices and edges indicate some form of inheritance relationship between the types directed from the child type to the parent type. As Python does not have the Java equivalent of interface, our model only has one kind of vertex (class) and one kind of edge. As Python allows multiple inheritance, the result is still a DAG (not a tree). We exclude the `__builtin__.object` class and all the *exception* classes, as developers have no choice but to use inheritance with these. Only edges that are incident on at least one system class are included, that is, there are no edges between Standard Library (SL) classes. In order to retrieve the DAG we parsed the source code using Understand [103], a well-known software for code analysis.

### 6.4.2 Inheritance Metrics

We distinguish between local and global inheritance metrics. Local metrics provide measurements for a single class whereas the global metrics provide measurements on the overall system. We use the subset of the metrics proposed by Tempero et al. appropriate to Python, as described below.

## Local metrics

- NOC : Number of Children — the number of classes inheriting directly from the class, that is, the number of vertices with an edge leading to the vertex representing the class.

- NOD : Number of Descendants — the number of classes inheriting transitively from the class, that is, the number of vertices with a path leading to the vertex representing the class.

- NOP : Number of Parents — the number of classes the given class inherits directly from, that is, the number of vertices reachable via an edge from the vertex representing the class.

- NOA : Number of Ancestors — the number of classes the given class inherits transitively from, that is, the number of vertices reachable via a path from the vertex representing the class.

- DIT : Depth of Inheritance Tree — the longest path from a class to all its ancestors

## Global metrics

- DUI : Defined Using Inheritance — the percentage of classes using inheritance, namely the fraction of children classes, that is, the percentage of vertices that have an outgoing edge.

- IF : Inherited From — the percentage of classes inherited from other classes, namely the fraction of parent classes, that is, the percentage of vertices representing system classes (not SL or third-party classes) that have an incoming edge.

| Metric | Max Metric Value | System |
|---|---|---|
| NOA | 419 | wxPython |
| NOP | 317 | wxPython |
| NOC | 376 | VisTrails |
| NOD | 1829 | wxPython |
| DIT | 11 | buildbot |

Table 6.3: Maximum values for local metrics

## 6.5 Results

### 6.5.1 Local Metrics

Table 6.3 shows the maximum values for the local metrics. Figure 6.3 reports the frequency distributions of NOA, NOP, NOC and NOD metrics for all the classes in the entire dataset. The top two plots report the metrics from a bottom-up perspective (NOP and NOA), namely for classes inheriting from others, whereas the bottom ones report the metrics from a top-down perspective (NOC and NOD), for classes inherited by other classes. All distributions consistently show a large number of classes with low values of the metrics, and a decreasing trend, roughly linear in a log-log scale, where the number of classes decreases when the metric values increase. But there is also a scattered right tail for large values of the metrics, suggesting a complex pattern.

In order to check for the existence of some power-law relationship among the metrics and the number of classes we performed a best fitting analysis according to the method proposed by Clauset et al. [23]. The results are displayed in Fig. 6.5 where the values for the Kolmogorov-Smirnov test (KS) are also reported, in order to evaluate the best fitting quality. Considering a level of confidence of 0.10, the KS value for each fitting is quite low (significantly under the chosen value) meaning that none of the distributions of local metrics are following a power-law. All figures indicate cut-off values on the right tail, where the distribution suddenly changes shape.

Figure 6.4 reports the distribution of DIT values over the entire dataset. The max DIT is 11, and the distribution is strongly skewed with very few classes having DIT larger than 7-8.

### 6.5.2 Global Metrics

Figures 6.6 and 6.7 report the histograms for the distributions of the DUI and IF metrics for all the systems. DUI values are spread along all the

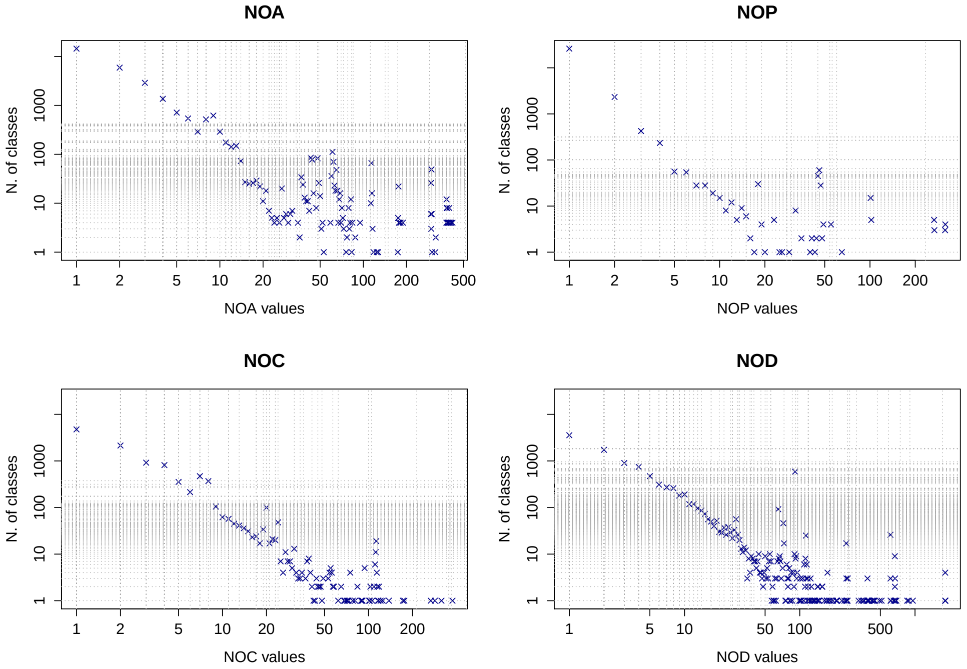


Figure 6.3: Frequency distributions of local metric measurements

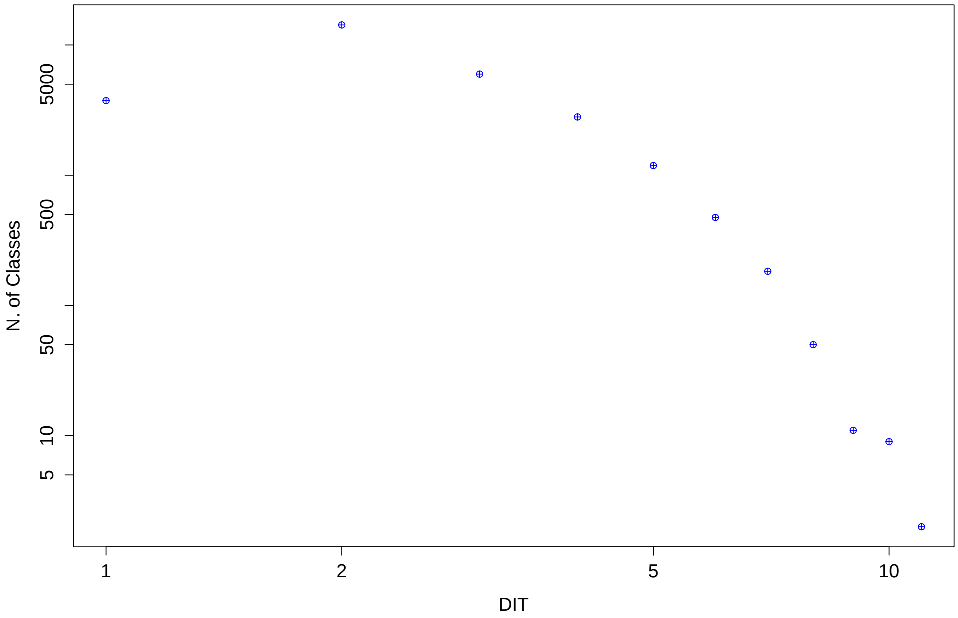


Figure 6.4: Frequency distribution for DIT

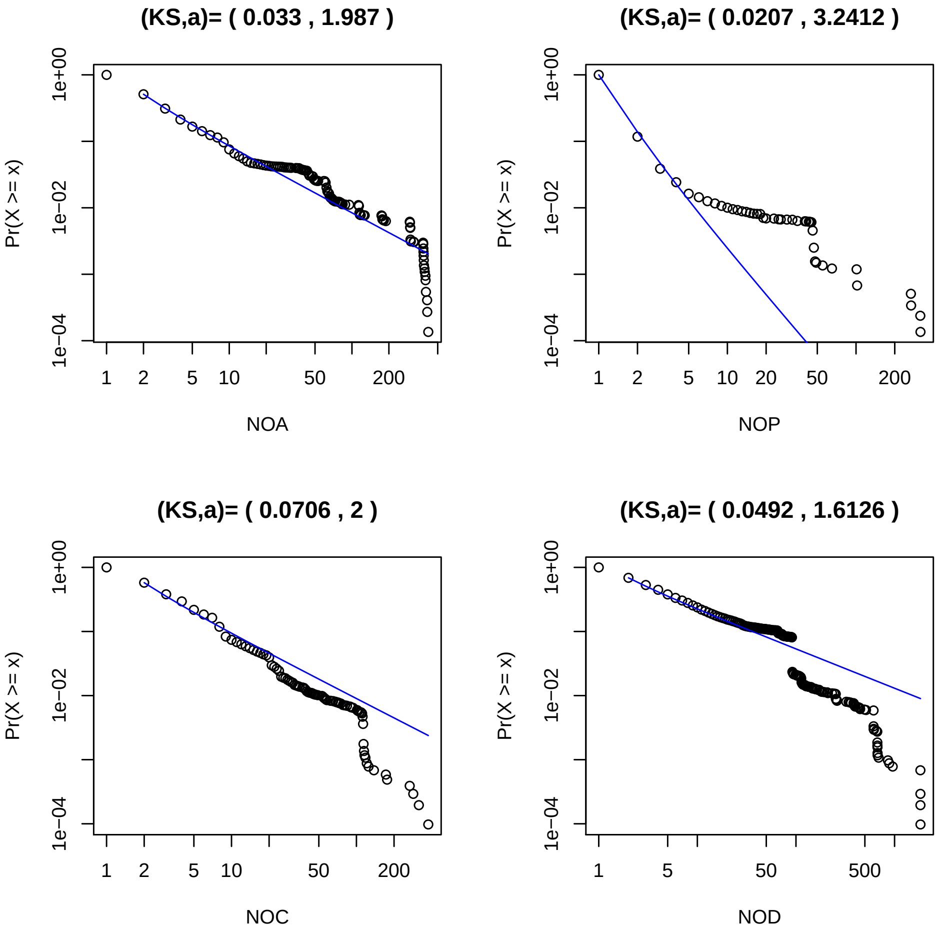


Figure 6.5: Complementary cumulative frequency distributions (CCDF) for local metrics

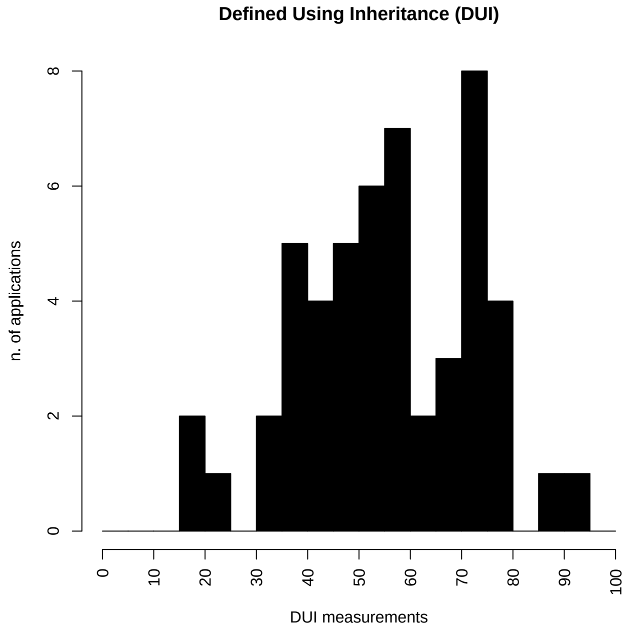


Figure 6.6: Frequency distribution for the DUI metric

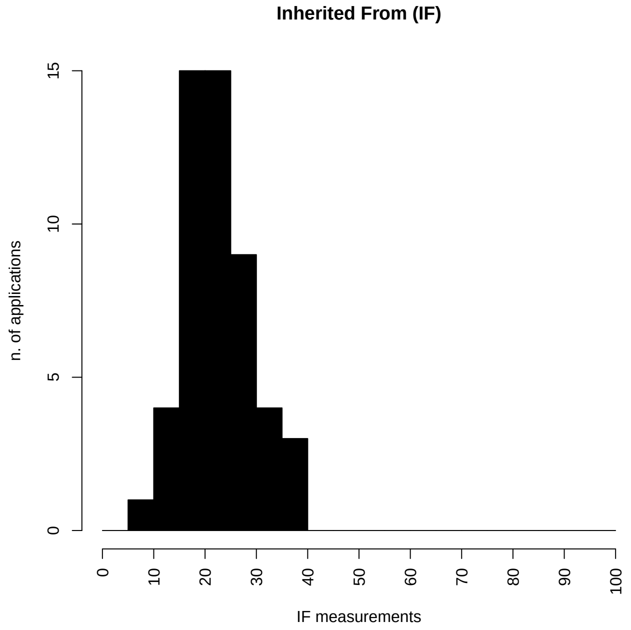


Figure 6.7: Frequency distribution for the IF metric

possible range, from 15% to 95%, meaning that the fraction of classes defined using inheritance can vary largely from one system to another. Although the distribution of values for DUI is somewhat irregular, the majority of systems have values that are in the high part of the range. The median is 55%. This means that a significant number of the classes are defined using inheritance.

On the contrary the range for the IF distribution histogram is much narrower ranging from 5% to 40% and with a peaked shape around 20%, meaning that the fraction of classes inherited from other classes is roughly the same for all the examined systems. The median is 22%. This suggests that, even if we have a lot of classes defined using inheritance, the inherited classes are a small percentage of the available classes.

## 6.6 Discussion

In comparison to the study of Java, we see broadly similar results but some differences in detail. As with Java, the Python local metric measurements have distributions that are somewhat similar to power-law distributions. In particular, measurements for relationships that are from a top-down perspective (NOC and NOD) are much closer to a power-law (straight line on a log-log scale) than those from a bottom up perspective. Unlike the Java results, the Python measurements show clear departures from a power-law or anything like it. Figure 6.5 shows power-law fits performed on the complementary cumulative distribution with the method presented by Clauset et al. [23]. The results indicate that the distribution does not fit a power law distribution. The reported values of KS are in fact quite low.

The similarity between the two languages suggests that such distributions might be expected in other languages. We should note that such distributions were also hinted at in early empirical data [20]. The differences need to be understood. It could be that they are an artifact of the corpora used (in particular, the Python corpus is smaller), or it could be an indication of something different in how inheritance is used in the respective languages.

Multiple inheritance does not seem to play a marginal role in Python. As we can see in the complementary cumulative frequency plot in Fig. 6.5, about 10% of the classes have 2 or more parents. Again looking at Fig. 6.5, there is a point where there is a noticeable drop in the values. This could be due to the presence of some outliers, but there might be also another explanation. The graph of the systems formed by the inheritance relationships among classes is widely unconnected, with a relevant part of it formed by smaller directed graphs and a few larger graphs that do not constitutes a giant component. Being the maximum connected component for these graphs relatively small,

Table 6.4: Spearman correlation between mean values of the metrics and size (n. of classes)

| | size | mean.NOC | mean.NOD | mean.NOP | mean.NOA |
|---|---|---|---|---|---|
| size | 1.000 | 0.676 | 0.678 | 0.730 | 0.715 |
| mean.NOC | 0.676 | 1.000 | 0.881 | 0.774 | 0.800 |
| mean.NOD | 0.678 | 0.881 | 1.000 | 0.715 | 0.944 |
| mean.NOP | 0.730 | 0.774 | 0.715 | 1.000 | 0.795 |
| mean.NOA | 0.715 | 0.800 | 0.944 | 0.795 | 1.000 |

for each system there is no chance to find a number of children over a certain threshold, that fixes an upper bound for our distribution in correspondence of the the cut-off values.

As we can see from Table 6.4, the very high value of the correlation among the NOA and the NOD metrics suggests a sort of symmetry in the usage of inheritance between classes inheriting from and classes inherited by. In fact such very strong correlation, which is computed on the averaged values of NOA and NOD for the entire systems, means that when the average NOD is low so is the average NOA and vice-versa.

Considering the role the multiple inheritance mechanism provided by Python, we have devised two extreme situations. In the first, the system DAG is slim and tall, for example a linear direct graph. In this case the situation is completely symmetric between ancestors and descendants, and the addition of one more child involves also the addition of one more parent. But the limited values of DIT suggest that this mechanism must stop after a few steps. Nevertheless, as already reported, examining the inheritance graphs we found many unconnected subgraphs corresponding to one system. This justifies the presence of a large number of smaller graphs structured in the way described above, which may account for the symmetry. In the opposite situation the graph is fat and shallow in both directions, namely the bottom-up, going from children to parents, and the top-down, going from parents to children. This means that the multiple inheritance practice is relatively diffuse among Python programmers and there are quite enough classes inheriting from many parents. This might be supported by the results reported in Fig. 6.3, where the values of NOP are not rarely larger than 10 or even more.

The results for DUI and IF are also both similar to and different from the Java results. Like the Java results, the distributions seem more normally distributed. In fact, a Shapiro-Wilk test for normality does not rule out their being normally distributed ($p \approx 0.21, 0.11$ respectively). The normality of the distribution is less pronounced for the Python DUI measurements, but this could be due to the smaller sample size. The Python medians differ from the Java medians for both DUI and IF reported by Tempero et al. (74% and 17% respectively). According to a Mann-Whitney test the differences between Java and Python for both DUI and IF was significant at the 0.01 level. This suggests that on average fewer classes are defined using inheritance in Python than in Java, but more are used in inheritance relationships. One possible explanation is that the fan-out (NOC) for Python is lower than for Java. This is not supported by the NOC measurements. Another explanation is that fan-in (NOP) is higher for Python. This is consistent with the use of multiple inheritance we noted above.

## 6.7 Threats to Validity

The present work suffers from some threats to validity, that we are reporting in the following. There are some construction threats, related to the adopted dataset. The systems have been retrieved from repositories, and even though we have discarded the classes evidently part of the infrastructure code (such as test classes, or examples) we have not performed a thorough analysis of the internal dependencies with third party libraries. Sometimes it happens that developers host also third party libraries. Moreover, it could also be the case that the code base might differ from that released in the official websites (i.e. Python Package Index [2]). Nevertheless, because of the way the corpus was developed [85] we are confident that the effects of such issues is likely to be quite small. Additionally, there are some domains that are not well represented in our corpus, which may consequently influence the results of our analysis. However this would only be the case if there are significant differences in how inheritance is used in different domains. In our manual investigation of members of the corpus we saw nothing to suggest that this is the case, and the fact that no one system dominates the maximum values (Table 6.3) supports this.

## 6.8 Conclusion

In this chapter we presented an empirical study on the use of inheritance in Python systems, replicating a previous study of Java by Tempero et al. [98] We chose the metrics from that work, we applied them and gathered measurements for these metrics from 51 Python systems that came in a variety of sizes and from a number of different domains.

We firstly described the curated collection of Python systems we built, along with the associated dataset of metrics. We illustrated the motivation that led us to build the corpus, the main issues involved in creating it, providing a description of the dataset and its limitations. We also reported some suggestions about the use of this dataset for empirical studies of Python systems. The main goal of this corpus is to provide a benchmark for empirical studies of Python systems that allows reproducibility of results and lowers the cost of experiments. This corpus is intended to be a constant work in progress [59].

In the future we plan not only to increase the number of systems available in the corpus, but also to include different versions of the same system to promote longitudinal studies. We are also considering to add information about the development process taken from the official repositories, along with information from issue tracking systems.

The second part of the present Chapter presents a study on the use of inheritance in Python software system. The study was conducted on the Python Corpus. We found that overall small measurements from (local) metrics for individual classes tended to be the majority with a roughly proportional reduction in number of classes as measurements increased. Our results suggest that multiple inheritance is quite common in Python software systems. However, unlike in other similar studies, we could not confirm that the measurements followed a power-law distribution. The complementary cumulative distributions confirm that there is not a power-law trend and display a cut-off for values over a metric-dependent threshold. We believe this behavior could be related to the presence of outliers as well as to the topology of the direct graph of the inheritance relationships among classes. Measurements for the (global) metrics that measure overall use of inheritance in a system were normally distributed. These indicated that for more than half the systems, more than half (DUI median of 55%) the classes in the systems relied on inheritance for their definition. Also, about one fifth (IF median 22%) of classes are parents of other classes. Of particular interest is that the global results are different to those for Java, with the DUI measurements for Python being on average smaller than for Java, but the IF measurements being larger. This supports the use of inheritance being higher in Python than in Java. This provides clear evidence indicating that inheritance is used differently in different languages. Our study provides another data point to help us understand how inheritance is used. Many more data points are needed to complete our understanding. This include replications of this study for other languages, and more in-depth studies of the purpose for which programmers use inheritance.

# Chapter 7

# Refactoring and Complex Networks

## 7.1 Introduction

Software systems evolve to meet new needs and often new features are added over time. It could happen that after several months and new versions, the code needs to be rewritten or abandoned or, eventually, if nothing is done on it, it will go through code decay. Software maintenance has the purpose of avoiding this by performing activities such as the addition of functionalities, but it requires lots of efforts and time. With good design and advance planning, refactoring can help in software maintenance.

According to Fowler's definition [43], refactoring consists in rearranging the internal structure of a piece of software without altering its external behavior, in order to improve code functionality and readability. It has the advantage of requiring short-term time and low work costs and allows to get long-term benefits. Refactoring is different from other activities such as rewriting or debugging code, or adding features or bug fixing. It is aimed at improving software design by making it more extensible, flexible, understandable, and at improving performance.

Since refactoring can be applied to classes which are strongly connected with each other, its impact can extend over the single class to involve other related classes. This phenomenon is studied in this Chapter using a software network approach [64, 108, 66, 77, 89], where classes are represented by network nodes and relationships among classes (such as inheritance, composition, etc.) are represented by network links.

This perspective allows to study software elements in the context of their reciprocal relationships, without neglecting the aspects that could be measured with the traditional metrics [19, 69].

Our goal is to perform a software network analysis of refactorings to understand if they are related to the network structure, in order to retrieve information which can be useful when planning refactoring activities, or to make predictions on future refactorings. To our knowledge, how and to which extent the impact of refactoring can spread over the associated software network has not been thoroughly studied so far.

In this chapter we are presenting the results of an analysis conducted on two datasets formed by several releases of a number of popular open source software systems written in Java (i.e. Ant, Jtopen, etc.). We are specifically interested to understand whether refactoring operations are applied randomly on the nodes of the software network or if they mostly involve classes that are linked together. In other words, we aimed at figuring out if developers apply refactoring taking into account just the class properties, not considering their dependency from other classes, or if they accidentally or explicitly evaluate the impact of the performed refactoring on the neighboring classes, namely on the network topology.

In the first case, refactoring operations should look like random interventions on the nodes corresponding to classes, whereas in the second one we expect to find connections among the different refactored classes, that we are referring to as clustering. If refactoring shows the tendency to spread among linked classes, this information could be helpful for developers to make predictions, to keep track of which classes need to be refactored, or to detect other code smells.

For every system we retrieved the associated software network by parsing the source code looking for relationships between classes (like dependency, inheritance and collaboration). We then relied on RefFinder to recover all the refactorings related to these systems, and associated them to the corresponding nodes in the software network. RefFinder works by comparing two different versions of the same piece code, belonging to two different releases of the same system. It uses a template-reconstruction approach and, to the best of our knowledge, it represents the state-of-the-art tool for refactoring detection [62, 88, 3].

To gain information about clustering properties of refactored classes, we compared sets of refactored nodes to randomly chosen nodes. To understand if refactoring activities spread among connected classes, we analyzed the neighbors of refactored classes in the software networks, looking for other refactorings, and then performed again a comparison with classes randomly selected. Other recent works [74, 75] analyzed refactorings in the context of software networks, presenting a relationship between refactorings and node degree, but not analyzing clustering properties.

Our results show that refactored classes tend to be more connected than randomly selected classes, and the analysis on the first neighbors indicates that devising the topological structure of the software network can be of help in identifying which classes need to be refactored. The reported results are purely empirical and, at the present stage we have not yet found a specific cause or explanation for these findings. Nevertheless, we consider them quite interesting because they appear counterintuitive.

The innovative approach is to combine information gained by analyzing source code differences given by RefFinder and a topological analysis on software networks. We believe that the information retrieved by this kind of analysis can help developers to identify a subset of classes to be involved in refactoring activities that could be worth to consider during software development. In fact, according to the definition of refactoring, it is mandatory that the changes performed on source code do not alter software external behavior. Thus refactored classes should have on average the same connection density as other classes. On the contrary, we found that refactored classes are more tightly connected than average.

To perform our analysis, we build the software networks associated to every release of our software projects and try to identify refactored classes and network connections among them.

## 7.2 Experimental Settings

### 7.2.1 Datasets

Our investigation is structured in two consecutive experiments, performed on two datasets of open source software system written in Java. The two dataset significantly overlap being formed by the same system, with some differences. The first dataset is composed by 5 systems: Ant, Azureus, Jedit, Jena and Xalan. In total, we have analyzed 29 releases, for a dataset of tenths of thousands of classes. The second dataset is composed by 7 systems: Ant, Azureus, Jedit, Jena, Jtopen, Tomcat and Xalan, for a total of 66 releases studied. The analyzed systems belong the the Java Qualitas Corpus [60], [97] (release version 20101126e)

### 7.2.2 Analisys

The experimental setting is structure as it follows. It consists in:

1. Retrieving the software network.

2. Retrieving the refactoring by means of RefFinder.

3. Associating the refactoring to the corresponding classes.

4. Performing the computation (as described in the following).

We also performed several tests on some kind of toy classes that we built from scratch in order to figure out if RefFinder is able to properly detect refactoring operations and to associate them to the right classes. In particular, we checked if RefFinder retrieves refactoring in the correct way, avoiding side effects on connected classes. In fact, an error in associating the refactorings to the proper classes would introduce a bias on the analysis of the connectivity of the classes.

Consider, for example, the case of "rename method" refactoring. When this refactoring is performed on a class, the renamed method is called also in the connected classes. However, the refactoring was performed on the first class and not on the connected ones. We would like to check whether RefFinder associated this specific refactoring to the classes where it was performed and not to its connected classes which call the renamed method. These connected classes would undergo code changes which should not be retrieved as refactorings. RefFinder could have introduced a bias, since in the successive analysis we label the classes as refactored according to the output that it provides. In order to check this out, we built a simple network of connected classes, and performed a "rename method" refactoring on some of them. As a result, we found that this refactoring is properly associated to the classes where it was performed, namely the classes to which the method belongs to, as it should be, and not to their connected classes.

### 7.2.3 Refactored Classes vs Random Classes

We built the undirected network corresponding to each release by associating nodes to Java classes, and links to relationships among them, like inheritance, composition, dependencies, aggregation, association and so on. These relationships have been obtained by parsing the source code. We extracted the maximal connected component of the obtained software networks and performed our analysis on them.

We then used RefFinder [3] to extract the information about refactoring activities for each release. RefFinder analyzes the differences among source codes of two releases, the source and the target, and identifies the occurred refactoring operations. We analyzed only the refactorings associated by RefFinder to the source release, and we discarded the refactorings associated to the target release. Every refactoring was associated to the corresponding class, and so we were able to understand if classes affected by refactoring are connected or not.

After retrieving the refactorings on a class, we associate them to the corresponding node in the software network and looked at the links among refactored classes in the software network, with specific regards to the clustering phenomenon. With the term “clustering” we mean the tendency of refactored classes to form subnetworks composed by connected nodes. The most general definition of cluster we devised is the following: we consider a set of $n$ nodes as belonging to the same cluster if there is a path of length $d$ connecting each pair of nodes inside the set. In this work we limit our study to the case of $d = 1$, so we consider clusters as connected subnetworks. We analyze the clusters formed by classes involved in different refactorings and perform a comparison with clusters formed by randomly chosen classes.

Our hypothesis is that when a refactoring is applied, the involved classes have a higher probability of being connected with each other. To verify the hypothesis we selected in the software network the classes affected by a specific refactoring and denoted with $n$ their number. Inside this set of size $n$ we computed the number $c$ of independent clusters. The number of clusters $c$ varies from 1, when all classes are connected into one single cluster, to $n$ when all the selected classes are isolated. We compared $c$ with the corresponding number of clusters $c_{rand}$ obtained examining a number $n$ of classes selected at random in the entire software network. In this last case we performed 100 samplings and computed the average number of clusters. If classes involved in the same refactoring are on average more connected, the number of clusters they form must be lower than the corresponding number obtained for randomly selected classes, on average.

Since we consider the links between classes as undirected, the clusters do not depend on the direction of edges. We have found clusters formed by classes affected by refactoring, and compared the number of these clusters with the average number of clusters formed by a random selection of a number of classes equal to the number of refactored classes.

Afterwards, we tried to understand if the knowledge of network nodes corresponding to refactored classes may be used to infer which other classes may be in need of refactoring. To check this conjecture we selected a random subsample of all refactored classes to start from, about 10% of the total, and looked for refactored classes that were close to the starting set. To get a measure of closeness, we selected their first neighbors, namely the classes at distance $d = 1$ from the refactored ones. Then we computed how many classes among the set of first neighbors had also been refactored. We finally compared the number of refactored neighbors with the number of first neighbors of an equivalent set of classes selected at random. Our work thus aims at answering to the following research questions:

- RQ1: *Do refactored classes tend to be more interconnected than not refactored ones for a given type of refactoring?*

- RQ2: *Is it possible to identify refactoring-prone classes from the knowledge of other refactored classes?*

## 7.3 Results

Our work is aimed at understanding if refactoring activities are related to clustering and connectivity inside a software network. In the plot of Fig. 7.1 we show the number of clusters formed by refactored classes, and the average number of clusters obtained by selecting at random the same number of classes among all system classes. The plot shows that, given a fixed cluster size, the random selection brings a higher number of clusters compared to refactored classes. This means that refactoring mainly does not affect classes in a random fashion, but if a class needs this operation, there is a certain probability that also another class connected to the first will need to be refactored. This could be of help in software development and maintenance.

We also would like to point out the nearly linear growth of the number of clusters along with the number of classes selected, meaning that the random selection forms a number of clusters related and almost equal to the number of classes, as one would expect. These results confirm our previous analysis in [29], and they are valid specifically for refactored classes. In Fig. 7.2 we report a few examples of clusters for different refactorings in various systems.

In order to check whether refactored classes tend to be more interconnected than average and to form clusters we computed the ratio among the number of clusters formed by refactored classes and the number of clusters formed by randomly chosen classes. While computing this ratio, in order to reduce any fluctuation due to statistical noise, we considered only the clusters with size larger than (or equal to) 10 which we set arbitrarily - any other number could work. This choice is a trade-off between two extreme situations. In the first, considering fewer than 10 refactored classes, the chance to find a similar number of clusters from randomly chosen classes is high. In fact, in the hypothesis that random classes are generally disconnected, they tend to form $n$ clusters. Since refactored classes are not completely connected into a single cluster, with $n$ low the two numbers will be very similar.

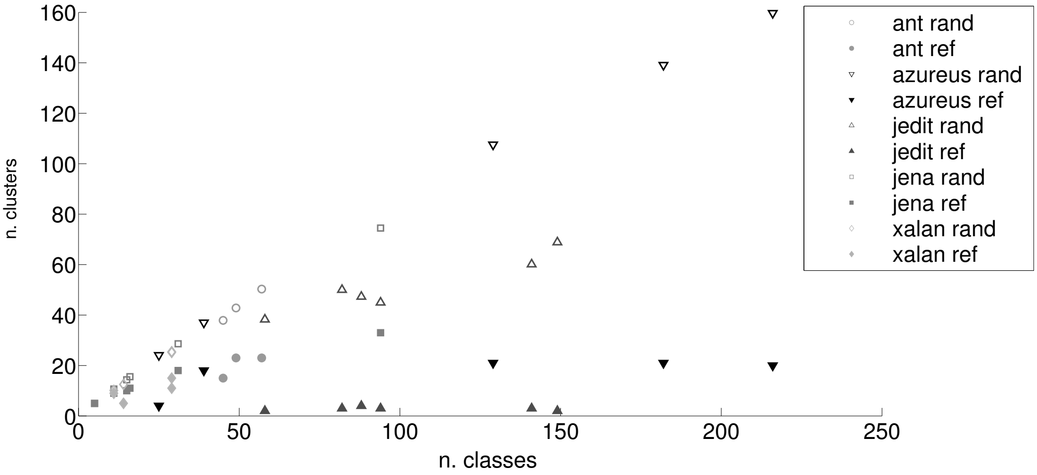


Figure 7.1: Comparison between the average number of clusters found by the random selection and the number of clusters formed by refactored classes. The average number of clusters for the random case (empty points of different shapes) is systematically bigger than in case of refactored classes, showing that the latter are more connected with each other.

In the second, considering only clusters with many more than 10 classes, this would optimize the ratio between the number of clusters formed by refactored classes and the number of clusters formed by randomly chosen classes, but the statistics will be drastically reduced, since the number of refactored classes per system is not very high (e.g. 45 for Ant 1.5 as reported in Tab. 7.4). Therefore the chosen value would provide a fair ratio with the mean square error (MSE) that increases according to $\sqrt{n}$ along with the number $n$ of the samples. Tab. 7.1 reports these ratios for one release for each analyzed project. Ratios are systematically lower than one for all refactorings suggesting that also one specific kind of refactoring involves classes which are more interconnected than average. Results in Tab. 7.1 provide a positive answer to RQ1: refactored classes tend to form interconnected clusters more than other classes on average.

In Fig. 7.3 we show an example of how refactoring activities could be related to a change in the topology of a software network. The example we show is a comparison between the releases 4.0.0.0 and 4.1.0.2 of the system Azureus. The figure shows the two sets of classes affected by the refactoring *replace method with method object*. After the refactoring operation, the connectivity among classes has changed, since the cluster grows by the addition of new classes. In particular, we can see that in the previous release, 4.0.0.0, the classes involved by this refactoring are 15, while in release 4.1.0.2 there

| | Ant 1.8.0 | Azureus 4.4.0.4 | JEdit 3.2 | Jena 2.1 | Jtopen 5.0 | Tomcat 5.5.3.1 | Xalan 2.5.0 |
|---|---|---|---|---|---|---|---|
| Add Parameter | - | - | - | - | 0.206 | - | - |
| Cons. Cond. Expression | - | - | - | - | 0.091 | - | - |
| Cons. Dup. Cond. Fragments | - | 0.215 | 0.126 | 0.142 | - | - | - |
| Extract Interface | 0.5 | 0.037 | - | - | - | - | - |
| Introduce Local Extension | - | - | - | - | - | 0.503 | - |
| Introduce Null Object | - | - | - | - | - | 0.405 | - |
| Inline Temp. | 0.503 | - | 0.021 | - | - | - | - |
| Move Fields | - | 0.056 | 0.168 | - | - | - | 0.119 |
| Remove Ass. to Param. | - | - | 0.084 | - | - | - | - |
| Remove Control Flag | - | - | - | - | - | - | 0.079 |
| Rep.Magic N. with Const. | - | - | 0.021 | 0.142 | 0.251 | - | - |

Table 7.1: Ratio between the number of clusters formed by different types of refactoring and the number of clusters formed by randomly selected classes

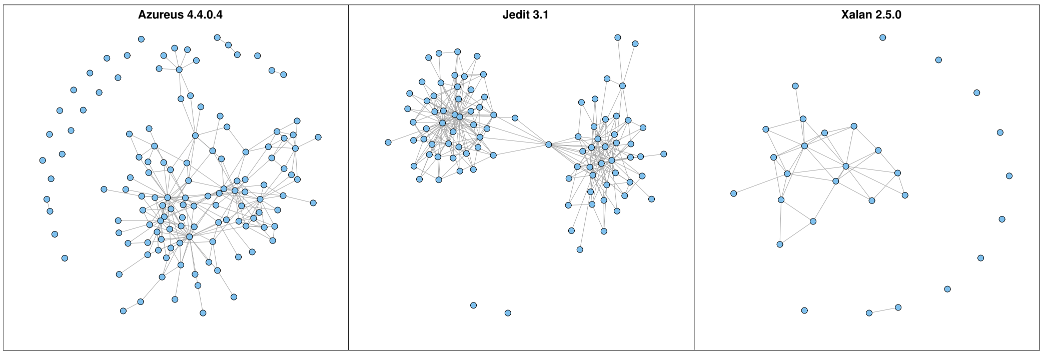


Figure 7.2: Three examples of networks composed only of classes involved in refactoring operations for Azureus 4.4.0.4 (left), JEdit (centre), Xalan 2.5.0 (right).

is a bigger cluster, composed of 20 classes, among which there are also the classes which were subject to the same refactoring, but nearly isolated in release 4.0.0.0, such as classes n. 1046 and n.7. We can consider for example the node n. 318 that corresponds to the class `PlatformConfigMessenger` in release 4.0.0.0. This class contains methods called `urlCanRPC(String url)` and `urlCanRPC(String url, boolean showDebug)`. In the next release, 4.1.0.2, these are found inside `UrlFilter` class, not present in the previous release, that it is exactly the method object. The clustering coefficient changes from 0.2 to 0.3 and this is coherent with the kind of refactoring applied. The analysis of cluster changes can thus be used to infer the kind of refactoring applied.

We now show the results of the analysis on the classes directly connected to refactored classes. We select at random a subset of refactored classes, $S_{ref}$, of size $n_{S_{ref}}$, which is about 10% of all refactored classes, and examine the set of classes directly connected to this subset. We denote the set of "first neighbor" classes with $SN_{ref}$. We repeat the procedure selecting at random subsets of classes regardless of refactoring, namely randomly chosen, of the same size $n_{S_{ref}}$, and examine the corresponding set of "first neighbor" classes, $SN_{rand}$. For both neighbor sets, $SN_{ref}$ and $SN_{rand}$, we compute the fraction of classes affected by refactoring operations, $F_{ref}$ and $F_{rand}$ respectively, and compare the two results.

These fractions represent the probability of finding a class in need to be refactored when starting from a set of refactored classes or starting from a set of random classes respectively. We averaged these fractions over 1000 cases where the set of refactored classes and that of random classes were repeatedly selected at random. Tables 7.2 and 7.3 report these results. Tab.

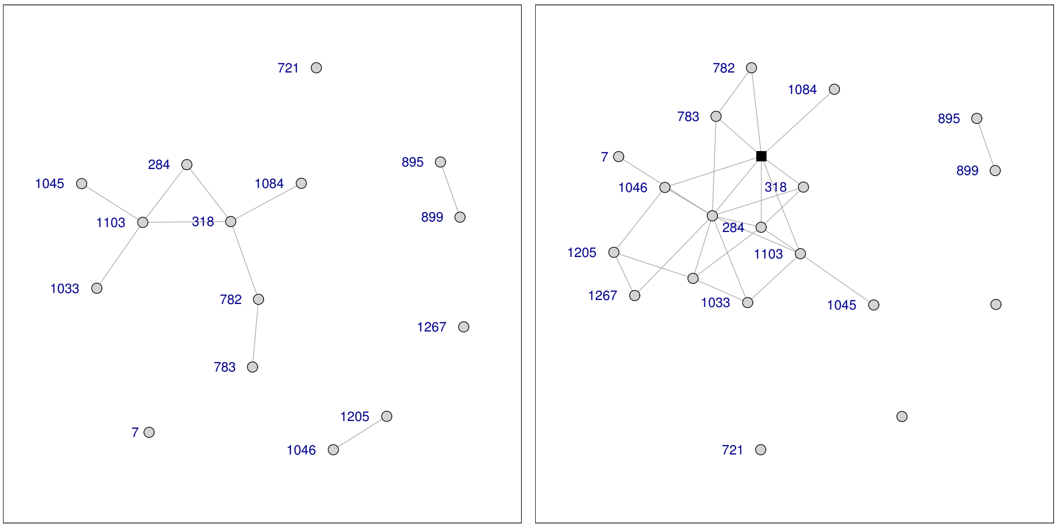


Figure 7.3: Comparison between Azureus release 4.0.0.0 (left) and 4.1.0.2 (right). For each release, we report the sets of classes affected by the refactoring named *replace method with method object*. The node corresponding to `UrlFilter` class has a squared shape and it is black on the right plot.

7.2 shows the fractions $F_{ref}$ and $F_{rand}$ mediated over the releases for each of the 7 projects, giving a general overview of the results, while in Tab. 7.3 we report some selected examples for both releases “Source” and “Target”, where the Source is the release before refactoring and the Target is the release after the application of refactorings.

The interpretation of this result is straightforward. $F_{ref}$ provides the empirical probability of finding a class in need to be refactored when picking up classes among the neighbors of a small set of refactored classes. $F_{rand}$ provides instead the same probability when choosing the classes at random inside the entire set. Thus, when looking for classes in need to be refactored, it is more convenient to examine first a set of classes directly linked to already refactored classes in the software network. This provides developers with an empirical practice to use when checking for classes to refactor. The ratio among the two fractions can be considered as an empirical index related to the "convenience" of looking in the first neighbors of a refactored set.

In Tab. 7.4 we present some representative cases taken from our dataset (one release for each system) that show the clusters formed by refactored classes (up to 7, due to space constraints), together with their size, the size of the set of first neighbors relative to each cluster, indicated by $n_i$, and the size of the set of first neighbor classes relative to the entire set of refactored classes, indicated by $n_{all}$. This analysis confirms the convenience of

| System | $F_{ref}$ Sources | $F_{ref}$ Targets | $F_{rand}$ Sources | $F_{rand}$ Targets |
|---|---|---|---|---|
| Ant | 0.218 | 0.196 | 0.062 | 0.057 |
| Azureus | 0.151 | 0.163 | 0.052 | 0.053 |
| Jedit | 0.458 | 0.448 | 0.281 | 0.271 |
| Jena | 0.094 | 0.085 | 0.024 | 0.024 |
| Jtopen | 0.107 | 0.11 | 0.038 | 0.04 |
| Tomcat | 0.25 | 0.249 | 0.166 | 0.163 |
| Xalan | 0.293 | 0.294 | 0.032 | 0.034 |

Table 7.2: Average values the fractions of refactored classes in the first neighbors network and the corresponding mean values computed for randomly selected classes, for each analyzed system.

examining refactored first neighbor classes when looking for possible classes in need to be refactored. In fact the more common situation is the presence of a large cluster of refactored classes (which we name cluster $C$ from now on) along with a set of smaller clusters, many of them containing just one class. Considering the largest cluster $C$, and observing the set of its first neighbor classes, its size $n_C$ is close to the size $n_{all}$ of all refactored classes and the number $n_{ref}$ of all refactored classes. Consider now a developer with a set of already refactored classes and in search for more classes to refactor, adopting the strategy of examining linked classes.

1. The probability that among the set of classes already refactored there will be at least one belonging to the larger cluster is very high.

2. Examining neighbor classes the entire cluster will be explored.

3. Examining all neighbor classes, there is an upper limit to the number of classes to check, given by $c_{all}$, which is a small fraction of system's size.

4. This upper limit will almost be reached starting from classes into cluster $C$, and thus the probability of reaching it is very high.

5. If all the classes selected at first belong to isolated refactorings, most of the classes in need to be refactored will not be reached, but at the same time the effort is minimal, since $\sum_i n_i$ for $i \neq C$ is small.

At the same time one can work jointly with other strategies devised for detecting classes to refactor, like code smells detection, [56, 92] in order to reduce the number of neighbor classes to examine. Since the fraction of

| **Source Release** | **Target Release** | $F_{ref}$ **Source** | $F_{ref}$ **Target** | $F_{rand}$ **Source** | $F_{rand}$ **Target** |
|---|---|---|---|---|---|
| Ant 1.5 | Ant 1.6.0 | 0.137 | 0.111 | 0.072 | 0.063 |
| Ant 1.6.0 | Ant 1.7.0 | 0.106 | 0.105 | 0.069 | 0.065 |
| Ant 1.7.0 | Ant 1.8.0 | 0.130 | 0.130 | 0.062 | 0.063 |
| Azureus 4.0.0.0 | Azureus 4.1.0.2 | 0.175 | 0.195 | 0.084 | 0.085 |
| Azureus 4.1.0.2 | Azureus 4.2.0.2 | 0.048 | 0.054 | 0.019 | 0.019 |
| Jedit 3.0 | Jedit 3.1 | 0.419 | 0.400 | 0.218 | 0.213 |
| Jedit 3.1 | Jedit 3.2 | 0.481 | 0.493 | 0.347 | 0.357 |
| Jena 2.0 | Jena 2.1 | 0.124 | 0.163 | 0.014 | 0.015 |
| Jena 2.1 | Jena 2.2 | 0.092 | 0.146 | 0.028 | 0,030 |
| Jtopen 3.3 | Jtopen 4.0 | 0.254 | 0.303 | 0.074 | 0.091 |
| Jtopen 4.0 | Jtopen 4.1 | 0.204 | 0.217 | 0.121 | 0.129 |
| Tomcat 4.1.4.0 | Tomcat 5.0.0 | 0.154 | 0.14 | 0.052 | 0.046 |
| Tomcat 5.0.0 | Tomcat 6.0.0 | 0.206 | 0.205 | 0.067 | 0.059 |
| Xalan 2.5.0 | Xalan 2.6.0 | 0.331 | 0.334 | 0.034 | 0.038 |
| Xalan 2.6.0 | Xalan 2.7.0 | 0.175 | 0.21 | 0.037 | 0.04 |

Table 7.3: Fractions of refactored classes in the first neighbors network and the corresponding values for randomly selected classes. The values refer both to “Source” and “Target” releases, of two releases of each system.

|  |  |  | Clus. 1 |  | Clus. 2 |  | Clus. 3 |  | Clus. 4 |  | Clus. 5 |  | Clus. 6 |  | Clus. 7 |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| System Name | $n_{ref}$ | $n_{all}$ | size | $n_1$ | size | $n_2$ | size | $n_3$ | size | $n_4$ | size | $n_5$ | size | $n_6$ | size | $n_7$ |
| Apache Ant 1.5 | 45 | 401 | 24 | 357 | 6 | 14 | 2 | 60 | 2 | 2 | 1 | 1 | 1 | 3 | 1 | 1 |
| Azureus 4.2.0.2 | 216 | 1603 | 191 | 1551 | 7 | 52 | 1 | 28 | 1 | 22 | 1 | 21 | 1 | 16 | 1 | 13 |
| JEdit 4.1 | 141 | 338 | 138 | 331 | 2 | 2 | 1 | 12 | - | - | - | - | - | - | - | - |
| Jena 2.3 | 5 | 268 | 1 | 249 | 1 | 11 | 1 | 5 | 1 | 2 | 1 | 1 | - | - | - | - |
| Jtopen 4.1 | 46 | 632 | 43 | 629 | 1 | 3 | 1 | 3 | 1 | 2 | - | - | - | - | - | - |
| Tomcat 6.0.0 | 157 | 530 | 134 | 447 | 2 | 28 | 2 | 19 | 4 | 12 | 1 | 8 | 1 | 7 | 1 | 6 |
| Xalan 2.4.0 | 14 | 146 | 8 | 126 | 2 | 17 | 2 | 8 | 1 | 14 | 1 | 1 | - | - | - | - |

Table 7.4: Values of the number of neighbors and the clusters size $n_i$ for the first 7 clusters of 7 software releases, in a decreasing order by cluster size. $n_{ref}$ is the total number of classes involved in refactoring operations affecting the corresponding release.

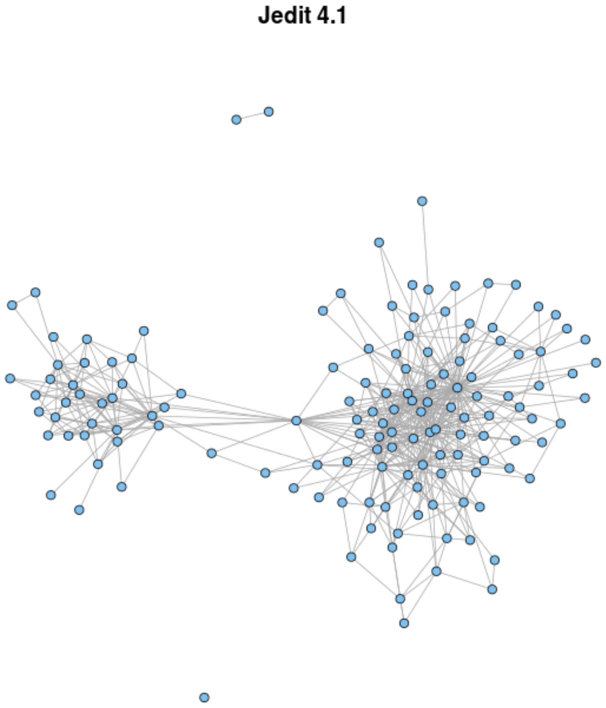


Figure 7.4: Jedit refactored classes network.

refactored classes is usually not too high, a fixed number of classes to refactor can be programmed in advance, and once this number is reached, the search among the first neighbors can stop and the classes in need to be refactored eventually missing will be very few.

Next we discuss, as an example, the case of Jedit 1.4. We are going to suggest how our empirical results can be used, together with other methodologies, to find refactor-prone classes. Fig. 7.4 reports the network of refactored classes for this case study. In this specific case the above mentioned strategy could be applied starting from one of these classes, and proceeding by inspecting the neighboring classes looking for refactoring opportunities. JEdit 1.4 is characterized by a total of 974 refactorings distributed over 46 classes, that represents the 7% of the entire system and they mainly belong to a cluster whose dimension are close to the totality of the refactored classes, as reported in Tab. 7.4. Refactored classes are reported in decreasing order of number of refactoring into Tab. 7.5. We consider one of the most refactored classes, namely `org.gjt.sp.jedit.textarea.JEditTextArea`.

In Fig. 7.5 the network of first neighbors for this class is represented. Tab. 7.5 reports, along with the number of refactorings per class, also the

| Class Name | N. Ref. | In-Ratio | Out-Ratio |
|---|---|---|---|
| bsh.ParserTokenManager | 83 | 1 | 0.25 |
| org.gjt.sp.jedit.syntax.ParserRule | 72 | 1 | 0.6 |
| org.gjt.sp.jedit.textarea.JEditTextArea | 40 | 0.5 | 0.71 |
| bsh.Parser | 35 | 1 | 0.45 |
| org.gjt.sp.jedit.Buffer | 29 | 0.4 | 0.64 |
| org.gjt.sp.jedit.jEdit | 27 | 0.39 | 0.69 |
| org.gjt.sp.jedit.browser.VFSBrowser | 26 | 0.3 | 0.72 |
| bsh.NameSpace | 25 | 0.65 | 0.6 |
| bsh.Interpreter | 21 | 0.84 | 0.75 |
| org.gjt.sp.jedit.syntax.DisplayTokenHandler | 19 | 1 | 0.6 |
| org.gjt.sp.jedit.browser.BrowserView | 15 | 0.22 | 0.75 |
| org.gjt.sp.jedit.gui.DockableWindowManager | 15 | 0.26 | 0.55 |
| bsh.Reflect | 14 | 0.75 | 0.62 |
| org.gjt.sp.jedit.search.SearchAndReplace | 14 | 0.45 | 0.79 |

Table 7.5: Number of refactorings per classes affected by more than 13 refactorings. In-Ratio and Out-Ratio columns report for, respectively, in-links and out-links, the fraction of neighboring classes that, among all the neighboring classes, are also involved in refactoring activities and the total number of neighbors.

ratio between the neighboring classes affected by refactorings and the total number of neighbors. As reported in Tab. 7.5 for this specific class, the fraction of neighboring classes that, among all the neighboring classes, are also involved in refactoring activities is 0.5 if they are connected by in-links (In-Ratio) and 0.71 if they are connected by out-links (Out-Ratio). It is worth to report that some neighbors are connected with both in-links and out-links.

So we consider now the scenario of a developer that is carrying out some refactoring operations and is working on `JEditTextArea`. We already know that, according to RefFinder, the neighboring classes were involved in refactoring operations. If he is looking for refactoring opportunities, even selecting at a random a class belonging to the set of neighboring classes, he has from 50% to 71% of chance to find a refactor-prone class, depending on which kind of connection he is looking at - in-links or out-links. The mean value of the fraction of refactored classes reported in Tab. 7.5 is 0.63 and 0.62 for neighboring classes connected respectively with in and out-links. Fig. 7.6 and 7.7 report a series of box plots representing in-links and out-links distributions and statistics for each analyzed release, showing that often the ratio is higher than 0.5. It is worth to underline that the suggested strategy is not expected to work alone and in any circumstance, but it could be useful when the soft-

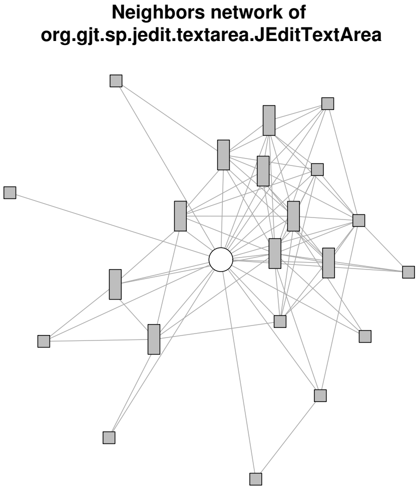


Figure 7.5: Neighboring classes for `org.gjt.sp.jedit.textarea.JEditTextArea`. The white circle represents the vertex corresponding to `JEditTextArea` class, whereas the rectangle-shaped and the squared-shaped vertices, represent the neighbors connected to `JEditTextArea` by respectively in-link and both in and out-link.

ware network present a specific topology. For this reason our proposal is not to use clustering information alone, but in cooperation with other topological analysis and integrated by other strategies as those described in [72].

### 7.3.1 Threats to Validity

The present study is affected by some threats to validity. All these threats are to be taken into account while replicating the study. In this section we present them according to the usual division in threats to the internal, external and construction validity.

**Internal validity.** We empirically found a significant relationship between classes involved in refactoring activity and network topology. However, the tendency of refactored classes to be more connected than others could be due to some undetected factors that we did not yet thoroughly investigate. For example we did not consider class complexity, and the fact that refactored classes could belong to the same package. Moreover, we studied refactoring activities without making distinctions among different refactorings. Our empirical results could

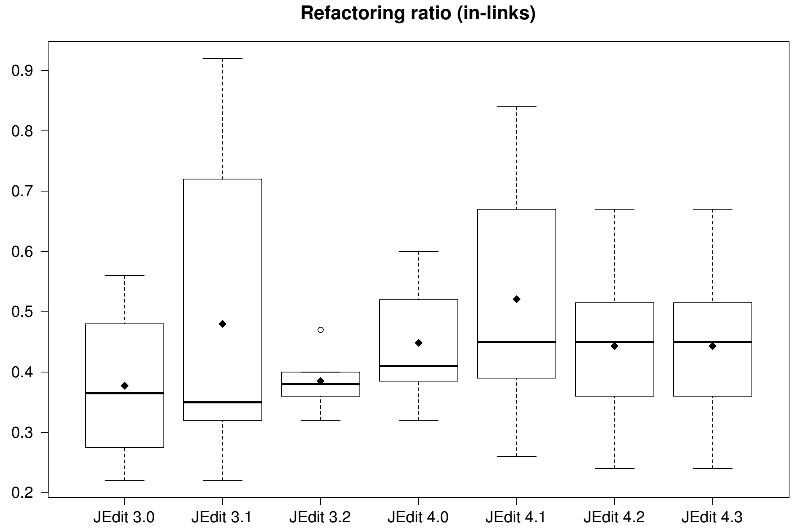


Figure 7.6: Average values of the fraction of the refactored neighboring classes linked by in-links for all the JEdit analyzed releseas. Diamond-shaped point represents the mean value.

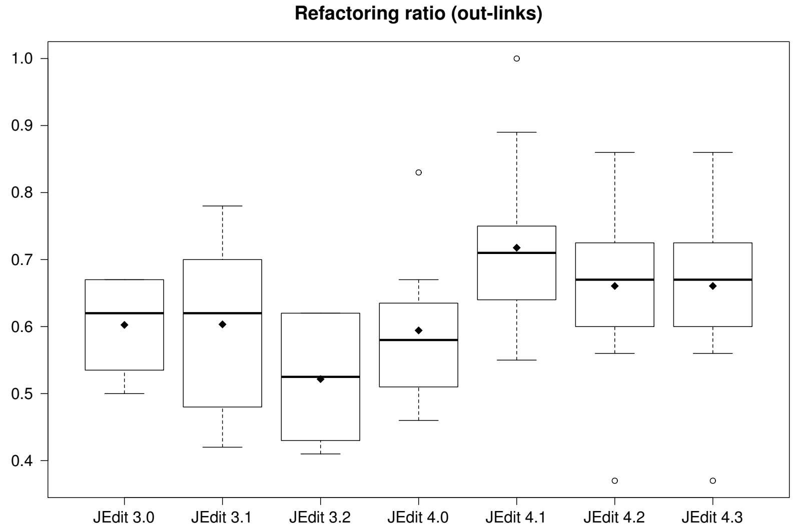


Figure 7.7: Average values of the fraction of the refactored neighboring classes linked by out-links for all the JEdit analyzed releseas. Diamond-shaped point represents the mean value.

be determined only by part of them. We considered all refactorings together in order to enhance our statistics and we did not have enough data for investigating each single refactoring separately.

**External validity.** We considered a certain number of software systems belonging to different categories, and performed different tasks. We made this choice in order to analyze a representative set of Java software system, that encompasses different kind of software, in order to avoid any influence of the specific domain on the results. Nevertheless, our sample is certainly limited. We analyzed only open source software, since it gave us the opportunity to freely parse the source code. We can not exclude the fact that different dynamics taking place in proprietary development environment could influence refactored classes topology. Additionally, all the analyzed software is written in Java (RefFinder parse only Java code) and all the software is written using the object oriented paradigm. Refactoring activities carried out on different languages, or in software designed according to different programming approaches, could lead to different results.

**Construction validity.** Refactorings retrieved using RefFinder depend on the reliability of this software. Despite being an acknowledged software for this kind of task, RefFinder cannot always retrieve all the refactorings. In addition, it cannot retrieve all the refactorings in the original Fowler's catalog and it is possible that results might change while studying these not covered refactorings.

## 7.4 Conclusions

In this Chapter we presented a study on the clustering of the classes interested by refactoring activities performed during software evolution. We analyzed several Open Source Object Oriented Java software systems using a complex network approach.

After retrieving the source code and building the software network for each release of the analyzed systems, we extracted the refactoring operations using RefFinder. We then analyzed every release of the software systems to understand if classes are more connected with each other after undergoing a refactoring operation. We compared the number of clusters formed by refactored classes to the average number of clusters formed by a random selection of classes, where the size of the sample was set equal to the number of refactored classes. Our results seem to support the initial hypothesis that refactored classes tend to form clusters. This suggests that randomly selected classes are poorly coupled with respect to refactored classes, thus confirming our hypothesis. In order to deepen our understanding on the relationship between refactoring activities and node connectivity, we studied the subnetworks composed by the first neighbors of refactored clusters. We showed that not only the refactored classes form clusters according to the provided definition, but also a significant fraction of their first neighbors are interested by refactoring activities, providing a comparison with a null model represented by randomly sampled classes. These results suggest some practical applications in the field of software engineering, since it could allow developers to find out other classes to refactor while carrying out refactoring activities.

Indentifying the presence of clusters among refactored classes can help developers to distinguish and decide which other classes are good candidates for being refactored, once a class or a file has been chosen to be refactored. In fact, since refactored classes or files tend to form clusters, one should look at the nearest neighbours of the refactored ones. Moreover, since we have shown an example where refactoring activities change the cluster structure of a software network, one can in principle understand something about these activities by simply analyzing the cluster structure of subsequent releases of a software system. For example, it could be possible to make a prediction about which classes need to be refactored, or understand if some classes were refactored, by looking at the cluster structure in the proximity of the involved classes.

These preliminar results could be extended in order to understand if it is possible to make such predictions. Since refactoring generally affects software quality by improving coupling an cohesion, the clustering properties of refactoring activities could be also related to this feature, and then to software quality. For example, we could analyze the topology of such clusters along software evolution to understand if they are related to an improvement of software quality.

These preliminary analysis involved only Java systems from the Qualitas Corpus, but it could be extended to other systems in order to have a bigger statistics and to better distinguish also among different types of refactoring. Indeed, this analysis can be extended also to the other types of refactorings from Fowler's catalogue, to understand also if some types of refactorings tend to form clusters more than others. Our analysis involves different releases over time, and so it can be also viewed as a study of software evolution for what concerns refactoring activities.

# Chapter 8

# Concluding Remarks

This thesis reports the most meaningful outcomes of a research work conducted during the three years of the author's Ph.D., in collaboration with Professors and colleagues of the Agile Group at the Department of Electrical and Electronic Engineering (DIEE) of the University of Cagliari.

This research is devoted to shedding some light on how software systems are structured and evolve, in order to acquire useful insights that can help software engineers to develop programs of better quality with an advantageous trade-off between costs and benefits.

The author's research interests involve both product metrics - particularly new metrics derived from the concept of complex networks, and software engineering practices. I focused specific attention on the assessment of software quality and the impact of development practices on software systems.

With regards to the product, my efforts were addressed to figure out if novel metrics might be of any help to detect or forecast software defectiveness during software development. I found a meaningful correlation between some network metrics and software defectiveness.

As for the development practices, I investigated some aspects of software development, concerning to the use of inheritance in the development of Python systems and the use of refactoring in Java software systems. Even if I did not adopt explicitly process metrics, I derived some interesting information from the topological property of software networks.

Throughout this dissertation, I mainly adopted a novel approach to assess software quality and, in general, to measure software properties, based on the concept of complex network. I illustrated how and to which extent software systems can be represented as complex networks, presenting the advantages of this approach.

In Chapters 4 and 5, I analyzed some software systems performing both a case study and a longitudinal study. The results reported in Chapter 4, show the relationship between some network metrics (number of communities, modularity, and clustering) and software defectiveness. If used in the context of predictive analysis, this relationship supports the use of network metrics to forecast the extremal values of defectiveness in future releases.

In Chapter 5 I illustrated the results of an analysis performed on some releases of two software systems, showing that classes affected by Issues have a clear tendency to be connected with each other. This preliminary result may lead to interesting practical application, being possible to exploit this property to enable developers to find most bug-prone classes.

The sixth Chapter reports a study on the use of inheritance in Python development. I described the Python *corpus*, namely the curated collection of Python programs that I collected in order to perform this and future empirical studies on Python systems. Then I presented the results of our analysis, using classic and new metrics. Being a replication study one of the outcomes is represented by the comparison of the same analysis previously conducted on Java programs. I found some meaningful differences and provided an interpretation of the results. This study represents the first step of a wider research aimed at understanding if and to which extent developers use properly the best programming practices.

Chapter 7 present a study on the use of refactoring on Java software systems. I analyzed the impact of refactoring activities on software topology, showing that classes involved in refactoring activities tend to be more connected than other classes and form clusters. Also, this result might have significant implication in future research and practical applications, since it could be used, in co-operation with other techniques, in the design of recommender systems for refactoring.

# Chapter 9

# List of Publications Related to this Thesis

1. A study of the community structure of a complex software network, Concas, G.; Monni, C.; Orrú, M.; Tonelli, R. in Emerging Trends in Software Metrics (WETSoM), 2013 4th International Workshop on, vol., no., pp.14-20, 21-21 May 2013
doi: 10.1109/WETSoM.2013.6619331

2. Two Case Studies on Clusterization of Refactored Classes G. Concas, M. Marchesi, C. Monni, M. Orrú, R. Tonelli - International Workshop on Refactoring & Testing (RefTest) 2013

3. Are Refactoring Practices Related to Clusters in Java Software? Giulio Concas, Cristina Monni, Matteo Orrú, and Roberto Tonelli - 15th International Conference on Agile Software Development - May 26th - 30th - Rome Agile Processes in Software Engineering and Extreme Programming Volume 179 of the series Lecture Notes in Business Information Processing pp 269-276

4. Refactoring Clustering in Java Software Networks G. Concas, C. Monni, M. Orrú, M. Ortu, and R. Tonelli - International Workshop on Refactoring & Testing (RefTest) - May 2014, Rome, Italy

5. Giulio Concas, Cristina Monni, Matteo Orrú, and Roberto Tonelli. 2014. Clustering of defects in Java software systems. In Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics (WETSoM 2014). ACM, New York, NY, USA, 59-65.
DOI=http://dx.doi.org/10.1145/2593868.2593879

6. Could Micro Patterns Be Used as Software Stability Indicator? Marco Ortu, Giuseppe Destefanis, Matteo Orrú, Roberto Tonelli, Michele L. Marchesi - Patterns Promotion and Anti-patterns Prevention (PPAP)

7. The Evolution of Knowledge in the Refactoring Research Field Matteo Orrú, Simone Porru, Michele Marchesi, Roberto Tonelli - 16th International Conference on Agile Software Development (XP2015)

8. Predicting Software Quality through Network Analysis Matteo Orrú, Cristina Monni, Michele Marchesi, Giulio Concas, Roberto Tonelli - 8th Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE 2015)

9. Matteo Orrú, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and Giuseppe Destefanis. 2015. A Curated Benchmark Collection of Python Systems for Empirical Studies on Software Engineering. In Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '15). ACM, New York, NY, USA, , Article 2 , 4 pages.
   DOI=http://dx.doi.org/10.1145/2810146.2810148

10. A Complex Network Approach for Museum Services - A Model for Digital Content Management Filippo Eros Pani , Simone Porru , Matteo Orrú, Simona Ibba - 7th International Conference on Knowledge Management and Information Sharing

11. Hashtag of Instagram: From Folksonomy to Complex Network Simona Ibba , Matteo Orrú , Filippo Eros Pani, Simone Porru - 7th International Conference on Knowledge Engineering and Ontology Development

12. A Preliminary Study on Mobile Apps Call Graphs through a Complex Network Approach Matteo Orrú, Simone Porru, Roberto Tonelli, Michele Marchesi - COMPLEX NETWORKS 2015 - The 4th International Workshop on Complex Networks and their Applications. November 23-27, 2015 Bangkok, Thailand Collocated with: The 11th International Conference on Signal Image Technology & Internet Based Systems SITIS 2

13. How Do Python Programs Use Inheritance? A Replication Study Matteo Orrú, Ewan Tempero, Michele Marchesi and Roberto Tonelli - ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC2015)

# Bibliography

[1] In *Cvs. http://www.nongnu.org/cvs/.* [cited at p. 9]

[2] Python Package Index: https://pypi.python.org/pypi. [cited at p. 56, 65]

[3] Reffinder. https://webspace.utexas.edu/kp9746/www/reffinder/. [cited at p. 13, 68, 70]

[4] The Promise Repository of Empirical Software Engineering data: http://openscience.us/repo, 2015. [cited at p. 49, 50, 52]

[5] Deepak Advani, Youssef Hassoun, and Steve Counsell. Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1713–1720, New York, NY, USA, 2006. ACM. [cited at p. 13]

[6] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, January 2002. [cited at p. 5, 10]

[7] C. Andersson and P. Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Trans. Softw. Eng.*, 33(5):273–286, May 2007. [cited at p. 38]

[8] Mahir Arzoky, Stephen Swift, Allan Tucker, and James Cain. Munch: An efficient modularisation strategy to assess the degree of refactoring on sequential source code checkings. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 422–429, Washington, DC, USA, 2011. IEEE Computer Society. [cited at p. 13]

[9] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *Proceedings of the 2007 conference of the center for advanced*

*studies on Collaborative research*, CASCON ’07, pages 215–228, New York, NY, USA, 2007. ACM. [cited at p. 10]

[10] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity Volume 1.* MIT Press, Cambridge, MA, USA, 1999. [cited at p. 16]

[11] A. Barabasi, R. Albert, and H. Jeong. Scale-free characteristics of random networks: the topology of the world wide web. *Phys. A*, 281:69–77, 2000. [cited at p. 2, 5, 7, 15, 37]

[12] A.-L Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999. [cited at p. 38]

[13] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, October 1996. [cited at p. 47, 49]

[14] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006. [cited at p. 50]

[15] Bart Du Bois and Tom Mens. Describing the impact of refactoring on internal program quality. pages 37–48, Vrije Universiteit Brussel, 2003. [cited at p. 8]

[16] Lionel C. Briand, John W. Daly, Victor Porter, and JÃ¼rgen WÃ¼st. A comprehensive empirical validation of product measures for object-oriented systems, 1998. [cited at p. 47, 49]

[17] Michelle Cartwright and Martin Shepperd. An empirical view of inheritance, 1998. [cited at p. 47]

[18] HernÃ¡n A. Makse Chaoming Song, Shlomo Havlin. Self-similarity of complex networks. *Nature*, 433(4):392–395, January 2005. [cited at p. 7, 15, 37]

[19] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, June 1994. [cited at p. 2, 8, 16, 68]

[20] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994. [cited at p. 49, 63]

[21] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Softw. Eng.*, 24(8):629–639, August 1998. [cited at p. 47]

[22] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70(6):066111, December 2004. [cited at p. 6, 11, 19, 20]

[23] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, 51(4):661–703, November 2009. [cited at p. 59, 63]

[24] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Softw. Pract. Exper.*, 37(6):581–641, May 2007. [cited at p. 47, 49]

[25] G. Concas, M. Marchesi, A. Murgia, R. Tonelli, and I. Turnu. On the distribution of bugs in the eclipse system. *IEEE Transactions on Software Engineering*, 37(6):872–877, November 2011. [cited at p. 7, 38]

[26] Giulio Concas, Michele Marchesi, Giuseppe Destefanis, and Roberto Tonelli. An empirical study of software metrics for assessing the phases of an agile project. *Int. J. Soft. Eng. Knowl. Eng*, 22:525, 2012. [cited at p. 16]

[27] Giulio Concas, Michele Marchesi, Alessandro Murgia, and Roberto Tonelli. An empirical study of social networks metrics in object-oriented software. *Adv. Soft. Eng.*, 2010:4:1–4:21, January 2010. [cited at p. 18]

[28] Giulio Concas, Cristina Monni, Matteo Orrù, and Roberto Tonelli. A study of the community structure of a complex software network. In *4th International Workshop on Emerging Trends in Software Metrics, WETSoM 2013, San Francisco, CA, USA, May 21, 2013*, pages 14–20, 2013. [cited at p. 57]

[29] Giulio Concas, Cristina Monni, Matteo Orrù, and Roberto Tonelli. Two case studies on clusterization of refactored classes. In *International Workshop on Refactoring and Testing (RefTest)*, XP2013, 2013. [cited at p. 23, 72]

[30] A. Lombardoni D. Cahllet. Bug propagation and debugging in asymmetric software structures. *Phys. Rev E*, 70, 2004. [cited at p. 7]

[31] John W. Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996. [cited at p. 47, 49]

[32] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010. [cited at p. 7]

[33] Tom DeMarco. *Controlling software projects : management, measurement and estimation*. Yourdon Press, New York, NY, 1982. [cited at p. 2]

[34] Giuseppe Destefanis. Technical report: Which programming language should a company use? a twitter-based analysis. *CRIM - Technical Report*, 2014. [cited at p. 47, 48]

[35] Giuseppe Destefanis, Steve Counsell, Giulio Concas, and Roberto Tonelli. Software metrics in agile software: An empirical study. In *Agile Processes in Software Engineering and Extreme Programming*, pages 157–170. Springer, 2014. [cited at p. 49]

[36] Giuseppe Destefanis, Roberto Tonelli, Ewan Tempero, Giulio Concas, and Michele Marchesi. Micro pattern fault-proneness. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 302–306. IEEE, 2012. [cited at p. 57]

[37] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005. [cited at p. 50]

[38] M. Eaddy, T. Zimmermann, K.D. Sherwood, V. Garg, G.C. Murphy, and et al. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, November 2008. [cited at p. 10]

[39] Gabriel Farah, Juan Sebastian Tejada, and Dario Correal. Openhub: A scalable architecture for the analysis of software quality attributes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 420–423, New York, NY, USA, 2014. ACM. [cited at p. 50]

[40] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, August 2000. [cited at p. 38]

[41] Sergio Focardi, Michele Marchesi, and Giancarlo Succi. *A stochastic model of software maintenance and its implications on extreme programming processes*, pages 191–206. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. [cited at p. 7, 15]

[42] S. Fortunato. Community detection in graphs. *Physics Report*, 486:75–174, February 2010. [cited at p. 5, 6, 7]

[43] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. [cited at p. 8, 12, 13, 67]

[44] T. Nepusz G. Csardi. The igraph software package for complex network research. *InterJournal of Complex Systems*, page 1695, 2006. [cited at p. 12, 19]

[45] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. [cited at p. 47]

[46] Birgit Geppert, Audris Mockus, and Frank RÃ¶Ãler. Refactoring for changeability: A way to go? In *IEEE METRICS*, page 13. IEEE Computer Society, 2005. [cited at p. 8]

[47] Joseph Yossi Gil and Itay Maman. Micro patterns in Java code. In *ACM SIGPLAN Notices*, volume 40, pages 97–116. ACM, 2005. [cited at p. 57]

[48] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Science*, 99:7821–7826, June 2002. [cited at p. 16, 38]

[49] B. H. Good, Y. A. De Montjoye, and A. Clauset. Performance of modularity maximization in practical contexts. *Physical Review E*, 81(4):046106, 2010. [cited at p. 33]

[50] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. [cited at p. 50]

[51] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, July 2000. [cited at p. 38]

[52] Philip Guo. Python is now the most popular introductory teaching language at top us universities. *BLOG@ CACM, July*, 2014. [cited at p. 47]

[53] R. Siket I. Gyimothy, T. Ferenc. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31, 2005. [cited at p. -]

[54] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, Oct. 2005. [cited at p. 8, 31]

[55] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012. [cited at p. 7]

[56] H. Hamza, S. Counsell, T. Hall, and G. Loizou. Code smell eradication and associated refactoring. In *Proceedings of the 2Nd Conference on European Computing Conference*, ECC'08, pages 102–107, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS). [cited at p. 77]

[57] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *J. Syst. Softw.*, 52(2-3):173–179, June 2000. [cited at p. 47, 49]

[58] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 38]

[59] Susan Hunston. *Corpora in applied linguistics.* Cambridge University Press, 2006. [cited at p. 66]

[60] Java Qualitas Corpus. http://qualitascorpus.com/. [cited at p. 69]

[61] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. *Software Maintenance, 2002. Proceedings. International Conference on*, pages 576–585, 2002. [cited at p. 8]

[62] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 371–372, New York, NY, USA, 2010. ACM. [cited at p. 68]

[63] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Trans. Softw. Eng.*, 21(12):929–944, December 1995. [cited at p. 47]

[64] G. A. Kohring. Complex dependencies in large software systems. *Advances in Complex Systems (ACS)*, 12(06):565–581, 2009. [cited at p. 38, 67]

[65] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, (4):046110. [cited at p. 32]

[66] Deyi Li, Yanni Han, and Jun Hu. Complex network thinking in software engineering. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 01*, CSSE '08, pages 264–268, Washington, DC, USA, 2008. IEEE Computer Society. [cited at p. 16, 37, 67]

[67] Johnny Wei-Bing Lin. Why Python is the next wave in earth sciences computing. *Bulletin of the American Meteorological Society*, 93(12):1823–1824, 2012. [cited at p. 47]

[68] Yang-Yu Liu, Jean-Jacques Slotine, and Albert-László Barabási. Controllability of complex networks. *Nature*, 473(7346):167–173, 2011. [cited at p. 38]

[69] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Approach.* Prentice Hall, 1994. [cited at p. 68]

[70] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The PROMISE Repository of empirical software engineering data, June 2012. [cited at p. 50]

[71] S. Milgram. The small world problem. *Psych. Today*, 2:60–67, 1967. [cited at p. 5]

[72] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36, January 2010. [cited at p. 82]

[73] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does refactoring improve reusability? In Maurizio Morisio, editor, *ICSR*, volume 4039 of *Lecture Notes in Computer Science*, pages 287–297. Springer, 2006. [cited at p. 8]

[74] A. Murgia, M. Marchesi, G. Concas, R. Tonelli, and S. Counsell. Parameter-based refactoring and the relationship with fan-in/fan-out coupling. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 430–436, Washington, DC, USA, 2011. IEEE Computer Society. [cited at p. 8, 68]

[75] A. Murgia, R. Tonelli, M. Marchesi, G. Concas, S. Counsell, J. McFall, and S. Swift. Refactoring and its relationship with fan-in and fan-out: An empirical study. In *Proceedings of Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, CSMR '12, pages 63–72, 2012. [cited at p. 7, 8, 68]

[76] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012. [cited at p. 8, 13]

[77] Christopher R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E*, 68(4):046116, Oct 2003. [cited at p. 2, 7, 15, 16, 37, 38, 67]

[78] Sebastian Nanz and Carlo A. Furia. A Comparative Study of Programming Languages in Rosetta Code. *CoRR*, abs/1409.0252, 2014. [cited at p. 49]

[79] M. E. Newman. Fast algorithm for detecting community structure in networks. *Phys. Rev E*, 69(6):066133, June 2004. [cited at p. 6, 11]

[80] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, February 2004. [cited at p. 6, 7, 11, 15, 16, 32]

[81] M. E. J. Newman. The structure and function of complex networks. *SIAM REVIEW*, 45:167–256, 2003. [cited at p. 5, 10, 16]

[82] Thanh H. D. Nguyen, Bram Adams, and Ahmed E. Hassan. A case study of bias in bug-fix datasets. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 259–268, Washington, DC, USA, 2010. IEEE Computer Society. [cited at p. 7]

[83] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, December 1996. [cited at p. 38]

[84] William F. Opdyke. Refactoring object-oriented frameworks. Technical report, 1992. [cited at p. 7]

[85] Matteo Orrú, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and Giuseppe Destefanis. A curated benchmark collection of Python systems for empirical studies on software engineering. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '15, pages 2:1–2:4, New York, NY, USA, 2015. ACM. [cited at p. 65]

[86] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355, 2005. [cited at p. 7]

[87] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972. [cited at p. 16]

[88] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. [cited at p. 13, 68]

[89] Valverde S. Cancho R. and V. Sole. Scale free networks from optimal design. *Europhysics Letters*, 60, 2002. [cited at p. 6, 7, 38, 67]

[90] Markus Lumpe Rajesh Vasa and Allan Jones. Helix - Software Evolution Data Set. [cited at p. 50]

[91] R. Sanchez and J. T. Mahoney. Modularity, flexibility, and knowledge management in product and organization design. *Strategic Management Journal*, 17:pp. 63–76, 1996. [cited at p. 16]

[92] Frank Simon, Frank Steinbr"uckner, and Claus Lewerentz. Metrics based refactoring. In *In Proc. 5th European Conference on Software Maintenance and Reengineering*, pages 30–38, 2001. [cited at p. 8, 77]

[93] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, WoSQ '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 8]

[94] LOVRO SUBELJ, SLAVKO ZITNIK, NELI BLAGUS, and MARKO BAJEC. Node mixing and group structure of complex software networks. *Advances in Complex Systems*, 0(0):1450022, 0. [cited at p. 16, 23]

[95] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. An empirical exploration of the distributions of the Chidamber and Kemerer Object-Oriented Metrics Suite. *Empirical Softw. Engg.*, 10(1):81–104, January 2005. [cited at p. 47, 49, 50]

[96] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010. [cited at p. 48, 49, 50, 51, 56]

[97] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010. [cited at p. 69]

[98] Ewan Tempero, James Noble, and Hayden Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag. [cited at p. 47, 49, 57, 65]

[99] Ewan Tempero, Hong Yul Yang, and James Noble. What programmers do with inheritance in Java. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 577–601, Berlin, Heidelberg, 2013. Springer-Verlag. [cited at p. 49]

[100] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of

the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013. [cited at p. 50]

[101] Ivana Turnu, Giulio Concas, Michele Marchesi, Sandro Pinna, and Roberto Tonelli. A modified yule process to model the evolution of some object-oriented system properties. *Information Sciences*, 181:883–902, 2011. [cited at p. 7]

[102] Ivana Turnu, Michele Marchesi, and Roberto Tonelli. Entropy of the degree distribution and object-oriented software quality. In *Proceedings of the 2012 ICSE Workshop on Emerging Trends in Software Metrics*, WETSoM '12, pages 77–82, 2012. [cited at p. 7, 15, 37]

[103] Understand. Scitools.com: https://scitools.com. [cited at p. 52, 56, 57]

[104] L. Šubelj and M. Bajec. Community structure of complex software systems: Analysis and applications. *Physica A Statistical Mechanics and its Applications*, 390:2968–2975, August 2011. [cited at p. 6, 7, 15, 16, 37, 38]

[105] Sergi Valverde and Ricard V. Sole. Hierarchical small worlds in software architecture. arXiv:cond-mat/0307278v2, 2003. [cited at p. 6, 7]

[106] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1 edition, 2003. [cited at p. 8]

[107] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998. [cited at p. 38]

[108] Lian Wen, Diana Kirk, and R. G. Dromey. Software systems as complex networks. In *Proceedings of the 6th IEEE International Conference on Cognitive Informatics*, COGINF '07, pages 106–115, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 7, 67]

[109] Lian Wen, Diana Kirk, and R. Geoff Dromey. Software systems as complex networks. In Du Zhang, Yingxu Wang, and Witold Kinsner, editors, *IEEE ICCI*, pages 106–115. IEEE, 2007. [cited at p. 16]

[110] Dirk Wilking, Umar Farooq Kahn, and Stefan Kowalewski. An empirical evaluation of refactoring. *e-Informatica*, 1(1):27–42, 2007. [cited at p. 8]

[111] Ian H. Witten, Sally Jo Cunningham, and Mark D. Apperley. The New Zealand digital library project. *New Zealand Libraries*, 48:146–152, 1996. [cited at p. 50]

[112] Hongyu Zhang. On the distribution of software faults. *IEEE Transactions on Software Engineering*, 34(2):301–302, 2008. [cited at p. 38]

[113] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM. [cited at p. 7, 16]

[114] Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors. *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013.* IEEE Computer Society, 2013. [cited at p. 50]