PALOMA MAIRA DE OLIVEIRA

EXTRACTING RELATIVE THRESHOLDS FOR SOURCE CODE METRICS

Thesis presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Ciência da Computação.

Advisor: Marco Tulio Valente

Belo Horizonte

December 2015

Acknowledgments

This work would not have been possible without the support of many people.

I thank God to provide me the discipline and persistence to reach a Ph.D. degree.

I thank my dear husband Arthur, who has had patience in difficult moments and has always been at my side.

I thank my whole family—especially my father Ari, my mother Clélia, and my sister Fernanda—for having always supported me.

I thank my advisor M. T. Valente for his attention, motivation, patience, dedication, and immense knowledge. I certainly could not complete my Ph.D study without him.

I thank A. Serebrenik for giving me the opportunity to work under his supervision in The Netherlands.

I thank A. Bergel for the valuable collaboration in the case studies.

I thank my colleagues of the IFMG—especially A. F. Camargos, B. Ferreira, D. F. Resende, E. Valadão, F. P. Lima, G. Ribeiro, M. P. Junior, and W. A. Rodrigues—for the friendship and cooperation.

I thank the members of the ASERG research group—especially A. Hora, C. Couto, H. Borges, and L. L. Silva—for the friendship and technical collaboration.

I would like to express my gratitude to the members of my thesis defense—D. Serey (UFCG), E. Figueiredo (UFMG), E. S. Almeida (UFBA), K. A. M. Ferreira (CEFET-MG) and M. A. S. Bigonha (UFMG).

Abstract

Meaningful thresholds are needed for promoting software metrics as an effective instrument to measure the internal quality of systems. To address this challenge, in this PhD Thesis, we propose the concept of Relative Thresholds (RT) for evaluating metrics data following heavy-tailed distributions. The proposed thresholds assume that metric thresholds should be followed by *most* entities, but that it is also natural to have a number of entities in the "long-tail" that do not follow the defined limits. We describe an empirical method for deriving RT from a corpus of systems. We also perform an extensive analysis of RT: (i) we apply RT on a sample of 308 GitHub repositories. We found that most repositories follow the extracted RT; (ii) We compare the proposed RT with thresholds extracted according to a method used by the software industry. We concluded that both methods convey similar information. However, our method derives RT that can be automatically used to detect noncompliant systems; (iii) we evaluate the influence of context in our results and we concluded that the impact on RT of context changes is limited; (iv) we perform a historical analysis to check whether the proposed RT are followed by different versions of the systems under analysis. We found that our RT capture enduring software properties; (v) we check the importance of classes that do not follow the upper limit of a RT and we found these classes are important in terms of maintenance activities; (vi) we investigate the relation between the presence of bad smells in a system and its adherence to the proposed RT. We do not found evidence that noncompliant systems have more density of bad smells; (vii) we evaluated the dispersion of the metric values in the systems respecting the proposed RT, using the Gini coefficient. We found that there are different distributions of methods per class among the systems that follow the proposed RT; and (viii) We conducted a qualitative study to evaluate our method with developers. The results indicate that well-designed systems respect the RT. In contrast, we observed that developers usually have difficulties to indicate poorly-designed systems.

Keywords: Source code metrics, Thresholds, Heavy-tailed Distributions.

Contents

Acknowledgments **xi**

Resumo **xiii**

Abstract **xv**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**

1.1 Motivation 1
1.2 Problem Statement 2
1.3 Goals and Contributions 4
1.4 Thesis Outline 6
1.5 Publications 7

2 Background **9**

2.1 Software Quality 9
2.1.1 Software Process Quality 9
2.1.2 Software Product Quality 10
2.2 Source Code Metrics 12
2.2.1 The CK Metrics Suite 13
2.3 Statistical Properties of Source Code Metrics 15
2.3.1 Metrics Values Distributions 17
2.3.2 Discussion 19
2.4 Thresholds Definitions 19
2.4.1 Extracting Thresholds using Traditional Techniques 20
2.4.2 Extracting Thresholds from Repositories 21

2.4.3 Extracting Thresholds using Error Models . . . 23
2.4.4 Extracting Thresholds using Clustering Algorithms . . . 24
2.4.5 Discussion . . . 25
2.5 Studies with Developers . . . 26
2.6 Final Remarks . . . 27

3 Proposed Method **29**
3.1 Relative Thresholds . . . 29
3.2 Illustrative Example . . . 31
3.3 Method Properties and Characteristics . . . 33
3.3.1 Adherence to Requirement of Metric Aggregation Techniques . . . 33
3.3.2 Staircase Effects . . . 34
3.3.3 Tolerance to Bad Smells . . . 35
3.3.4 Statistical Properties . . . 35
3.4 *RTTool* . . . 36
3.4.1 Example of usage . . . 37
3.4.2 Performance . . . 39
3.4.3 Availability . . . 40
3.4.4 Related Tools . . . 41
3.5 Final Remarks . . . 41

4 Relative Thresholds for the Qualitas Corpus **43**
4.1 Corpus and Metrics . . . 43
4.2 Study Setup . . . 45
4.3 Results . . . 45
4.4 Application on Popular GitHub Repositories . . . 46
4.4.1 Study Setup . . . 48
4.4.2 Results . . . 49
4.5 Comparison with SIG Method . . . 51
4.5.1 Results . . . 52
4.6 Contextual Analysis . . . 53
4.6.1 Study Setup . . . 53
4.6.2 Results . . . 55
4.7 Historical Analysis . . . 57
4.7.1 Study Setup . . . 58
4.7.2 Results . . . 59
4.8 Change Analysis . . . 64

4.8.1 Study Setup . . . 64
4.8.2 Results . . . 65
4.9 Bad Smells Analysis . . . 68
4.9.1 Study Setup . . . 68
4.9.2 Results . . . 69
4.10 Inequality Analysis . . . 72
4.11 Threats to Validity . . . 73
4.12 Final Remarks . . . 74

5 Validating Relative Thresholds with Developers 77
5.1 Study Design . . . 77
5.1.1 Research Questions . . . 77
5.1.2 Corpus and Metrics . . . 78
5.1.3 Methodology and Participants . . . 79
5.2 Results . . . 80
5.2.1 Relative Thresholds for the Pharo Corpus . . . 80
5.2.2 RQ 9: Do systems perceived as well-written by the expert developers follow the derived relative thresholds? . . . 80
5.2.3 RQ 10: Do systems perceived as poorly-written by the expert developers do not follow the derived relative thresholds? . . . 84
5.2.4 RQ 11: Do the noncompliant systems require more effort to maintain? . . . 85
5.3 Threats to Validity . . . 87
5.4 Final Remarks . . . 88

6 Conclusion 89
6.1 Summary . . . 89
6.2 Applications of Relative Thresholds . . . 91
6.2.1 A Comparative Study on Metric Thresholds for Software Product Lines . . . 91
6.2.2 Extracting Relative Thresholds for Feature Annotations Metrics 92
6.2.3 Using Relative Thresholds in Industrial Context . . . 94
6.3 Contributions . . . 94
6.4 Further Work . . . 95

Bibliography 97

Chapter 1

Introduction

In this chapter, we start by presenting our motivation (Section 1.1). Next, we present our problem statement (Section 1.2) and an overview of the proposed solution (Section 1.3). Finally, we present the outline of the thesis (Section 1.4) and our publications (Section 1.5).

1.1 Motivation

With software systems constantly growing in complexity and size, better support is required for measuring and controlling software quality [40]. Essentially, software quality is the degree to which a software meets its requirements [47]. However, evaluating a software system in order to improve its overall quality is not a trivial task. To this purpose, Meyer proposed a set of properties that can be used to evaluate software quality [73]. According to the author, software quality can be evaluated by external factors, *i.e.*, those factors perceived by users, and by internal factors, *i.e.*, those factors only perceived by the development team (developers and maintainers).

Since the inception of the first programming languages, we are witnessing the proposal of a variety of metrics to measure both internal and external quality factors [1, 22, 34, 45, 54, 61]. For example, internal quality factors can be measured by source code metrics, including properties such as modularity, coupling, cohesion, size, inheritance, and complexity. External quality factors include properties such as efficiency, correctness, robustness, extensibility, reusability, and ease of use. These metrics provide a quantitative measure of a wide spectrum properties of a software system and they can be used to control the software development and maintenance process. Particularly, software quality managers can rely on metrics to evaluate and control the internal and external quality of a software system, *e.g.*, to certify new components

or to monitor the degradation in quality that happens due to software aging. Metrics can also be used to compare and rate the quality of software products, and thus help to define acceptance criteria or service-level agreements between software producers and clients [5, 58]. In spite of such potential benefits of metrics, they are rarely used to control in an effective way the quality of software products [34]. To promote the use of metrics as an effective measurement instrument, it is essential to establish credible thresholds [5, 35, 44, 92]. Metric thresholds are defined by Lorenz and Kidd [67] as:

"Heuristic values used to set ranges of desirable and undesirable metric values for measured software. These thresholds are used to identify anomalies, which may or may not be an actual problem."

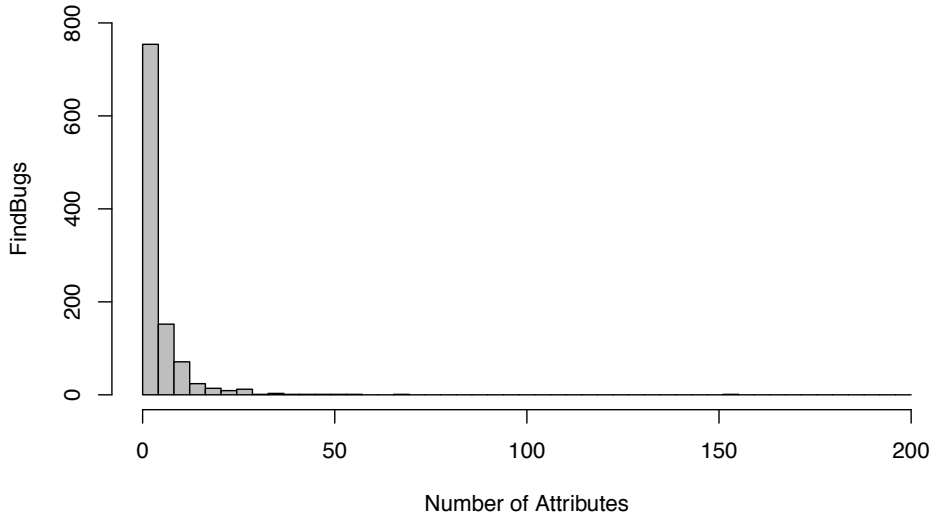
Thresholds have been already defined for many metrics. For example, McCabe proposed a threshold value of 10 for his complexity metrics, beyond which a subroutine is deemed unmaintainable and untestable [72]. Another example is that industrial code standards for Java recommend that classes should have no more than 20 methods and that methods should have no more than 75 lines of code [18]. These threshold values are inspired by personal experience and therefore they are not intended as universally applicable. Recently, Alves *et al.* proposed a more transparent method to derive thresholds from benchmark data [5]. They illustrate the application of the method in a large software corpus and derived, for example, thresholds stating that methods with McCabe complexity above 14 should be considered as very-high risk. However, for most metrics, thresholds are still missing or they do not generalize beyond the context of their inception.

1.2 Problem Statement

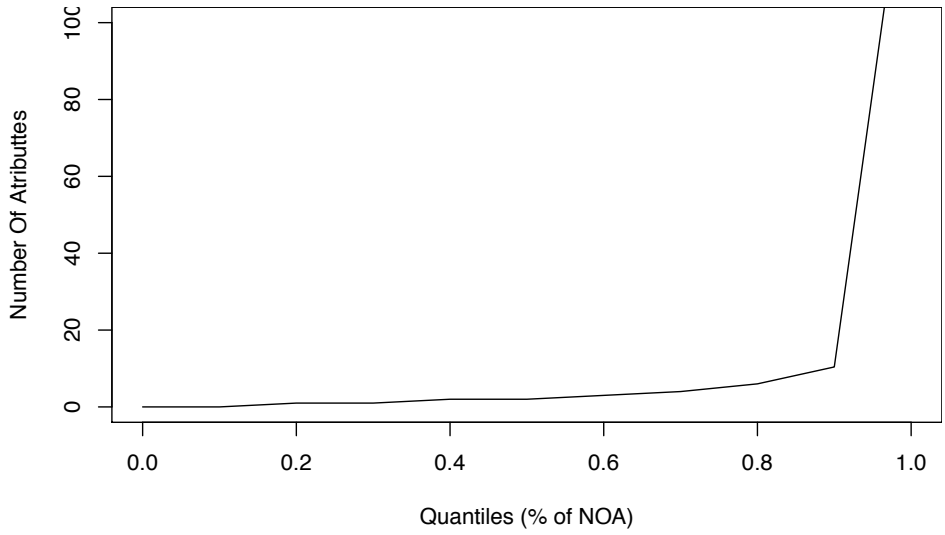
The definition of thresholds for source code metric is not a trivial task, because metric values usually follow right-skewed or heavy-tailed distributions [13, 64, 68, 86]. Heavy-tailed distribution are found in many object-oriented properties, describing a common behavior which states that there are few very complex modules while most modules have low complexity [76].

To illustrate such distributions, we will use the distribution of the number of attributes (NOA) for the *FindBugs* system. *FindBugs* is a system that uses static analysis to search for bugs in Java code. Figure 1.1a plots the histogram of the NOA values for the 1,047 classes in *FindBugs*. The x-axis represents the metric values and the y-axis represents the number of classes that have the metric value (frequency).

The histogram is highly right-skewed, meaning that while the bulk of the distribution occurs for fairly small sizes—most classes have few attributes (NOA $\leq$ 10)—there is a small number of classes with NOA much higher than the typical value, producing the long tail to the right of the histogram. In order to allow an alternative visualization of the full metric distribution, Figure 1.1b depicts the distribution of the NOA values for *FindBugs* using a quantile plot. The x-axis represents the percentage of observations (percentage of classes) and the y-axis represents the NOA metric values. In Figure 1.1b we can observe that 90% of classes have NOA $\leq$ 10.



(a)



(b)

Figure 1.1. (a) Histogram of the number of attributes(NOA) for the classes in *FindBugs*. (b) Quantile plot for the same data.

Therefore, in this PhD thesis we assume that in most systems it is "natural" to have source code entities not respecting the existing metric thresholds for several reasons, including complex requirements, performance optimizations, machine-generated code, etc. In the particular case of coupling for example, a recent study shows that high coupling can never be entirely eliminated from software design and that in fact some degree of high coupling might be quite reasonable [96].

Existing methods for extracting metric thresholds rely for example on the personal experience of software quality experts [18, 25, 72, 75], on standard statistical measures (*e.g.*, arithmetic mean and standard deviation) [32, 61], machine learning algorithms [44], log transformations [92], and linear regression [14]. There also methods that rely on benchmark for derive thresholds [5, 35]. Thus, a method to define metric thresholds should consider the right-skewed behavior normally observed in source code metric values, as widely reported in the literature.

1.3 Goals and Contributions

We claim in this PhD thesis that metric thresholds should be complemented by a second piece of information, denoting the percentage of entities the upper limit should be applied to. In this context, the **main goal** of this PhD thesis is to propose and to validate the concept of *relative thresholds* for evaluating source code metrics. Basically, we propose an empirical method to derive relative thresholds based on the analysis of a software corpus. A relative threshold is represented by pairs $[p, k]$ and have the following format:

$$\boxed{\textbf{p\%} \text{ of the } \textbf{entities} \text{ should have } \mathbf{M \leq k}}$$

where M is a source code metric calculated for a given software entity (method, class, etc.), k is an upper limit, and p is the minimal percentage of entities that should follow this upper limit. For example, a relative threshold can state that "80% of the classes should have NOA $\leq$ 8". Essentially, this threshold expresses that high risk classes impact the quality of a system whenever they represent more than 20% of the whole population of classes. In other words, the percentage of classes that exceeds the upper limit k do not constitute a threat to the internal quality of the entire project nor an indication of an excessive technical debt [29, 71].

Relative thresholds should constitute a trade-off between *real design rules*, widely followed by the systems in the considered corpus, and the need to reflect *idealized design rules*, based on accepted software quality principles [61]. Indeed, while a threshold

stating that "99% of the classes should have less than 100 attributes" is probably satisfied by most systems in any corpus, it is hardly useful or can be seen as reflecting an acceptable quality principle.

Figure 1.2 presents an overview of our method. Initially, we assume that the values of p and k that characterize a relative threshold for a metric M should emerge from a curated set of systems, which we call our *Corpus*. A relative threshold $[p, k]$ is derived using two functions, called *ComplianceRate* and *ComplianceRatePenalty*. The function *ComplianceRate* $[p, k]$ returns the percentage of systems in the *Corpus* that follow the relative threshold defined by the pair $[p, k]$. However, this function on its own is not sufficient to optimize p and k. Hence, we introduce the notion of *penalties* to find the values of p and k. We penalize a *ComplianceRate* function in two situations. The first penalty fosters the selection of thresholds followed by at least 90% of the systems in the *Corpus*. The goal is to derive thresholds that reflect real design rules, which are widely common in the *Corpus*. Furthermore, *ComplianceRate* $[p, k]$ receives a second penalty whenever k is greater than metric values that are perceived as being very high. Finally, the *ComplianceRatePenalty* function is the sum of $penalty_1[p, k]$ and $penalty_2[k]$. A derived relative threshold is the one with the lowest $ComplianceRatePenalty[p, k]$. A detailed description of our method is presented in the Chapter 3.

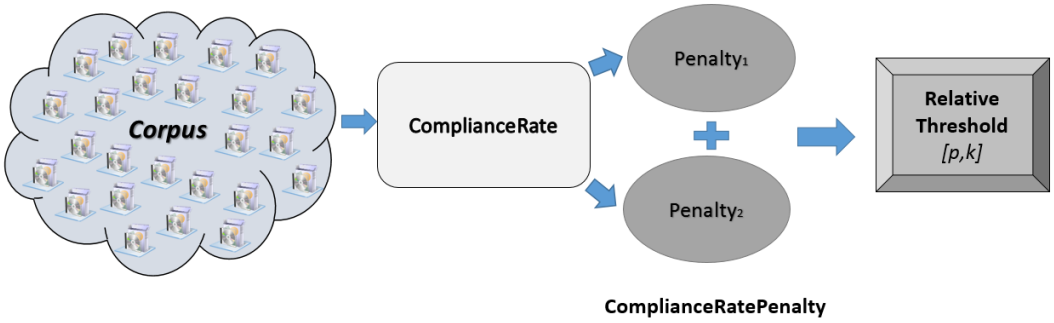


Figure 1.2. Relative threshold method

This PhD thesis makes five major **contributions**. First, we provide a review of the state-of-the-art with respect to statistical properties of source code metrics and on methods to derive metric thresholds. Second, we introduce a novel method to derive source code metric thresholds based on the analysis of a software corpus. Third, we implemented a prototype tool called *RTTool* that automates our method. Fourth, we evaluate the use of the proposed method in 106 real-world Java systems, using six source code metrics. Fifth, we describe a validation study with expert developers,

who are the right experts to check whether metric thresholds are indeed able to infer maintainability and design problems.

1.4 Thesis Outline

This thesis is structured in the following chapters:

- *Chapter 2* provides a general discussion on software quality and source code metrics. This chapter also presents related work to our research, such as statistical properties of source code metrics and methods to derive source code metrics thresholds.

- *Chapter 3* presents our method to extract relative thresholds from the measurement data of a benchmark of software systems. An illustrative example of the proposed method is also presented. Finally, the chapter discusses some aspects and properties of the proposed method. We conclude by presenting *RTTool*, an open source tool that automates our method.

- *Chapter 4* reports an extensive evaluation, through which we apply our method to extract relative thresholds for six source code metrics using Qualitas Corpus. Section 4.4 investigates whether popular open source Java repositories, available at GitHub, follow the relative thresholds; Section 4.5 compares our results with thresholds extracted using a method proposed by the Software Improvement Group (SIG method); Section 4.6 evaluates the influence of context in our results; Section 4.7 checks how the proposed thresholds apply to different versions of the systems under analysis; Section 4.8 investigates the importance of classes that do not follow the upper limit of a relative threshold, by checking how often such classes are changed; Section 4.9 investigates the relation between the presence of bad smells in a system and its adherence to the proposed relative thresholds; Section 4.10 evaluates the dispersion of the metric values in the systems respecting the proposed thresholds, using the Gini coefficient.

- *Chapter 5* reports the results of a final study designed to validate our method to extract relative thresholds. We extract thresholds from a benchmark of 79 Pharo/Smalltalk systems, which are validated with five Pharo experts and 25 Pharo developers.

- *Chapter 6* presents the final considerations of this PhD thesis, including applications of relative thresholds conducted by other authors, contributions, and future work.

1.5 Publications

This PhD thesis generated the following publications and therefore contains material from them:

1. Reference [80]: Paloma Oliveira, Marco Tulio Valente, Alexandre Bergel and Alexander Serebrenik. Validating Metric Thresholds with Developers: an Early Result. In 31th International Conference on Software Maintenance and Evolution- Early Research Achievements (ICSME - ERA Track), pages 546—550, 2015.

2. Reference [79]: Paloma Oliveira, Fernando Lima, Marco Tulio Valente and Alexander Serebrenik. *RTTool*: A Tool for Extracting Relative Thresholds for Source Code Metrics. In 30th International Conference on Software Maintenance and Evolution (ICSME - Tool Demo Track), pages 629—632, 2014.

3. Reference [81]: Paloma Oliveira, Marco Tulio Valente and Fernando Lima. Extracting Relative Thresholds for Source Code Metrics. In IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), pages 254—263, 2014.

4. Reference [78]: Paloma Oliveira, Hudson Borges, Marco Tulio Valente and Heitor Costa. Metrics-based Detection of Similar Software. In 25th International Conference on Software Engineering and Knowledge Engineering (SEKE), pages 447—450, 2013.

5. Reference [83]: Paloma Oliveira; Hudson Borges; Marco Tulio Valente; Heitor Costa. Uma Abordagem para Verificação de Similaridade entre Sistemas Orientados a Objetos. Em XI Simpósio Brasileiro de Qualidade de Software (SBQS), pages 1—15, 2012.

Chapter 2

Background

In this chapter, we discuss background work related to our PhD thesis. First, Section 2.1 provides a discussion about software quality. Second, Section 2.2 provides an overview on source code metrics. Third, Section 2.3 describes different statistical distributions, which are used to describe source code metrics. Moreover, we also present related work that use such distributions to study software metrics. Fourth, Section 2.4 discusses methods to extract thresholds. Finally, Section 2.6 concludes this chapter with a general discussion.

2.1 Software Quality

The primary goal of software engineering is to produce high quality software. Many definitions of software quality are proposed in the literature and the focus of most of them is the attendance of the customer needs [54, 85, 95].

Software quality can be reach using two important concepts: Software Process Quality and Software Product Quality [85]. In the next sections, we describe these two concepts of software quality.

2.1.1 Software Process Quality

A software process is a set of activities, practices, methods, and transformations used to develop and to maintain software and the associated products (*e.g.*, project plans, design documents, code, test cases, and user manuals) [85]. The adopted development process reflects in productivity, cost, and in the software quality [45].

Currently, there are several reference models for improving the software process that are widely accepted by software organizations and professionals. The most

known models are CMMI-DEV— Capability Maturity Model Integration for Development [89], ISO/IEC 15504 or SPICE [50], ISO/IEC 9000 [51], and MR-MPS—Reference Model for Software Process Improvement [94].

These reference models focus on processes improvement defining generic practices, requirements, and guidelines to help organizations to reach their goals in a more structured and efficient way. They contain essential elements of effective processes for one or more disciplines and they describe an evolutionary improvement path from ad hoc, immature processes to disciplined, mature ones with improved quality and effectiveness [23].

CMMI, SPICE, and MPS organizations are appraised to a certain compliance level defining the extent to which the organization follows the defined guidelines. These levels are called maturity level in CMMI and MPS, and capability level in SPICE. Moreover, ISO/IEC 9000 organizations are certified via a certification body. A process maturity model provides a indication of the process "maturity" presented by a software organization [85]. A key aspect to the success of these models is the fact that they provide foundations for measurement, comparison, and evaluation.

2.1.2 Software Product Quality

Software product quality has been given less importance when compared to other areas of software quality, with exception for testing. For a long time, reliability (as measured in number of failures) has been the single criteria for gauging software product quality [49]. In this section, we present an overview about software product quality.

The recognition of the need of a well-defined criteria for software product quality lead to the development of the standards ISO/IEC 9126[1] [49] and ISO/IEC 14598[2] [48]. ISO/IEC 9126 and 14598, which are closely related to each other. More recently, a new standard was proposed named ISO/IEC 25000[3], also known as SQuaRE [52]. SQuaRe is a standard family that combines and replaces the older ISO/IEC 9126 and the ISO/IEC 14598.

SQuaRe defines a complete evaluation process for a software product and it assists in specifying and evaluating of the quality requirements [52]. SQuaRE recommends the use of a quality model, which refines the required quality into characteristics and sub-characteristics and clarifies the relationship among them. SQuaRE is divided into five different divisions[4], as follow [52]:

[1] ISO/IEC 9126—International Standard for Software Engineering—Product Quality.
[2] ISO/IEC 14598—International Standard for Software Engineering—Product evaluation.
[3] ISO/IEC 25000—Software Quality Requirements and Evaluation standard family.
[4] "n" indicates numbers stand for one of the 10 digits.

1. Quality Management Division (ISO/IEC 2500n): The standard proposed by this division defines all common models, terms, and definitions referred further by all other standards from SQuaRE. This division also provides requirements and guidance for a support function which is responsible for the management requirement specification and evaluation.

2. Quality Model Division (ISO/IEC 2501n): The standard proposed by this division presents a detailed quality model including characteristics for internal and external software quality, and software quality in use. Furthermore, the internal and external software quality characteristics are decomposed into sub-characteristics.

3. Quality Measurement division (ISO/IEC 2502n): The standard proposed by this division includes a software product quality measurement reference model, mathematical definitions of quality measures, and practical guidance for their application. Moreover, this division also defines general requirements for quality metrics and guides the users to use those metrics.

4. Quality Requirements Division (ISO/IEC 2503n): The standard proposed by this division helps specifying quality requirements. These requirements can be used in the process of quality requirement elicitation for a software product or as input for an evaluation process.

5. Quality Evaluation Division (ISO/IEC 2504n): The standard proposed by this division defines general requirements for software quality specification and evaluation.

To summarize, the SQuaRE standard replaced the ISO/IEC 9126 and ISO/IEC 14598 standards. SQuaRE binds into one standard family providing best practices and lessons learned from both ISO/IEC 9126 and ISO/IEC 14598 standards. The differences between SQuaRE, ISO/IEC 9126, and ISO/IEC 14598 standards are as follow: (i) the introduction of a reference model; (ii) the introduction of measurement primitives; (iii) the introduction of quality requirement division; and (iv) an adapted version of evaluation process [52].

Software products are getting larger in size and in number of components, where different components exchange information using several interfaces to other components. This means that the overall complexity of the systems grows. It is has been estimated that 50-80% of costs of the software project goes to maintenance [54]. This is the reason why it is important for a software company to understand the quality of their products in order to increase efficiency of the software development. One of

the challenges of software quality research is to identify how to use metrics to drive the development processes and to improve the software product. Then, Section 2.2 presents an insight to the most popular source code metrics suites. Next, Section 2.3 discuss some distinguishing statistical properties of source code metrics.

2.2 Source Code Metrics

Source code metrics can be used to identify possible problems or chances for improvements in software quality [34, 85]. A variety of metrics to measure source code properties like size, complexity, cohesion, and coupling have been proposed [1, 10, 22, 58, 61]. However, source code metrics are rarely used to support decision making because they are ultimately just numbers that are not easy to interpret [85, 95]. Usually, metrics are classified into three categories: process, project, and product, as described next [45, 54]:

- **Process metrics:** enable the organization to evaluate the development process. They can be used to improve software development and maintenance practices. As examples of process metrics, we can mention function point, change metrics, number of files involved in bug fixing, etc.

- **Project or resources metrics:** enable the organization to evaluate the progress of a software project. Basically, they describe the project characteristics and execution. As examples of project or resources metrics, we can mention number of developers, cost, schedule, and productivity.

- **Product metrics:** enable software engineers to evaluate internal properties of a software product. As examples of product metrics, we can mention size, complexity, coupling, and cohesion.

Particularly, in this PhD thesis, we focus on product metrics, since they are most adequate to the quantitative assessment of internal quality of software systems [5, 34, 85]. Whitmire describes nine distinct and measurable characteristics for product metrics [106]:

1. Size: it is usually defined in terms of four views: population (static count of entities), volume (dynamic count of entities), length, and functionality (an indirect indication of the value delivered to the customer by a system).

2. Complexity: it is defined in terms of structural characteristics by examining how classes of an object-oriented design are interrelated to each other.

3. Coupling: it is the physical connections between entities of the system.

4. Sufficiency: it compares the abstraction from the point of view of the current application.

5. Completeness: it has an indirect implication about the degree to which the abstraction or design component can be reused.

6. Cohesion: it is determined by examining the degree to which the set of properties it possesses is part of the problem or design domain.

7. Primitiveness: it is the degree to which a method is atomic. It is related to simplicity of entities.

8. Similarity: it is the degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose.

9. Volatility: it measures the likelihood that a change will occur.

Each characteristic is associated with a set of metrics, moreover, a particular metric may be associated with more than one characteristic. In the following sections, we discuss the CK metrics suite that provides an indication of quality at object-oriented systems.

2.2.1 The CK Metrics Suite

Chidamber and Kemerer have proposed one of the most widely referenced sets of object-oriented software metrics [21, 22]. Often referred to as the CK metrics suite, it includes six class-based design metrics:

1. Weighted Methods per Class (WMC): represents the complexity of the class as measured by its methods. The calculation of the metric is given by the sum of the complexity of the methods in the class. According to Chidamber and Kemerer, WMC is an indicator of how much time and effort are required to develop and maintain a given class. Currently, some authors define WMC as the number of methods in the class.

2. Depth of Inheritance Tree (DIT): indicates the depth of a class in the inheritance tree, which is given by the length of the path from the class to the root of the tree. DIT is nowadays considered an indicator of design complexity.

3. Number of Children (NOC): denotes the number of immediate subclasses of a class. This metric is an indicator of the importance that a class has in the system. If a class has a large number of children, it might, for example, require more tests.

4. Coupling between Object Class (CBO): indicates the number of classes to which a certain class is coupled to. For Chidamber and Kemerer, a coupling between two classes exists when the methods implemented in one class use methods or instance variables defined by other classes. This metric can be used to reveal design problems. For example, it is widely accepted that excessive coupling is harmful to modular design, because the more independent a class is, more easy is to reuse it in other systems.

5. Response for a Class (RFC): indicates the number of methods that can be called in response to a message received by a class, defined as the number of methods of the class plus the number of methods invoked by them. RFC is considered an indicator of coupling.

6. Lack of Cohesion in Methods (LCOM): indicates the lack of cohesion between the methods in a class. Chidamber and Kemerer propose that cohesion between methods can be captured by the use of common instance variables. In this way, LCOM is usually computed as the number of method pairs that have no instance variables in common minus the number of method pairs with common instance variables. Therefore, the smaller the value of LCOM, the more cohesive is the class.

In summary, CK metrics cover different internal properties of software systems, such as complexity (WMC), coupling (CBO and RFC), inheritance (DIT and NOC), and cohesion (LCOM). It is also important to state that, there are other object-oriented metrics cited in the literature [1, 61, 67]. Among such metrics, we can mention number of lines of code (SLOC), number of methods (NOM), number of attributes (NOA), number of other classes referenced by a class (FAN-OUT), etc.

In order to use source code metrics as an effective instrument of measurement is interesting to understand the statistical distribution that better describe their data. Thus, in the Section 2.3, we discuss about statistical properties of source code metrics.

2.3 Statistical Properties of Source Code Metrics

There are many studies on the distribution of source code metrics. However, usually all of such studies indicate that source code metric values follow right-skewed or heavy-tailed distributions [2, 9, 13, 64, 68, 84, 86, 105]. Heavy-tailed is a distribution that has been found in many object oriented properties. A heavy-tailed describes a common behavior which states that there are few very complex modules while most modules have low complexity [76].

A classic example of this type of distribution is the size of towns and cities [76]. Figure 2.1 plots the histogram of the size of cities. In figure (a) is showed a simple histogram of the distribution of US city sizes. The histogram is highly right-skewed, meaning that while the bulk of the distribution occurs for fairly small sizes—most US cities have small populations—there is a small number of cities with population much higher than the typical value, producing the long tail to the right of the histogram. Figure 2.1 (b) shows the histogram of city sizes again, but this time replotted using a logarithmic scale in the horizontal and vertical axes. As can be observed, a remarkable pattern emerges: the histogram follows a straight line [76, 110].

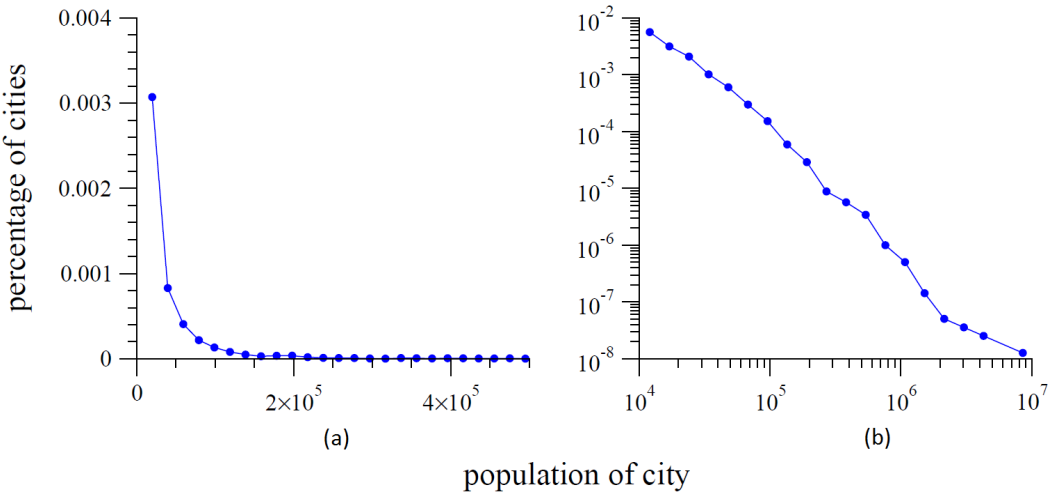


Figure 2.1. (a) histogram of the populations of all US cities with population of 10 000 or more. (b) another histogram of the same data, but plotted on logarithmic scales. The approximate straight-line form of the histogram in the right panel implies that the distribution follows a heavy-tailed. Data from the 2000 US Census. Figure and caption originally used by Newman *et al.* [76]

There are many heavy-tailed distributions, which Power law is one of the more cited in the literature on source code metrics analysis. Mathematically, a quantity x follows a power law if it is drawn from a probability distribution

$$p(X = x)\ \alpha\ Kx^{(-\alpha)} \quad (1)$$

where α is a constant parameter of the distribution known as the exponent or scaling parameter. The scaling parameter typically lies in the range $2 < \alpha < 3$, although there are exceptions. In practice, few empirical phenomena obey power laws for all values of x. More often the power law applies only for values greater than some minimum x_{min}. In such cases, we state that the tail of the distribution follows a power law.

Another examples of heavy-tailed distributions are Pareto, Lognormal, Exponencial, Caughy, and Weibull etc. [13, 35, 37, 76, 104]. There are several approaches to check whether a population follow a heavy-tailed distribution, which we highlight four:

1. **Histogram and doubly logarithmic plot**: This is a visual approach and it is most often used. This approach consists in plotting a histogram, applying linear regression, and taking the logarithm of both sides of Equation 1. We can see that the distribution obeys $\ln(p(x)) = \mathrm{K} - \alpha \ \ln(x)$, implying that it follows a straight line on a doubly logarithmic plot. A common way to check for power-law behavior, therefore, is to measure a variable of interest x, construct a histogram representing its frequency distribution, and plotting that histogram on doubly logarithmic axes. If we discover a distribution that approximately falls on a straight line, we can say that the distribution follows a heavy-tailed, with a scaling parameter α given by the absolute slope of the straight line. Typically, this slope is extracted by performing a least-square linear regression on the logarithm of the histogram. Although this approach is frequently described in the literature, it is subjected to systematic errors under relatively common conditions, and as a consequence the results it provides might not be reliable [24, 103].

2. **Statistical approach proposed by Clauset *et al.* [24]**: this approach is a statistical framework for discerning and quantifying heavy-tailed behavior in empirical data. It combines maximum-likelihood fitting methods with goodness-of-fit tests based on the Kolmogorov-Smirnov statistic and likelihood ratios. It also uses numeric methods to estimate the parameters X_{min} and α, where X_{min} indicates the start of the tail and α represents the scaling parameter of the dataset. Next, the approach calculates the goodness-of-fit between the data and the power law. If the resulting *p-value* is greater than 0.1 the heavy-tailed is a plausible hypothesis for the data, otherwise it is rejected.

3. **Quantile Function**: this approach examines a distribution of values and plots its Cumulative Density Function (CDF) or the CDF inverse, the Quantile function. The use of the quantile function is interesting to determine thresholds (the

dependent variable) as a function of the percentage of observations (independent variable). Also, by using the percentage of observations instead of the frequency, the scale becomes independent of the size of the system making it possible to compare different distributions. Moreover, the quantile function allows for better visualization of the full metric distribution. Therefore, in this PhD thesis all distributions are depicted with quantile plots. Alves *et al.* also use this approach [5].

4. **Adherence test**: This approach is also frequently mentioned in the literature [13, 35]. It relies on rigorous best-fits to several distributions, and checks first whether it is reasonable to fit a heavy-tailed, second whether a given distribution is more reasonable than the others, third whether the data can be divided into two or more groups according to which distribution fits "best".

2.3.1 Metrics Values Distributions

Wheeldon and Counsell analyzed three Java systems: JDK (Java Development Kit), Apache Ant, and Tomcat using 12 metrics related to object-oriented coupling, namely, inheritance, aggregation, interface, parameter type and return type [105]. These metrics are: Number of methods(nM), Number of fields(nF), Number of constructors(nC), Subclasses (SP), Implemented interfaces (IC), Interface implementations (IP), References to class as a member (AP), Members of class type (AC), References to class as a parameter (PP), Parameter-type class references (PC), and References to class as return type (RP). To identify the power laws the authors perform linear regression on log-log data plots. They concluded that all metric values follow Power Law distributions.

Baxter *et al.* analyzed 17 metrics in 56 Java systems for verifying their internal structure [13]. The authors performed adherence tests to identify three types of distribution: Power Laws, Lognormal, and Exponential. The goal of this work is to extend the work proposed by Wheeldon and Counsell [105] in order to check heavy-tailed distribution in 17 object-oriented metrics. However, they added the following metrics: Methods returning classes (RC), Depends on (DO), Depends on inverse (DOinv), Public method count (PubMC), Package size (PkgSize), and Method size (MS). The authors report that, AP, PP, RP, SP, IC, and MS are metrics that follow heavy-tailed distributions. But AC, PC, RC, PubMC, nF, nM, Do, IP, and DoInv do not follow such distributions. Finally, the results for nC and Ms are not conclusive.

However, Louridas *et al.* analyzed coupling metrics using 11 systems developed in multiple languages (C, Perl, Ruby, and Java) using coupling metrics: FAN-IN and FAN-OUT [68]. The authors concluded that most metrics are in conformity with

heavy-tailed distributions, independently of programming language. These findings are different than those of Baxter *et al.*, which suggests that out-degree metrics are not heavy-tailed [13]. Studies conducted by Potanin *et al.* [84], Gao *et al.* [41], and Taube-shock *et al.* [96] confirm such results for coupling metrics. Potanin *et al.* analyzed 35 systems and they concluded that coupling metrics are in conformity with heavy-tailed distributions [84]. Gao *et al.* analyzed four open source Java systems and they also concluded that out-degree and in-degree metrics are in conformity with heavy-tailed distributions [41]. Taube-shock *et al.* analyzed coupling metrics using 97 open source Java systems from the Qualitas Corpus [96]. The goal of this work was checking the following hypothesis: (i) the between-module connectivity network of source code entities follows a heavy-tailed distribution; and (ii) The between-module connectivity network of source code entities follows a heavy-tailed distribution, and the degree of left skewness has some maximum level. The authors concluded that these two hypothesis can be accepted and that high coupling is impracticable to eliminate entirely from software design.

Concas *et al.* examined 10 source code metrics of three systems: one implemented in Smalltalk (VisualWorks) and two implemented in Java (JDK e Eclipse) [26]. The goal of this work was to check whether large object-oriented systems follow heavy-tailed distributions. Jing *et al.* found heavy-tailed in the values of Weighted Methods per Class (WMC) and Coupling Between Objects (CBO) for four open source software systems [53]. Ichii *et al.* examined four source code metrics on six systems, finding that in-degree follows a Power Law while out-degree follows other heavy-tailed distribution [46].

Queiroz *et al.* analyzed the scattering degree of # `ifdefs` in five C-pre-processor-based systems (vi, libxml2, lighttpd, MySQL, and Linux kernel) [86]. In the case of four systems, they reported that feature scattering has characteristics of heavy-tailed distributions, with a good-fit with power laws. Vasa *et al.* noted that many software metrics have a skewed distribution, which makes the reporting of data using central tendency statistics unreliable [100]. To address this, they recommended adopting the use of the Gini coefficient, which has been used in the field of economics to characterize the relative equality of distributions. They examined 50 systems developed in Java and C# using 10 metrics. Landman *et al.* also found evidences of skewed distributions with a long tail for two metrics: cyclomatic complexity and lines of source code. For this, they used a corpus of 17.8M methods from 13K open source Java projects [60]. Lin and Whitehead analyzed four object-oriented Java systems and they also found heavy-tailed distributions in measures such as file size, change size, and in-degree of methods [64].

2.3.2 Discussion

In this section, we provided a discussion about source code metrics distributions, which is summarized in Tables 2.1 and 2.2. We reported that source code metrics tend to follow a heavy-tailed distribution. This means that, typically, software systems follow this pattern: few software entities contain much of the complexity and functionality, whereas the others define simple data abstractions and utilities [101]. Moreover, since non-Gaussian distributions are common in the case of source code metric values descriptive statistics, *e.g.*, mean and variance, are not adequated to define thresholds for such metric data. Although, the works reported in this section have theorical value, they fall short in concluding how to use these distributions and their coefficients in practical terms, to establish baseline values to judge systems. Therefore, the next section presents methods to derive source code metric thresholds.

Table 2.1. Source code metric distributions

Authors	**# Systems**	**Language**
Wheeldon and Counsell [105]	3	Java
Baxter *et al.* [13]	56	Java
Louridas *et al.* [68]	11	C, Perl, Ruby, and Java
Potanin *et al.* [84]	35	Java
Gao *et al.* [41]	4	Java
Taube-Schock *et al.* [96]	97	Java
Concas *et al.* [26]	3	Java and Smalltalk
Jing *et al.* [53]	4	Java
Ichii *et al.* [46]	4	Java
Queiroz *et al.* [86]	5	C
Vasa *et al.* [100]	47	Java and C#
Landman *et al.* [60]	13K	Java
Lin *et al.* [64]	4	Java

2.4 Thresholds Definitions

In this section, we present different methods to derive thresholds. These methods are organized in four groups: (a) extracting threshold using traditional techniques (Section 2.4.1); (b) extracting threshold from repositories (Section 2.4.2); (c) extracting threshold using error models (Section 2.4.3); and (d) extracting threshold using clustering algorithms (Section 2.4.4).

Table 2.2. Source code metric distributions

Authors	Method	Are heavy-tailed?
Wheeldon and Counsell [105]	linear regression on log-log data plots	All metrics
Baxter *et al.* [13]	log-log data plots and adherence test	Six out 15 metrics
Louridas *et al.* [68]	linear regression on log-log data plots	All metrics
Potanin *et al.* [84]	linear regression on log-log data plots	All metrics
Gao *et al.* [41]	linear regression on log-log data plots	All metrics
Taube-Schock *et al.* [96]	linear regression on log-log data plots	All metrics
Concas *et al.* [26]	log-log data plots and adherence test	All metrics
Jing *et al.* [53]	linear regression on log-log data plots	All metrics
Ichii *et al.* [46]	linear regression on log-log data plots	All metrics
Queiroz *et al.* [86]	histogram and Clauset Method	All metrics
Vasa *et al.* [100]	Gini coefficient	All metrics
Landman *et al.* [60]	linear regression on log-log data plots	All metrics
Lin *et al.* [64]	linear regression on log-log data plots	All metrics

2.4.1 Extracting Thresholds using Traditional Techniques

Erni and Lewerentz proposed the use of mean (μ) and standard deviation (σ) to derive a threshold T from project data [32]. For this, the authors used coupling, complexity, and cohesion metrics. A threshold T is calculated as $T_{low} = \mu+\sigma$ or $T_{high} = \mu-\sigma$, indicating that high or low values of a metric can cause problems, respectively. This method is a common statistical technique which data are normally distributed. However, the authors did not analyze the underlying distribution, and only applied it to one system, using three releases. Lanza and Marinescu also proposed a method based on descriptive statistics and experts experience [61]. They performed an experiment using source code metrics related to inheritance, coupling, size, and complexity. For this, the authors used 82 systems developed in C++ (37 systems) and Java (45 systems). This method consisted of a intervals of thresholds, where the mean as typical value and standard deviation as upper limit.

The problem with the use of these methods is that they assume that metric data are assumed to be normally distributed, thus compromising their validity in general. As mentioned in Section 2.3, software metrics generally follow heavy-tailed distributions. Consequently, the use of means and standard deviation is not adequate.

2.4.2 Extracting Thresholds from Repositories

Alves *et al.* proposed an empirical method to derive threshold values for source code metrics from a benchmark of systems [5]. Their ultimate goal was to use the extract thresholds to build a maintainability assessment model [8, 27, 42]. Specifically, the goal was to define quality profiles to rank entities according to four categories: low risk (0 to 70th percentiles), moderate risk (70th to 80th percentiles), high risk (80th to 90th percentiles), and very-high risk (90th percentile). For this purpose, metric values for a given program entity are first weighted according to the size of the entities in terms of lines of code (SLOC), in order to generate a new distribution where variations in the metrics values are more clear. They illustrated the method using as example the McCabe complexity metric [72] and a benchmark of 100 object-oriented systems, which it was implemented in C# (18 systems) and Java (82 systems), including both proprietary (77 systems) and open source (23 systems). The method of Alves *et al.* is summarized in six steps [5]:

1. Metrics extraction: the value of the metrics are extracted from a benchmark of systems. For each system S and for each entity E (*e.g.*, method or class), they record a metric value and weight metric. They considered as *weight* the SLOC of the entity. As example, for the Vuze system, there is a method (entity) called `MyTorrentsView.createTabs()` with a McCabe value of 17 and *weight* value of 119 SLOC;

2. Weight ratio calculation: in this step, for each entity E, they divide the entity *weight* by the sum of all *weights* of the same system. For each system, the sum of all entities *WeightRatio* must be 100%. As example, for the `MyTorrentsView.createTabs()` method entity, the result is 119 SLOC divided by 329,765 (total SLOC for Vuze) which represents 0.036% of the overall Vuze system;

3. Entity aggregation: they aggregate the weights of all entities per metric value, which is equivalent to computing a weighted histogram. As example, all entities with a McCabe value of 17 represent 1.458% of the overall SLOC of the Vuze system;

4. System aggregation: they normalize the weights for the number of systems and then aggregate the weight for all systems. Hence, they have a histogram describing a weighted metric distribution. As example, a McCabe value of 17 corresponds to 0.658% of all code in the benchmark they use to illustrate the method.

5. Weight ratio aggregation: in this step, a density function (or quantile function) is computed, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale. As example, in the benchmark they used, for 60% of the overall code the maximal McCabe value is 2.

6. Thresholds derivation: thresholds are extracted by choosing the percentage of the overall code we want to represent. For example, for McCabe metric the extracted thresholds are 6, 8 and 14, which represents 70%, 80%, and 90% quantiles.

The authors claimed that the distribution of the metric values is preserved and that the method is resilient to the influence of large systems or outliers. Thresholds were derived using 70%, 80% and 90% quantiles and checked against the benchmark to show that thresholds indeed represent these quantiles. This method was replicated using four other metrics from the SIG quality model: unit size, unit interfacing, FAN-IN, and module interface size. The method also was used by Luijten *et al.* to derive thresholds to other metrics of the SIG group [69]. Luijten *et al.*also found empirical evidence that systems with higher technical quality have higher issue solving efficiency. In a more recent work, Alves *et al.* improved their method to include the calibration of mappings from code-level measurements to system-level ratings, using an N-point rating system [4].

Ferreira *et al.* defined thresholds for six source code metrics from a benchmark with 40 open source Java systems. The analyzed metrics included coupling factor (COF), number of public fields (NPF), number of public methods (NPM), lack of cohesion in methods (LCOM), depth of inheritance tree (DIT), and afferent couplings (AC) [35]. The authors used EasyFit tool[5] to fit the data to various probability distributions, such as Bernoulli, Binomial, Uniform, Geometric, Hypergeometric, Logarithmic, Binomial, Poisson, Normal, t-Student, Chi-square, Exponential, Lognormal, Pareto, and Weibull. For each metric, the data was collected and two graphics were generated: a scatter plot and the same data in doubly logarithmic scale. Using the EasyFit tool, they concluded that the metric values, with exception of DIT, follow heavy-tailed distributions. After this conclusion, the authors established three threshold ranks: (i) good: refers to most common values; (ii) regular: refers to values with low frequency, but that are not irrelevant; and (iii) bad: refers to values with rare occurrences. However, they do not predefined the percentage of classes tolerated in these categories. For example, the LCOM threshold is: 0 (good cohesion), 1—20 (regular cohesion), and greater than 20 (bad cohesion).

[5] `http://www.mathwave.com/products/easyfit.html`

The authors extracted general thresholds for object-oriented software metrics, and thresholds by application domain, size, and system type (tool, library, and framework). They did not find relevant differences among them. The identified thresholds were evaluated in two case studies. The results of this evaluation indicated that the proposed thresholds can help to identify classes that violate design principles. Recently, Filo *et al.*extended and improved this work and they applied it to extract thresholds to 17 source code metrics using a benchmark with 111 open source Java systems [36].

The goal of the methods proposed by Alves *et al.* [5] and Ferreira *et al.* [35] is to rank entities, *i.e.*, classes or methods. In the work of Alves *et al.*, a new method was proposed—the use of weighting by size using SLOC metric. The goal of weighing by SLOC is to emphasize the metric variability when plotting the quantile function. Ferreira *et al.* extracted three thresholds for each metric, which are used to rank the classes as good, regular, or bad. In summary, this works extracted *absolute thresholds*, meaning that all classes with high value metric are considered as presenting high risk or bad quality. However, several works showed that source code metrics follow a heavy-tailed distribution. Consequently, in this type of distribution is natural to find entities with high values.

2.4.3 Extracting Thresholds using Error Models

Shatnawi *et al.* investigated the use of the ROC curves to extract thresholds for predicting the existence of bugs in different error categories [93]. They performed an experiment using 12 source code metrics and applied the method to three releases of Eclipse. The metrics analyzed were: number of attributes (NOA), number of operations (NOO), lack of cohesion of methods (LCOM), weighted methods complexity (WMC), coupling between objects (CBO), coupling through data abstraction (CTA), coupling through message passing (CTM), response for class (RFC), depth of inheritance hierarchy (DIT), number of child classes (NOC), number of added methods (NOAM), and number of overridden methods (NOOM). Catal *et al.* developed a noise detection approach that uses threshold values for software metrics in order to capture these noisy instances [20]. The thresholds of Catal *et al.* were calculated using an adaptation of the Shatnawi *et al.* [93] threshold calculation technique. They validated the proposed noise detection technique on five public NASA datasets. The results showed that this method is effective for detecting noisy instances. Although Shatnawi *et al.* and Catal *et al.* extracted thresholds using ROC curves, this method resulted in three drawbacks in their results. First, thresholds values can be not found. Second, for different releases of a system, different thresholds were derived. Third, the methodology does not

succeed in deriving monotonic thresholds, *i.e.*, lower thresholds were derived for higher error categories than for lower ones.

Benlarbi *et al.* analyzed the relation of source code metric thresholds and software failures using linear regressions [14]. This study was performed using five CK metrics (WMC, DIT, NOC, CBO, and RFC) and two C++ systems. The authors compared two error probability models, one with threshold and another without. For the model with threshold, zero probability of error exists for metric values below the threshold. They concluded that there was no empirical evidence supporting the model with threshold as there was no significant difference among the models. However, this result is only valid for this specific error prediction model and for the metrics the authors took into account. Other models can, potentially, give different results.

Herbold *et al.* used a machine learning algorithm to define a method for the calculation of metric thresholds [44]. For this, they analyzed 11 metrics related to size, coupling, complexity, and inheritance. In this work, an entity is analyzed according to a set of metrics and the global result is binary. This method is based on a given metric set M and a set of software entities X with known classifications Y. As result, the algorithm yields pairs of upper and lower bounds. Specifically, the thresholds T is zero (bad) when at least one metric m exceeds its threshold t, and is one (good) when none of the metrics exceeds its threshold. The authors performed four case studies using eight systems including C functions, C++, C# methods, and Java classes. The results showed that this method is able to improve the efficiency of existing metric sets. The proposed method, however, produces a binary classification and can therefore only differentiate between good and bad; further shades of gray are not possible. Another point is that the extracted thresholds are in entities level. Therefore, system level thresholds are not provided.

2.4.4 Extracting Thresholds using Clustering Algorithms

Yoon *et al.* investigated the use of the K-means clustering algorithm to identify outliers in the data measurements [108]. Outliers can be identified by observations that appear either in isolated clusters (external outliers), or by observations that appear far away from other observations within the same cluster (internal outliers). Oliveira *et al.* proposed a quantitative approach based in source code metrics to determine similarity in object-oriented systems [78, 83]. This approach also used K-means clustering algorithm to derive thresholds. The thresholds generated by this approach represents profiles of classes of a system. The authors performed two case studies using a dataset with more than 100 Java systems and 23 metrics.

However, K-means suffers from important shortcomings: it requires an input parameter that affects both the performance and the accuracy of the results. Thus, different thresholds can be extracted for the same dataset and metric.

2.4.5 Discussion

In this section, we provided a discussion about threshold extraction methods, which are summarized in Table 2.3 and 2.4. We observed that there are several methods for this purpose. However, there is not a method that is widely recognized by researchers and software engineers as an effective instrument to control the internal quality of software systems. We also observed that using benchmark of systems is an interesting approach, which tends to reflect the software development practice.

Table 2.3. Thresholds approaches

Authors	**Systems**	**Languages**	**Metrics**
Erni and Lewerentz [32]	1	Smaltalk	Complexity, coupling, and cohesion
Lanza and Marinescu [61]	82	C++ and Java	Inheritance, coupling, size, and complexity
Alves *et al.* [5]	100	C# and Java	McCabe complexity, unit size, unit interfacing, module interface size, and FAN-IN
Ferreira *et al.* [35]	40	Java	LCOM, DIT, COF, Afferent coupling, NOMP, and NOAP
Shatnawi *et al.* [93]	1	Java	CBO, RFC, WMC, LCOM, DIT, NOC, CTA, CTM, NOAM, NOOM, NOA, and NOO
Catal *et al.* [20]	5	C and C++	SLOC, MCave, EC, DC
Benlarbi *et al.* [14]	2	C++	WMC, DIT, NOC, CBO, and RFC
Herbold *et al.* [44]	8	C, C++, C# and Java	Size, coupling, complexity, and inheritance
Oliveira *et al.* [83]	86	Java	Size metrics
Oliveira *et al.* [78]	103	Java	Size, coupling, complexity, and cohesion

Table 2.4. Thresholds approaches

Authors	Method	Weaknesses
Erni and Lewerentz [32] Lanza and Marinescu [61]	mean and standard deviation	It requires an input parameter that affects both the performance and the accuracy of the results
Alves *et al.* [5]	quantile function analysis	The goal is to create quality profiles to rank entities
Ferreira *et al.* [35]	statistical distribution analysis	do not establish the percentage of classes tolerated in each category
Shatnawi *et al.* [93] Catal *et al.* [20]	ROC curves	This methodology does not succeed in deriving monotonic thresholds and thresholds values can be not found
Benlarbi *et al.* [14]	linear regression	There is no empirical evidence supporting the model
Herbold *et al.* [44]	machine learning	The methodology produces only a binary classification
Oliveira *et al.* [83] Oliveira *et al.* [78]	K-means algorithm	It requires an input parameter that affects both the performance and the accuracy of the results

2.5 Studies with Developers

In this section, we conclude by presenting some works which explore how developers rate different software quality attributes like readability [17], complexity [56], cohesion [30], and coupling [12]. Buse and Weimer explored the concept of code readability and investigate its relation to software quality [17]. This study involved 120 computer science students and they found that readability metrics correlate strongly with code changes, automated defects reports, and defect log messages. Katzmarski and Koschke investigated whether metrics agree with complexity as perceived by developers [56]. For this, they collected opinions from 206 developers. The authors concluded that data-flow metrics seem to better conform to developers opinions than control-flow metrics. Bavota *et al.* investigated how coupling metrics based on structural, dynamic, semantic, and logical information align with developers perception of coupling [12]. This study involved 64 developers, including students, academics, and industrial practitioners. The authors concluded that coupling is not a trivial quality attribute that can be captured and measured using only structural information. Silva *et al.* investigated what kind of cohesion metrics aligns with developers perception [30]. This study

involved 80 developers with different levels of experience and academic degree. They found that most of the developers are familiar with cohesion and that developers perceive cohesion as a measure of a class responsibilities. Moreover, the results showed that conceptual cohesion metrics capture the developers notion of cohesion better than traditional structural cohesion metrics. To the best of our knowledge, the study presented in Chapter 5 is the first on interviewing developers on metric thresholds.

2.6 Final Remarks

Measurement is a fundamental part of Software Engineering research and practice [95]. In this context, software metrics refer to measurements that can be applied to check the quality of processes, projects, and software products. Evaluating software quality through metrics allows to define quantitatively the success or failure of a particular attribute, identifying the needs of improvement. In this chapter, we provided a discussion about software quality and presented an overview of source code metrics, specifically, we presented the CK metric suite. We discussed the importance of considering the statistical distribution of software metrics in order to extracted credible thresholds. Next, we presented the state-of-the-art in methods to extract thresholds and we performed a critical appraisal of related work. Finally, we presented related work which explore how developers rate different software quality attributes.

In the next chapter, we introduce our method to derive relative thresholds. This method explicitly indicates that thresholds should be valid for most, but not for all classes in object-oriented systems.

Chapter 3

Proposed Method

This chapter presents the method to extract relative thresholds from a set of systems (Section 3.1). An illustrative example of its usage is presented in Section 3.2. Section 3.3 discuss some aspects and properties of the proposed method. Section 3.4 presents *RTTool*, an open source tool that automates our method.

3.1 Relative Thresholds

Software metrics have been proposed to analyze and evaluate software by quantitatively capturing a specific characteristic or view of a software system. Despite much research, the practical application of software metrics remains challenging.

We focus on source code metrics that follow heavy-tailed distributions, when measured at the level of classes as long as low(er) metric values are considered to be more desirable than the high(er) ones. Numerous metrics including NOA, NOM, FAN-OUT, RFC, and WMC satisfy these conditions [81, 91, 105]. Examples of a metric that does not follow the traditional heavy-tailed distribution and therefore should not be subject to our method are DIT (Depth of Inheritance) [35] and D_n [90].

Our goal is to derive *relative thresholds*, *i.e.*, pairs $[p, k]$ such that at least $p\%$ of the classes should have $M \leq k$, where M is a given source code metric and p is the minimal percentage of classes in each system that should respect the upper limit k. A relative threshold tolerates, therefore, $(100 - p)\%$ of classes with $M > k$.

We derive the values of p and k from a curated set of systems, which we call our *Corpus*. Figure 3.1 defines the functions used to calculate the parameters p and k for a given metric M. First, the function *ComplianceRate* $[p, k]$ returns the percentage of systems in the *Corpus* that follows the relative threshold defined by the pair $[p, k]$. The function *ComplianceRate* can be easily increased by increasing k or decreasing p.

Therefore, *ComplianceRate* on its own is not sufficient to optimize p and k. Hence, we introduce the notion of a *penalty* to find the values of p and k. We penalize a *ComplianceRate* function in two situations:

- A *ComplianceRate* $[p, k]$ less than 90% receives a penalty proportional to its distance to this percentile, as defined by function $penalty_1[p, k]$. As mentioned, the proposed thresholds should reflect real design rules that are widely common in the *Corpus*. Therefore, this penalty formalizes this guideline, by fostering the selection of thresholds followed by at least 90% of the systems in the *Corpus*. In other words, this penalty punishes systems that are somehow "atypical" in the *Corpus*.

- A *ComplianceRate* $[p, k]$ receives the second penalty proportional to the distance between k and the median of the 90-th percentiles, of the values of M in each system in the *Corpus*, denoted as *Median90*, as defined by function $penalty_2[k]$. We assume that *Median90* is an idealized upper value for M, *i.e.*, a value representing classes that, although present in most systems, have very high values of M[1].

$$
\begin{aligned}
\textit{ComplianceRate}\ [p,k] &= \frac{|\ \{\ S \in \textit{Corpus}\ |\ p\%\ \text{of the classes in}\ S\ \text{have}\ M \leq k\}\ |}{|\ \textit{Corpus}\ |} \\
\textit{penalty}_1[p,k] &= \begin{cases} \dfrac{90 - \textit{ComplianceRate}\ [p,k]}{90} & \text{if } \textit{ComplianceRate} < 90 \\ 0 & \text{otherwise} \end{cases} \\
\textit{penalty}_2[k] &= \begin{cases} \dfrac{k - \textit{Median90}}{\textit{Median90}} & \text{if } k > \textit{Median90} \\ 0 & \text{otherwise} \end{cases} \\
\textit{ComplianceRatePenalty}[p,k] &= \textit{penalty}_1[p,k] + \textit{penalty}_2[k]
\end{aligned}
$$

Figure 3.1. *ComplianceRate* and *ComplianceRatePenalty* functions

As defined in Figure 3.1, the final penalty of a given threshold is the sum of $penalty_1[p, k]$ and $penalty_2[k]$, as defined by function *ComplianceRatePenalty*. Finally,

[1]We selected the 90-th percentiles after experimental testings and we usually observed a fast growth of the metric values starting at the 90-th percentile.

the relative threshold is the one with the lowest $ComplianceRatePenalty[p, k]$. In case of ties, we defined a *tiebreaker criterion*: we select the result with the highest p and then the one with the lowest k.

3.2 Illustrative Example

To illustrate our method we derive a threshold for the Number of Attributes (NOA) metric, based on the systems in the Qualitas Corpus [97]. Figure 3.2 plots the values of the *ComplianceRate* function, for different values of p and k. As expected, for a fixed value of p *ComplianceRate* is a monotonically increasing function, on the values of k. Moreover, as we increase p the function starts to present a slower growth. This figure 3.2 shows the importance of $penalty_2$. For example, we can observe that $ComplianceRate\ [85, 17] = 100\%$, *i.e.*, in 100% of the systems at least 85% of the classes have NOA ≤ 17. In this case $Median90 = 9$, *i.e.*, the median of the 90th percentile for the NOA values in the considered *Corpus* is nine attributes. Therefore, the relative threshold defined by the pair $[85, 17]$ relies on a high value for k ($k = 17$) to achieve a compliance rate of 100%. To penalize a threshold like that, the value of $penalty_2$ is $(17 - 9)\ /\ 9 = 0.89$. Since $penalty_1 = 0$ (due to the 100% of compliance), we have that $ComplianceRatePenalty[85, 17] = 0.89$.

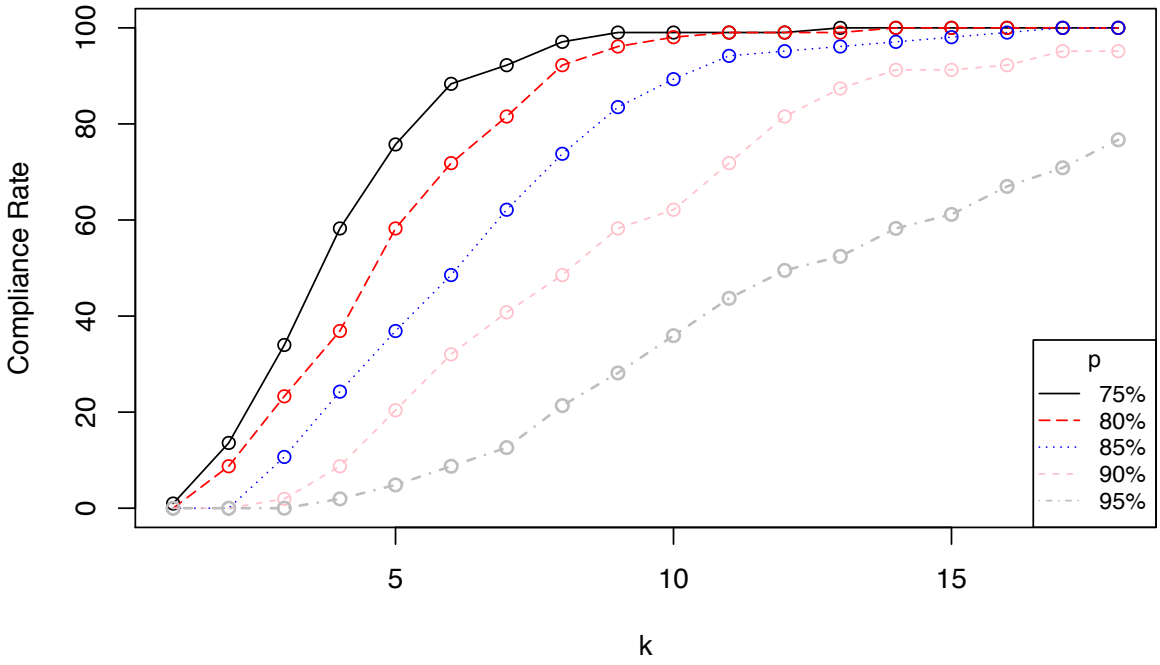


Figure 3.2. Compliance Rate Function (NOA metric)

As can be observed in Figure 3.3, *ComplianceRatePenalty* returns zero for the following pairs $[p, k]$: [75,7], [75,8], [75,9], [80,8], [80,9]. Based on our tiebreaker criteria, we select the result with the highest p and then the one with the lowest k, *i.e.*, [80, 8], which leads to the following relative threshold:

$$\boxed{80\% \text{ of the classes should have } NOA \leq 8}$$

This threshold represents a balance between the two forces the method aims to handle. First, it reflects a *real design rule*, followed by most systems in the considered corpus (in fact, it is followed by 102 out of 106 systems). Second, it is not based on rather lenient upper bounds. In other words, limiting NOA to eight attributes is compatible with an *idealized design rule*. For example, there are thresholds proposed by experts that recommend an upper limit of 10 attributes [18].

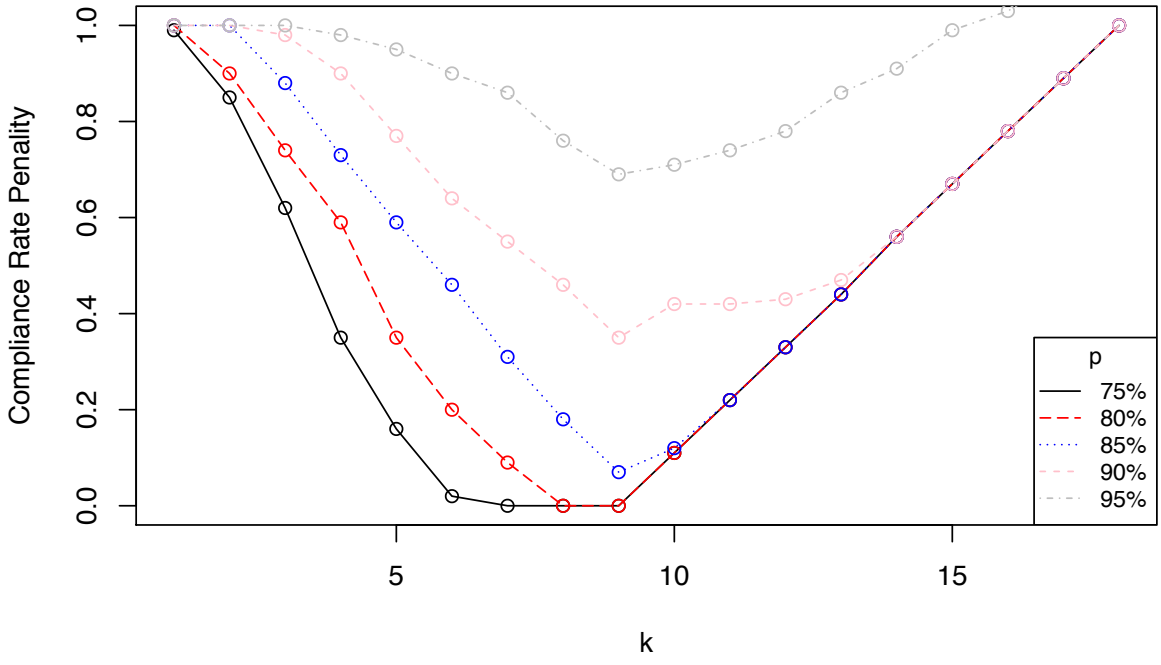


Figure 3.3. Compliance Rate Penalty Function (NOA metric)

To illustrate the classes that do not follow the proposed relative threshold, Table 3.1 presents the top-10 classes with the highest number of attributes in our *Corpus* (considering the 102 systems that follow the proposed threshold and only the largest class of each system). We manually checked the source code these classes and observed that classes with high NOA values are usually Data Classes [39], used to store global constants, like error messages in the *AspectJ* compiler or bytecode opcodes in the *Jasml* disassembler.

Table 3.1. Classes with highest NOA values

System	Class	NOA
GeoTools	gml3.GML	907
JasperReports	engine.xml.JRXmlConstants	600
Xalan	templates.Constants	334
Derby	impl.drda.CodePoint	324
AspectJ	core.util.Messages	317
Jasml	classes.Constants	301
POI	ddf.EscherProperties	275
DrJava	ui.MainFrame	266
RSSOwl	internal.dialogs.Messages	225
MegaMek	ui.swing.RandomMapDialog	216

In the Qualitas Corpus there are four systems (3.8%) that do not follow the relative threshold, which are *HSQLDB*, *IText*, *JMoney*, and *JTOpen*. For example, we manually checked that in the *JMoney* system 39.3% of the classes have more than 8 attributes. In this system, except for a single class, all other classes with NOA > 8 are related to GUI concerns: *e.g.*, the `AccountEntriesPanel` class has 37 attributes, including 25 attributes with types provided by the Swing framework. Another non-compliant system is *JTOpen*, a middleware for accessing applications running in IBM AS/400 hardware platforms. In this case, we counted 414 classes (25.2%) with NOA > 8, which are classes that implement the communication protocol with the AS/400 operating system. Therefore, the noncompliant behavior is probably due to the complexity of *JTOpen's* domain.

3.3 Method Properties and Characteristics

In this section, we discuss some properties and characteristics of the proposed method to derive relative thresholds. Specifically, we analyze the adherence of our method to requirement originally proposed to assess metric aggregation techniques, the robustness of the proposed method to staircase effects, its tolerance to bad smells, and some statistical properties.

3.3.1 Adherence to Requirement of Metric Aggregation Techniques

Mordal *et al.* defined a set of requirements to assess software metrics aggregation techniques [74]. We reused these categories to discuss our method mainly because metric

aggregation and metric thresholds ultimately share the same goal, *i.e.*, to support quality assessment at the level of systems. In the following discussion, we consider the two most important categories in this characterization (*must* and *should* requirements).

Must Requirements:

- *Aggregation*: Relative thresholds can be used to aggregate low level metric values (typically in the level of classes) and therefore to evaluate the quality of an entire project.

Should Requirements:

- *Highlight problems:* By their very nature, relative thresholds can indicate design problems under accumulation in the classes of object-oriented systems.
- *Do not hide progress:* The motivation behind this requirement is to reveal typical problems when using aggregation by averaging. On one hand, averages may fail due to a tendency to hide noncompliant systems. On the other hand, we argue that our method automatically highlights the presence of noncompliant systems above an expected value.
- *Decomposability:* Given a partition of the system under evaluation, it is straightforward to select the partitions that concentrate more classes not respecting the proposed thresholds. Possible partition criteria include package hierarchy, programming language, maintainers, etc.
- *Aggregation Range:* This requirement establishes that the aggregation should work in a continuous scale, preferably left and right-bounded. In fact, our relative thresholds can be viewed as predicates that are followed or not by a given system. Therefore, we do not strictly follow this requirement. We discuss the consequence of this fact in Section 3.3.2.
- *Symmetry:* Our final results do not depend on any specific order, *i.e.*, the classes can be evaluated in any order.

3.3.2 Staircase Effects

Staircase effects are a common drawback of aggregation techniques based on thresholds [74]. In our context, these effects denote the situation where small refactorings in

a class may imply in a change of threshold level, while more important ones do not elevate the class to a new category. To illustrate the scenario, suppose a system with n classes not following a given relative threshold. Suppose also that by refactoring a single class the system will start to follow the threshold. Although the scenarios before and after the refactoring are not very different regarding the global quality of the system, after the refactoring the system's status changes, according to the proposed threshold. Furthermore, when deciding which class to refactor, it is possible that a maintainer just selects the class more closer to the upper parameter of the relative threshold (*i.e.*, the "easiest" class to refactor).

Although subjected to staircase effects, we argue that any evaluation based on metrics—including the ones considering continuous scales—are to some extent subjected to quality treatments. In fact, treating values is a common pitfall when using metrics, which can only be avoided by making developers aware of the goals motivating their adoption [15].

3.3.3 Tolerance to Bad Smells

Because the thresholds tolerate a percentage of classes with high metric values, it is possible that they in fact represent bad smells, like God Class, Data Class, etc. [39]. However, when limited to a small number of classes—as required by our relative thresholds—our claim is that bad smells do not constitute a threat to the quality of the entire project nor an indication of an excessive technical debt. Stated otherwise, our goal is to raise quality alerts when bad smells change their status towards a disseminated and recurring design practice.

3.3.4 Statistical Properties

In the method to extract relative thresholds, the median of a high percentile is used to penalize upper limits that do not reflect the accepted semantics for a given metric values. We acknowledge that the use of the median in this case is not strictly recommended, because we never checked whether the 90-th percentiles follow a normal distribution. However, our intention was not to compute an expected value for the statistical distribution, but simply to penalize compliance rates based on lenient upper limits, *i.e.*, limits that are not observed at least in half of the systems in our corpus.

3.4 *RTTool*

In this section, we describe *RTTool*, a tool supporting the proposed method to extract relative thresholds [79]. *RTTool* can be used to help making decision in different way. For example, a software quality manager can run the tool in its portfolio of systems. The idea is to obtain its thresholds according with its development patterns, context, team, and others. After, the manager can use these thresholds as pattern for its new projects. Moreover, he also is able to identify systems with poor internal quality and to help in systems re-engineering.

The proposed tool has the following features:

- *RTTool* is applicable to any software metric as long as low(er) metric values are considered to be more desirable than the high(er) ones, and the metrics distribution is heavy-tailed. However, *RTTool* does not check whether the metric distribution is heavy-tailed.

- *RTTool* is flexible and independent of software metric collection tool.[2] Indeed, importance of differences between tools calculating "the same metrics" has been observed in the past [65]: *RTTool* does not take a stance in this debate.

- *RTTool* can be configured for different contexts, *e.g.*, system size or application domain, since context is known to be crucial when deriving metrics thresholds [5].

- *RTTool* indicates the systems that do not follow the relative thresholds for a given metric, which we called noncompliant systems.

- *RTTool* generates several partial results for user analyses, for example, the user can to view the plot of the Cumulative Density Function (CDF) or the CDF inverse, the Quantile function, to examine a distribution of values of a metric.

- *RTTool* includes graphs to visualize the results. These graphs includes different perspectives of the results.

The execution of the *RTTool* is divided into three stages: configuration, processing, and presentation (Figure 3.4). In the *configuration* stage, the user selects the dataset, with the metric values collected for a given *Corpus*. The current version of *RTTool* accepts CSV or XML files with metrics values as input. The *processing* stage is responsible for deriving the p and k parameters of the relative threshold. In this

[2]Currently, there are several tools available for collecting software metrics. Therefore, collecting metrics is not the aim of our tool.

stage, *RTTool* also identifies the systems that do not respect relative thresholds. More specifically, this stage calculates functions showed in Figure 3.1. Finally, in the *presentation* stage, the results are shown as spreadsheets and graphs. The spreadsheets summarize the relative thresholds derived and the systems that do not respect the relative thresholds. The presentation stage also plots a number of graphs including *ComplianceRate* $[p, k]$, *ComplianceRatePenalty*$[p, k]$, and Quantile Function.

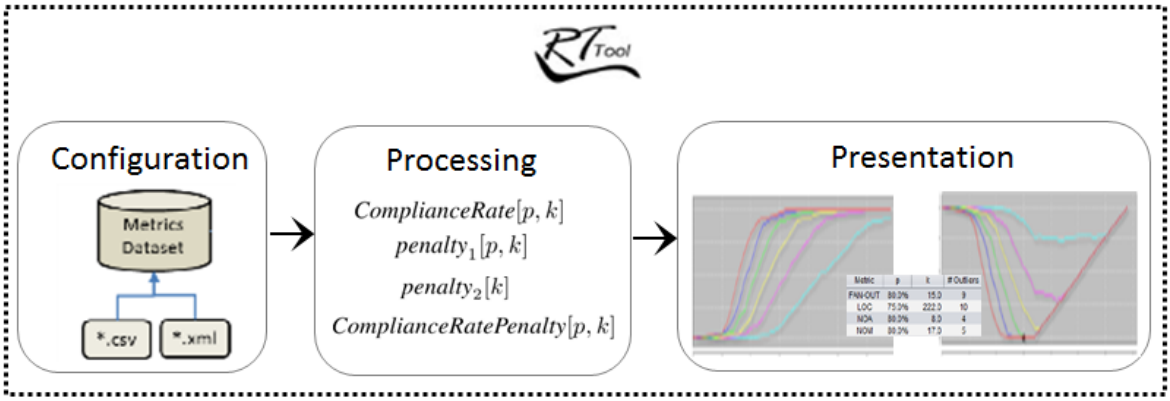


Figure 3.4. *RTTool* stages

3.4.1 Example of usage

In order to illustrate the usage of our tool, we derive relative thresholds for a number of metrics collected for the 106 systems in Qualitas Corpus (version 20101126r) [97]. We used the Moose platform [77] and VerveineJ[3] to compute the values of the metrics for each class of each system and store them as CSV files.

First, to use the *RTTool*, the user must select the metrics to extract relative thresholds. After uploading the CSV files generated by Moose, 20 metrics become available for analysis (Figure 3.5) and the user selects four of them: FAN-OUT, NOA, SLOC, and NOM.

Then, *RTTool* calculates the p and k values, that characterize relative thresholds for each metrics and shows the number and the names of the noncompliant systems (Figure 3.6). We can observe that the p values derived for different metrics are close suggesting that the k thresholds derived hold for 75%-80% of the systems. Moreover, we see that Weka, HSQLDB, and JTOpen appear as noncompliants for at least three metrics.

Finally, by inspecting Figures 3.7 and 3.8 we can see how *RTTool* has selected the relative thresholds. Indeed, by inspecting Figure 3.8 we observe that 80% is the

[3] `http://www.moosetechnology.org/tools/verveinej`, verified on 25/11/2014.

highest value of p such that there exists k satisfying $ComplianceRatePenalty[p, k] = 0$. This k equals 15 and is denoted with a small black circle on Figure 3.8. By consulting Figure 3.7 we observe that 90% of the *Corpus* systems follow the relative threshold $[80\%, 15]$ derived.

Using the slide bar on the right the user can select the p values she would like to inspect. The value on the slide bar indicates the lowest value to be visualized together with the curves obtained for p with increments of 5%. As expected, relaxing the relative threshold p value *e.g.*, to 70% results in a lower k equals 10 and in a comparable *ComplianceRate* of 82% (Figure 3.7).

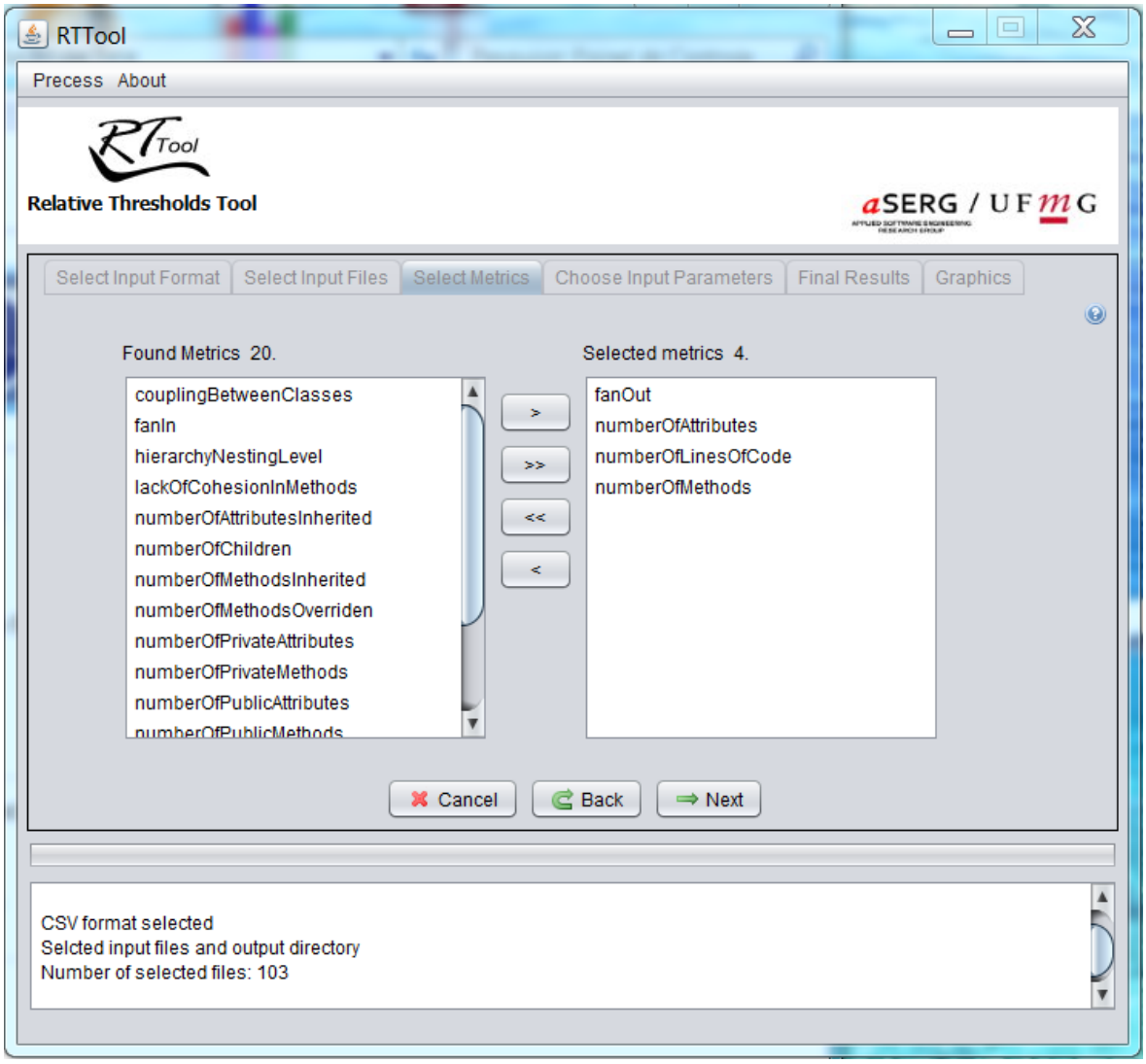


Figure 3.5. Configuration window

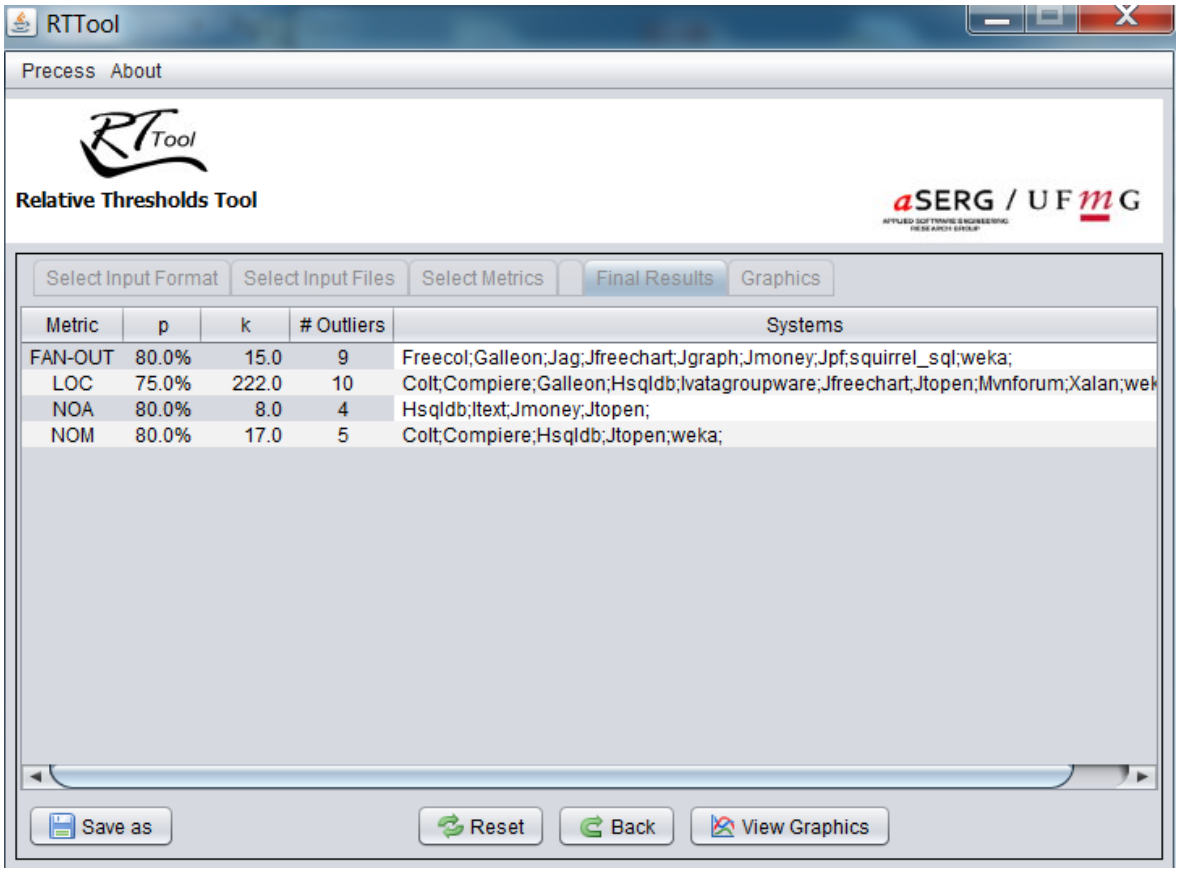


Figure 3.6. Final results — with thresholds and noncompliants systems for each metric

3.4.2 Performance

To evaluate the performance of *RTTool* we have measured the runtime in four experiments by varying the size of the corpus, *i.e.*, the entire Qualitas Corpus (106 systems) vs. Qualitas Corpus systems classified as Tools by the corpus curators (27 systems), and the number of metrics (four metrics selected as in the example above vs. all twenty metrics available in the dataset). Table 3.2 summarizes the runtime measurements as reported by *RTTool* itself. The experiments have been run on a device with core i5 processor and 4GB DDR3 memory.

As expected, increasing the number of metrics or the size of the corpus results in higher execution times. However, even for the largest corpus and the maximal number of metrics the calculation time remains acceptable, slightly exceeding one minute (75949 milliseconds).

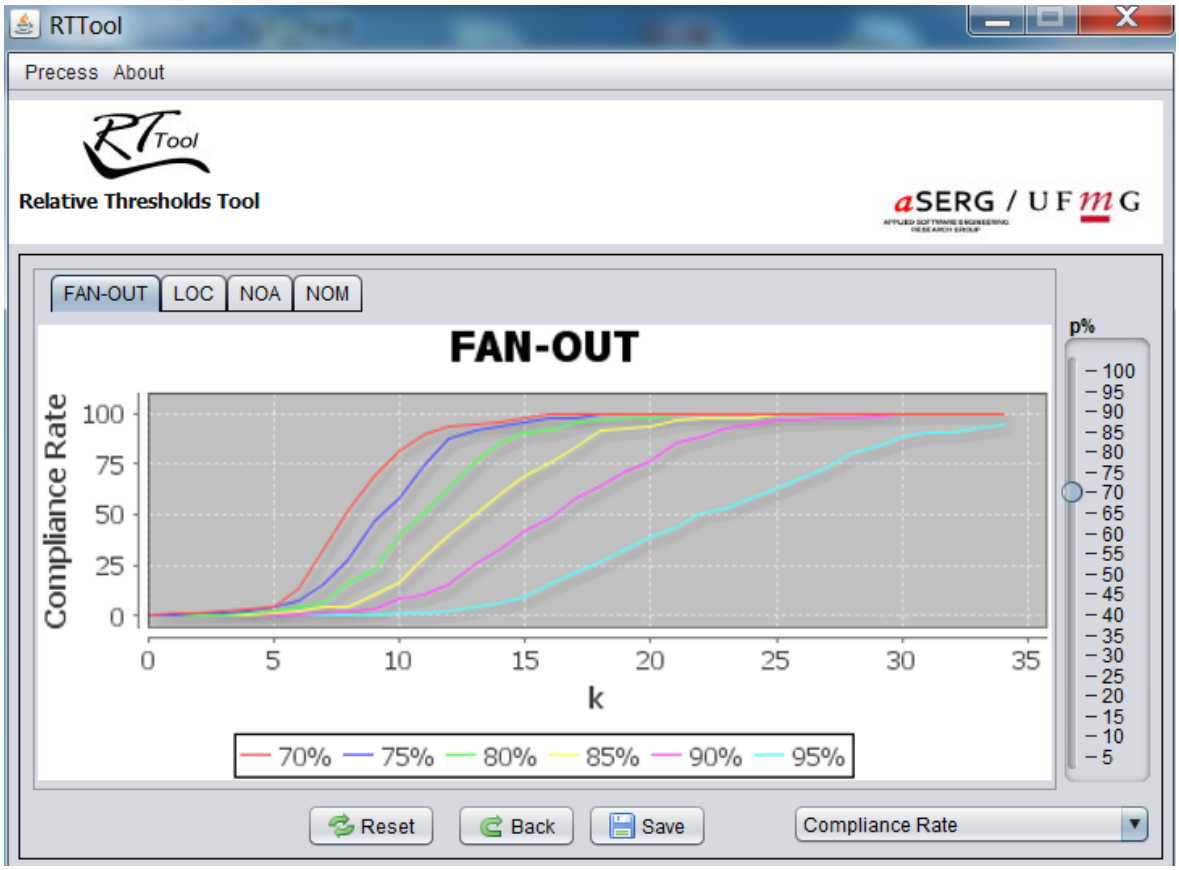


Figure 3.7. ComplianceRate function (FAN-OUT metric)

Table 3.2. Runtime of *RTTool*

Corpus	# systems	# metrics	time (ms)
Qualitas Corpus—Tools	27	4	13753
Qualitas Corpus—Tools	27	20	35056
Qualitas Corpus	106	4	15867
Qualitas Corpus	106	20	75949

3.4.3 Availability

RTTool is an open source project, distributed under the MIT license. We have opted for the MIT license since it permits reuse of the source code in the proprietary software: in this way we hope that the relative threshold calculation implemented in *RTTool* can find a way both to mainstream metrics calculation tools [19] and to research prototypes focusing on software analytics [99]. The proposed tool is available at `http://aserg.labsoft.dcc.ufmg.br/rttool`.

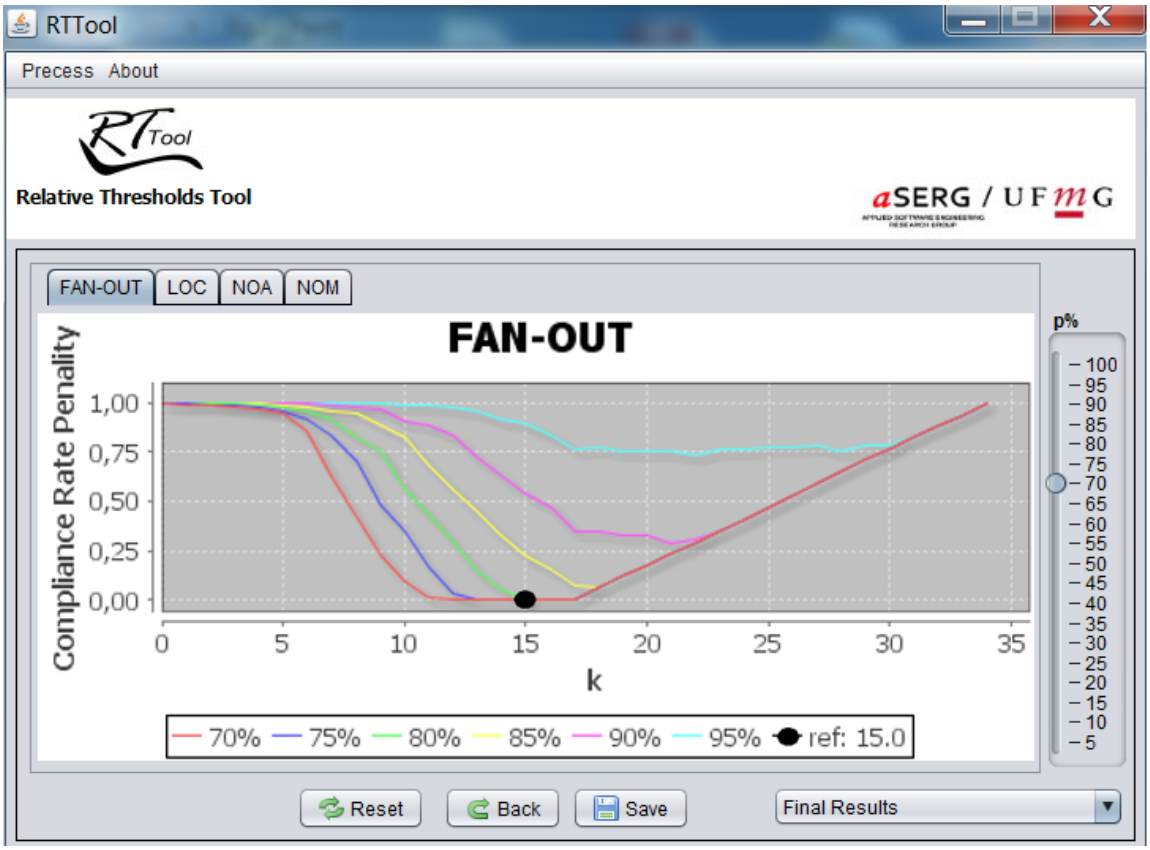


Figure 3.8. ComplianceRatePenalty function (FAN-OUT metric)

3.4.4 Related Tools

Extraction of thresholds based on system corpus has been studied in the literature [5, 35, 44, 93]. Unfortunately, most of these approaches are not supported by tools, the work of Alves, Ypma and Visser [5] being the only notable exception. The tool proposed in this work focuses on extracting absolute thresholds, weights the metrics based on SLOC of the corresponding entities and constructs four quality profiles corresponding to low risk (0 to 70th percentiles), moderate risk (70th to 80th percentiles), high risk (80th to 90th percentiles), and very-high risk (90th percentile). As opposed to this line of work, *RTTool* derives relative thresholds, does not perform weighting and considers only two "quality profiles" (adhering to the relative thresholds or not). Finally, the tool of Alves, Ypma and Visser is proprietary, while *RTTool* is open source.

3.5 Final Remarks

In this chapter, we proposed the notion of relative thresholds to deal with such metric distributions. Our approach explicitly indicates that thresholds should be valid for most

but not for all classes in object-oriented systems. We proposed a method that extracts relative thresholds from a *Corpus*. Next, we described an illustrative example of the our method, which we derive a threshold for the Number of Attributes (NOA) metric, based on 106 systems of the Qualitas Corpus. Then, we discussed some properties and characteristics of the proposed method to derive relative thresholds. Finally, we describe *RTTool*, an open source tool capable of extracting relative thresholds for software metrics based on benchmark collections.

Chapter 4

Relative Thresholds for the Qualitas Corpus

In this chapter, we derive relative thresholds for six source code metrics, using the Qualitas Corpus (version 20101126r). Next, we report an extensive study, which include: Section 4.4 investigates whether popular open source Java repositories, available at GitHub, follow the relative thresholds; Section 4.5 compares our results with thresholds extracted using a method proposed by the Software Improvement Group (SIG method), which also determines metric thresholds empirically from measurement data [5]; Section 4.6 evaluates the influence of context in our results; Section 4.7 checks how the proposed thresholds apply to different versions of the systems under analysis; Section 4.8 investigates the importance of classes that do not follow the upper limit of a relative threshold, by checking how often such classes are changed; Section 4.9 investigates the relation between the presence of bad smells in a system and its adherence to the proposed relative thresholds; Section 4.10 evaluates the dispersion of the metric values in the systems respecting the proposed thresholds, using the Gini coefficient; Section 4.11 discusses possible threats to validity; and Section 4.12 presents the final remarks.

4.1 Corpus and Metrics

In order to derive relative thresholds, we use a Qualitas Corpus (version 20101126r). This *Corpus* is a curated dataset with 106 open source Java-based systems, specially created for empirical research in software engineering [97]. Figure 4.1 describes the size of the systems in our corpus in terms of classes.

For this study, we used source code metrics related to distinct factors affecting

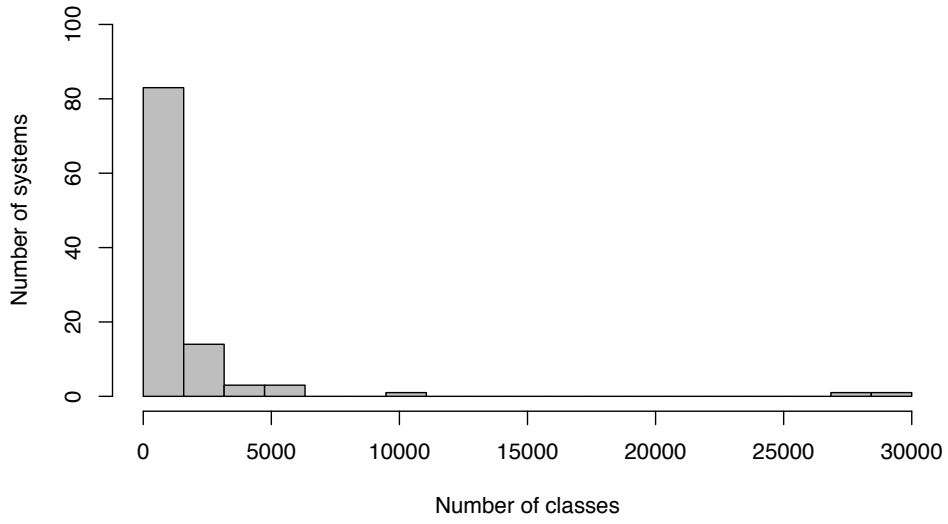


Figure 4.1. Size of the systems in the our *Corpus*

the internal quality of object-oriented systems, such as size, coupling, complexity, and cohesion. To compute the values of the selected metrics for each class of each system we use the Moose platform [77]. Particularly, we relied on VerveineJ[1]—a Moose application—to generate MSE files, a Moose specific format for representing source code models. Then, we use Moose to generate CSV files from MSE files. The following metrics have been selected:

- *Number of methods (NOM)* [67]: NOM is an indicator of the size of a class. Moose computes this metric by counting all methods in the class, including constructors, getters, and setters.

- *Number of Lines of Code (SLOC)* [43]: SLOC is also an indicator of the size of a class. Moose computes this metric by counting all lines of code with the exception of comments.

- *Number of Provider Classes (FAN-OUT)* [77]: FAN-OUT is a coupling metric that counts the number of other classes referenced by a class. Moose computes this metric by considering all types of class dependencies (due to inheritance, method calls, static accesses).

- *Response For a Class (RFC)* [22]: RFC is computed by Moose as the sum of the NOM metric value and the number of methods invoked by each method of the class.

[1]http://www.moosetechnology.org/tools/verveinej, verified 11/25/2014.

- *Weighted Method Count (WMC)* [22]: WMC is the sum of the cyclomatic complexities of each method in a class.

- *Lack of Cohesion in Methods (LCOM)* [21]: This metric measures the cohesion level of a class by evaluating the similarity of its methods. Moose computes this metric by counting the number of disjoint sets of methods, according to Chidamber and Kemerer, 1991.

4.2 Study Setup

Although the literature reports that object-oriented metrics usually follow heavy-tailed distributions [13, 96], we checked ourselves whether the metric values we extracted present this behavior. For this purpose, we used the EasyFit tool[2] to reveal the distribution that best describes our values. We configured EasyFit to rely on the Kolmogorov-Smirnov Test to compare our metrics data against reference probability distributions. Following a classification suggested by Foss et. al [37], we considered the metric distributions extracted for a given classes as heavy-tailed when the "best-fit" distribution returned by EasyFit is Power Law, Weibull, Lognormal, Cauchy, Pareto, or Exponential. Table 4.1 reports the number of systems whose metric values are classified as heavy-tailed. The results show that the extracted values follow heavy-tailed distributions in at least 100 systems (94.3%). The presence of a small number of non-heavy-tailed distributions does not invalidate our results, since they are based on the median of the 90-th percentiles. This median value is more robust the presence of other distributions, *e.g.*, a distribution with small values in the last percentiles.

Figure 4.2 shows the quantile functions for the considered metric values. In this figure, the x-axis represents the quantiles and the y-axis represents the upper metric values for the classes in a given quantile. The figure visually shows that the extracted metric values follow heavy-tailed distributions, with most systems having classes with very high metric values in the last quantiles. There are also systems with a noncompliant behavior, due to the presence of high-metrics values even in intermediary quantiles (*e.g.*, 50th or 60th quantiles).

4.3 Results

Table 4.2 presents the relative thresholds derived by our method, considering all systems of the *Corpus*. For each metric, the table shows the values of p and k that char-

[2] `http://www.mathwave.com/products/easyfit.html`, verified 11/25/2014.

Table 4.1. Number and percentage of systems with heavy-tailed metric values distributions

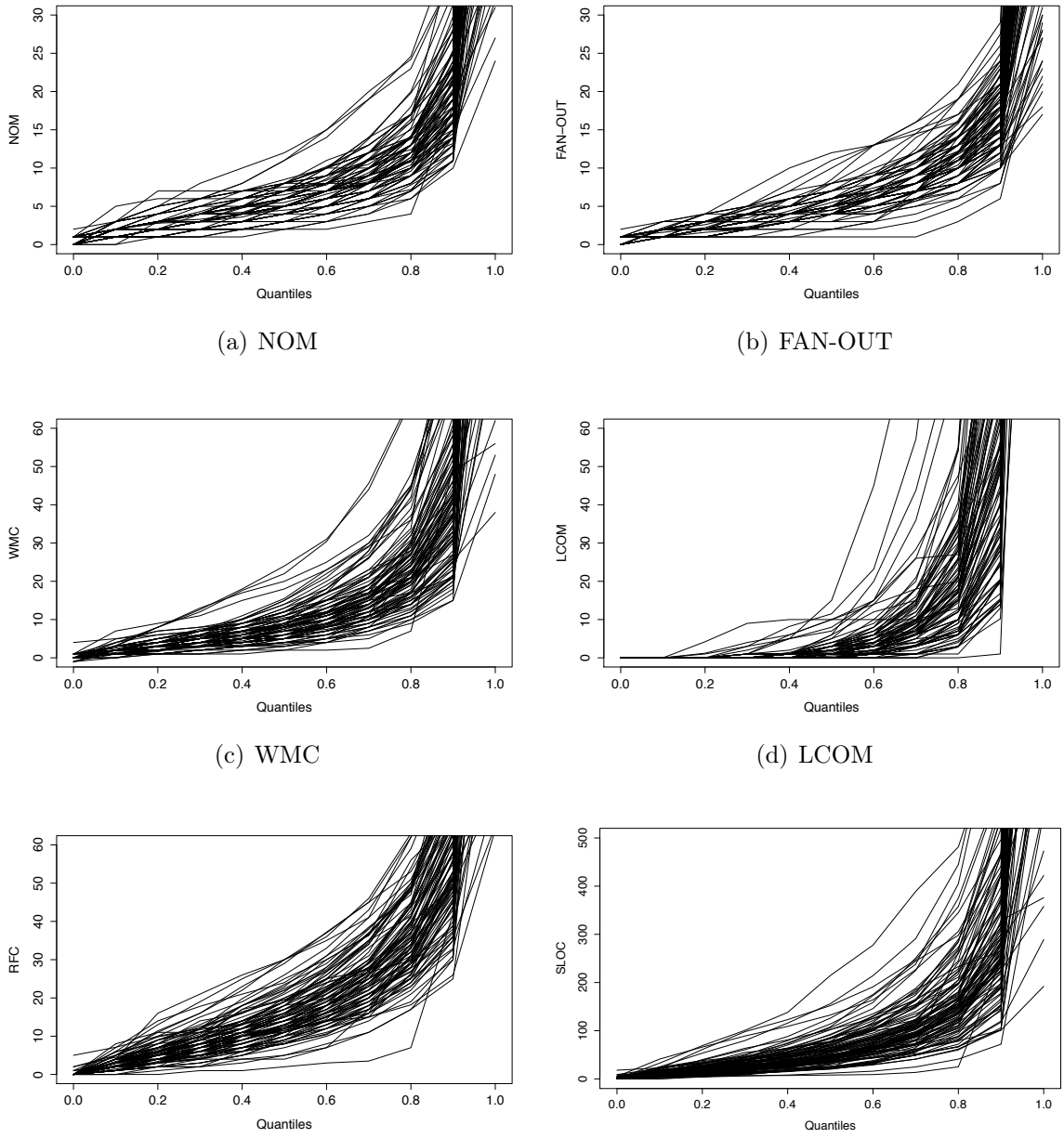
Metrics	# Systems	% Systems
NOM	100	94.3
SLOC	102	96.2
FAN-OUT	105	99.0
RFC	105	99.0
WMC	106	100.0
LCOM	100	94.3

acterize the relative thresholds. For example, for NOM the proposed relative threshold is "**80%** of the **classes** should have **NOM** $\leq$ **16**". The table also shows the number and the names of the systems violating these thresholds. We call these as systems *noncompliant*. Finally, Table 4.2 presents thresholds found in the literature. On the one hand, we can observe that our k parameters are usually lower than such thresholds. On the other hand, the proposed relative thresholds tolerate a percentage of classes in each system that do not respect the upper limit, while traditional thresholds assume that all classes adhere to them.

In Section 3.1, we suggest that relative thresholds should represent a commitment between *real* and *idealized* design rules. In fact, the number of systems with a noncompliant behavior ranges from five systems (NOM) to twelve systems (LCOM), *i.e.*, from 4.7% to 11.3% of the systems in the Qualitas Corpus (*real design rules*). The proposed thresholds seem also to represent *idealized* design rules, as can be observed by the values of the upper limit k. For example, well-known Java code standards recommend that classes should have no more than 20 methods [18, 44] and our method suggests an upper limit of 16 methods. This balance between *real* and *idealized* design rules is achieved by accepting that the thresholds are valid for a representative number of classes, but not for all classes in a system. In fact, the suggested upper limits apply to a percentage p of classes ranging from 75% (SLOC) to 80% (NOM, FAN-OUT, RFC, WMC, and LCOM).

4.4 Application on Popular GitHub Repositories

In this section, we explore whether popular open source Java repositories, available at GitHub, follow the proposed relative thresholds. For this analysis, we consider a repository as popular if it has at least 1,000 GitHub stars (starring is a GitHub feature that lets users show their interest on repositories). We assume that popular systems



(a) NOM (b) FAN-OUT

(c) WMC (d) LCOM

(e) RFC (f) SLOC

Figure 4.2. Quantile functions

have a good design and therefore most of them should follow the proposed thresholds.

Table 4.2. Relative Thresholds

Metrics	p	k	Noncompliant Systems	Literature
NOM	80	16	Colt, Compiere, HSQLD, JTOpen, Weka (5 systems)	20 methods [18, 44]
SLOC	75	222	Colt, Compiere, Derby, Galleon, HSQLD, Ivatagroupware, JFreeChart, JTOpen, Weka, Xalan, Xerces (11 systems)	500 lines [44]
FAN-OUT	80	15	Freecol, Galleon, JAG, JFreeChart, JGraph, JMoney, JPF, SQuirrelSQL, Weka (9 systems)	-
RFC	80	49	AspectJ, Compiere, Freecol, Galleon, HSQLDB, JAG, JFreeChart, JGraph, JMoney, SQuirrelSQL, Weka (11 systems)	50 methods [88], 27 methods [14], 100 methods [44], 44 methods [93]
WMC	80	32	AspectJ, Colt, Compiere, Derby, HSQLD, IText, JTOpen, mvnForum, Weka, Xerces (10 systems)	25 [88], 100 [44], 11 [14], 24 [93]
LCOM	80	36	ANTLR, AspectJ, Axion, Colt, Compiere, HSQLDB, Informa, IText, JFreeChart, JTOpen, Xerces, Weka (12 systems)	-

4.4.1 Study Setup

We selected all repositories ranked with at least 1,000 stars at GitHub, whose sole language is Java[3]. This search was performed on July, 2015 and resulted in 308 repositories. We used again Moose software analysis platform to compute source code metrics[4]. Considering all systems, the dataset includes more than 531K files, 61 MLOC, and 355K commits. After checking out the most recent version of each repository, we automatically inspected their source code to remove test classes. These classes were removed because they usually have a structure very different from functional code [55]. The number of stars of the selected systems range from 11,869 (*Elasticsearch*) to 1,005 (*LDrawer*). Table 4.3 shows the top-10 repositories selected for this study, including a brief description and their number of stars. After downloading the repositories, we evaluate their percentage of classes (p parameter) respecting the proposed upper limit (k parameter) of the relative thresholds reported in Table 4.2.

[3]We use the query `language=Java and stars>1,000`.

[4] `http://www.moosetechnology.org`

Table 4.3. Top-10 popular GitHub Java repositories (ordered by # stars)

Systems	Description	# Stars
ElasticSearch	Search engine built for the cloud	11,869
Universal Image Loader	Images Android library	9,211
Storm	Distributed realtime computation system	8,638
SlidingMenu	Slide-in menus Android library	7,886
ActionBarSherlock	Action bar design pattern Android library	6,766
Google I/O Android App	Android app for the conference	6,488
GitHub Android App	Source code for the GitHub Android	6,190
LibGDX	Java game development framework	6,162
Asynchronous Http Client	Http client for Android	5,981
Picasso	Image library for Android	5,734

4.4.2 Results

Table 4.4 summarizes the results of this evaluation. This table shows the percentage of repositories that follow the proposed relative thresholds for each metric. We can observe that more than 90% of the repositories follow our thresholds, in all cases. FAN-OUT is the metric with the highest percentage of repositories following its threshold (99%) and NOM is the metric with the lowest percentage (93%). Moreover, Table 4.5 details the results of this evaluation for the top-10 repositories. This table shows the percentage of classes in each repository that follow the proposed relative thresholds. For instance, the relative threshold for NOM is $[80, 16]$ and we can observe that 95% of the classes in *Storm* have 16 methods or less, *i.e.*, *Storm* respects the relative threshold for NOM. We can also observe that only *libGDX* does not follow the relative threshold proposed to NOM.

Table 4.4. Repositories that follow the proposed relative thresholds

Systems	# Repositories	% Repositories
NOM	287	93
SLOC	297	96
FAN-OUT	305	99
RFC	289	94
WMC	290	94
LCOM	292	95

Finally, we analyzed the main noncompliant repositories, *i.e.*, repositories that do not follow the proposed thresholds for at least three metrics, as presented in Table 4.6. We found 14 noncompliant repositories (4.6%). Only one repository

Table 4.5. Percentage of classes in the top-10 popular Java repositories that respect the upper limit k of a relative threshold (the underlined value is the only case when a threshold is not respected).

Repositories	**NOM** [80, 16]	**SLOC** [75, 222]	**FAN-OUT** [80, 15]	**RFC** [80, 49]	**WMC** [80, 32]	**LCOM** [80, 36]
ElasticSearch	93%	96%	92%	93%	94%	92%
Storm	95%	97%	94%	95%	96%	94%
Universal Image Loader	94%	95%	98%	97%	94%	94%
SlidingMenu	82%	100%	100%	100%	100%	100%
ActionBarSherlock	90%	94%	99%	93%	92%	91%
Google I/O Android App	96%	94%	98%	91%	94%	97%
GitHub Android App	97%	98%	100%	94%	98%	99%
LibGDX	<u>74%</u>	90%	96%	85%	83%	82%
Asynchronous Http Client	90%	93%	93%	90%	90%	93%
Picasso	97%	97%	97%	97%	97%	97%

violates all metrics (0.3%), five repositories violate five metrics (1.6%), six repositories violate four metrics (2.0%), and two repositories violate three metrics (0.7%). NOM has the highest number of violations (14 repositories) and FAN-OUT has the lowest one (three repositories). We manually analyzed the noncompliant repositories and found that they indeed have evidences of presenting a different structure than other systems. First, 11 out of 14 noncompliant repositories are Android applications and they have few classes (< 50), which typically have many methods and tend to represent God Classes [39]. To illustrate, *Smooth Progress Bar*, *Disk LRU Cache*, *ListView MaterialEditText*, and *ContextMenu* have four, nine, two, eight, and three classes, respectively. Moreover, we found the following notes in the documentation of *HTTP-Request* and *Pull To Refresh for Android*:

"The goal of this library (HTTP-Request) is to be a single class class with some inner static classes."

"This library (Pull To Refresh for Android) is deprecated, a swipe refresh layout is available in the v4 support library."

HTTP-Request is a library for using a *HttpURLConnection* to make requests, and it violates the thresholds for all metrics. *Pull To Refresh for Android* is an application

that provides a reusable pull to refresh Android widgets. The application violates the thresholds for four metrics.

Table 4.6. Noncompliant repositories for at least three metrics

Repositories	Metrics					
	NOM	SLOC	FAN-OUT	RFC	WMC	LCOM
HTTP-Request	✓	✓	✓	✓	✓	✓
Smooth Progress Bar	✓	✓		✓	✓	✓
Disk LRU Cache	✓		✓	✓	✓	✓
Joda-Time	✓	✓		✓	✓	✓
Processing	✓	✓		✓	✓	✓
MySQL Performance Analyzer	✓		✓	✓	✓	✓
ListView	✓	✓		✓	✓	
FQRouter	✓	✓			✓	✓
Pull To Refresh for Android	✓	✓		✓	✓	
MaterialEditText	✓			✓	✓	✓
Material	✓	✓		✓	✓	
OkHttp	✓			✓	✓	✓
Okio	✓				✓	✓
ContextMenu	✓			✓		✓

Summary of findings: We conclude that most popular GitHub repositories follow the proposed relative thresholds. Regarding the main noncompliant repositories, they are usually Android applications, with few classes that tend to follow a God Class structure.

4.5 Comparison with SIG Method

This section compares our results with thresholds extracted using a method proposed by the Software Improvement Group (SIG method), which also determines metric thresholds empirically from measurement data [5]. The thresholds extracted by this method are used as input to a maintainability assessment model [8, 42]. We compare our method with SIG because it is being used in industry to asses software quality for more than five years.

In the SIG method, metric values for a given program entity are first weighted according to the size of the entities in terms of lines of code (SLOC). After this step, quality profiles are used to rank classes according to four categories: low risk (0 to 70th percentiles), moderate risk (70th to 80th percentiles), high risk (80th to 90th percentiles), and very-high risk (> 90th percentile). In order to compare our results with thresholds extracted using SIG method, we decide to implement ourselves SIG

algorithm.[5] We use this implementation to extract thresholds for the same metrics and for the same systems used in Section 4.1, with exception of SLOC, since the SIG method uses SLOC when deriving thresholds for other metrics.

4.5.1 Results

Table 4.7 shows the thresholds derived by both methods. The NOM thresholds derived using the SIG method consist in the following profiles: classes up to 29 methods are characterized as with low risk, from 29 to 42 methods are characterized as with moderate risk, from 42 to 77 methods are characterized as with high risk, and classes having more than 77 methods are characterized as with very-high risk. Using the method proposed in this paper, the relative threshold derived for NOM is "80% of the classes should have NOM $\leq$ 16".

Table 4.7. Relative vs SIG thresholds

Metrics	**SIG Risk Profile**				**Relative Thresholds**	
	Low	Moderate	High	Very-High	p	k
NOM	29	42	77	$>$ 77	80	16
FAN-OUT	22	29	46	$>$ 46	80	15
RFC	88	127	224	$>$ 224	80	49
WMC	80	136	268	$>$ 268	80	32
LCOM	180	361	3,654	$>$ 3,654	80	36

In Table 4.7, we can observe that for all metrics the upper limit of a relative threshold (k parameter) is lower than the upper limit of the low risk classes in the SIG method. However, our method accepts that some classes exceed these upper limits (p parameter), *e.g.*, a system may have up to 20% of classes with more than 16 methods. Moreover, in our method there are two penalties to optimize p and k parameters. Thus, these penalties punishes systems that are somehow "atypical" in the *Corpus*.

Furthermore, we also investigate the percentage of high and very-high risk classes for each system in the *Corpus*, as represented in Figure 4.3. In this figure, the x-axis represents the 106 systems in our *Corpus* and the y-axis represents the percentage of high and very-high risk classes in each system. The noncompliant systems detected by our method are represented by black bars, while the remaining systems are represented by white bars. First, we observe that for all systems and metrics the percentage of classes characterized as high and very-high risk classes is low ($<$ 11%). Second, the noncompliant systems—according to our method— usually

[5]SIG does not provide an open source tool to derive their thresholds

have the highest percentage of high and very-high risk classes, *i.e.*, the black bars are usually the ones with the highest percentage of high and very-righ risk classes. This result shows that noncompliant systems—detected by our method— are ranked among the systems with the highest percentage of problematic classes, as detected by the SIG method.

Summary: We conclude that both methods convey similar information as showed in the case of noncompliant systems. However, the goal of SIG method is to rank entities according to four quality profiles. Thus, it does not indicate noncompliant systems automatically and it does not indicate the percentage of classes that we should tolerate in each risk profile. By contrast, our method derives relative thresholds that can be automatically used to detect noncompliant systems. Moreover, we extract relative thresholds that by construction tolerate high-risk classes, assuming they are natural in heavy-tailed distributions. Nevertheless, these classes should not exceed a percentage of the whole population of classes.

4.6 Contextual Analysis

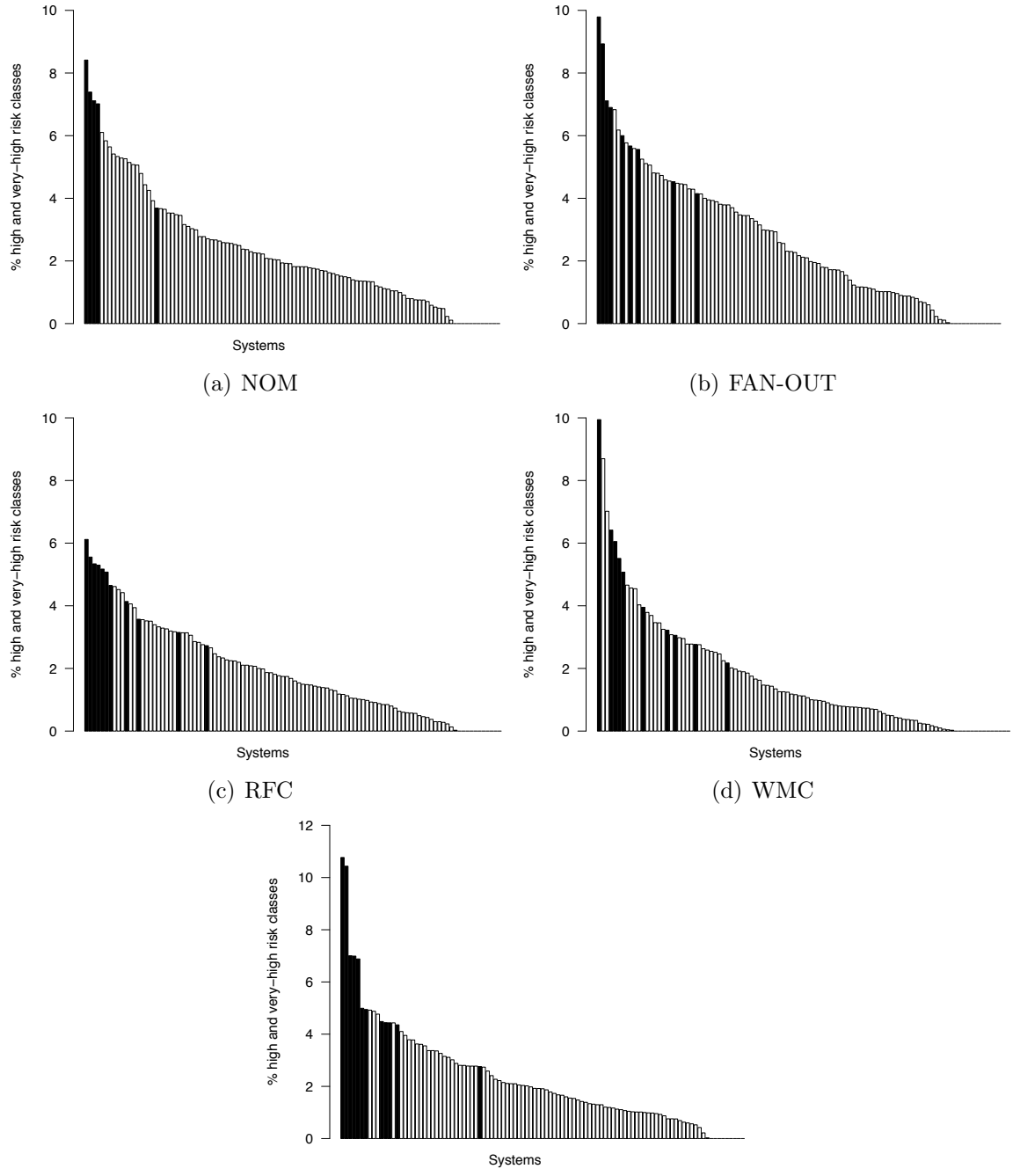
Several works highlight the importance of contextual factors, such as application domain, programming language, and size, when analyzing source code metrics [31, 35, 38, 109]. Therefore, to evaluate the influence of context in our results, specifically system's size and system's domain, we conduct a study to address the following research questions:

RQ #1 — What is the impact of context changes in the extracted relative thresholds? With this research question we aim to investigate how changes in context affect the parameters p and k, from relative thresholds.

RQ #2 — Do systems change their noncompliant status when context change? With this research question we aim to investigate how the context affects the systems classified as noncompliant.

4.6.1 Study Setup

To provide answers to these research questions, we recalculate the relative thresholds for three subsets of the Qualitas Corpus representing different application domain: *Tools*, *Middleware*, and *Testing*. The application domains are labeled by the curators



(a) NOM (b) FAN-OUT

(c) RFC (d) WMC

(e) LCOM

Figure 4.3. Percentage of high and very-high risk classes for each system in the Qualitas Corpus. Black bars represent noncompliant systems.

of the Qualitas Corpus; indeed, *Tools*, *Middleware*, and *Testing* are the three largest domains in Qualitas. Table 4.8 shows the number and the percentage of systems in each application domain subcorpus.

We also recalculate the relative thresholds for three subsets of systems with different

Table 4.8. Subcorpus by Application Domain

Subcorpus	# Systems	% Systems
Tools	26	26%
Middleware	17	16%
Testing	12	11%

size: up to 300 classes, from 301 to 1,000 classes, and with more than 1,000 classes. These categories are used because they generate subsets with similar number of systems, as in Table 4.9.

Table 4.9. Subcorpus by size

Subcorpus	# Systems	% Systems
≤ 300	37	35%
301 to $1,000$	36	34%
$> 1,000$	33	31%

4.6.2 Results

RQ #1 — What is the impact of context changes in the extracted relative thresholds?

Figure 4.4 shows the percentage of changes in the values of p (black bar chart) and k (gray bar chart) parameters, when comparing the thresholds obtained in the whole Qualitas Corpus with the thresholds derived in the proposed subcorpus. First, by observing the results in this figure, we conclude that variations in p are small. They are observed only for three metrics: SLOC, FAN-OUT, and WMC, and are generally less than 10%. In contrast, the variations in the k parameter are more common, but usually restricted to at most 20%. In fact, changes in this parameter ($> 20\%$) happen when restricting the analysis to two subcorpora: *Tools* (for FAN-OUT and WMC) and *Testing* (for SLOC and LCOM). In *Tools*, for example, there is an increase of more than 30% in k when deriving a threshold for FAN-OUT. In *Testing*, there is a decrement of more than 20% in k when extracting SLOC thresholds. In other words, systems in *Tools* tend to present higher coupling measures, than systems in the whole *Corpus*. Moreover, *Testing* classes tend to be smaller than the ones in the whole *Corpus*.

RQ #2 — Do systems change their noncompliant status when context change?

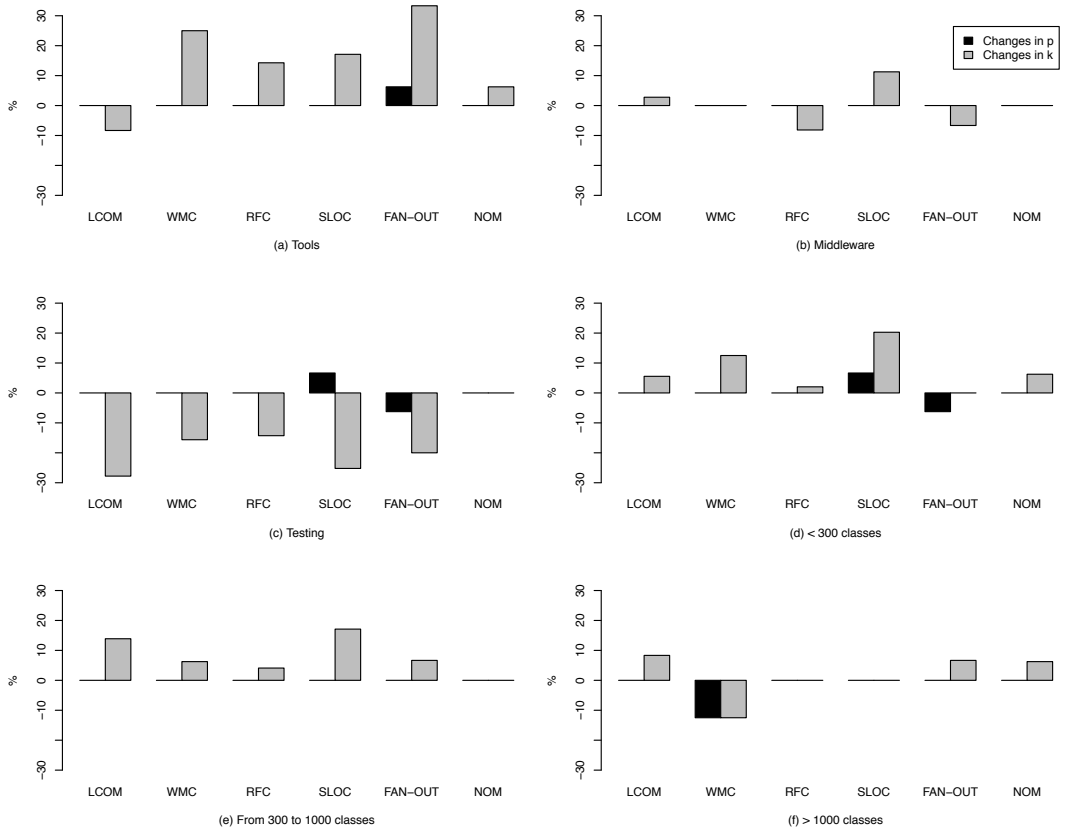


Figure 4.4. Contextual analysis

To answer this RQ, we analyze three categories of systems: (i) *New Noncompliant*—systems that follow the thresholds in the whole *Corpus*, but turned to be noncompliant when analyzing a restricted subcorpus; (ii) *Still Noncompliant*—systems that are noncompliant in the whole *Corpus* and that keep this status in the subcorpus; and (iii) *No Longer Noncompliant*—systems that are noncompliant in the whole *Corpus*, but that are no longer classified as such when analyzing the subcorpus.

Figure 4.5 shows the results for this question. With the exception of *Testing*, the presence of *New Noncompliant* (represented by white bars) is rare. Particularly, *Testing* has three *New Noncompliant*, which are *Cobertura* (NOM, SLOC and WMC), *JMeter* (FAN-OUT and RFC), and *HTMLUnit* (LCOM). Furthermore, most considered systems are classified as *Still Noncompliant* (represented by black bars). However, the number of *No Longer Noncompliant* is also considerable (represented by gray bars), which shows that the proposed method is able to reclassify the systems. Therefore, when moving from a general to a more homogeneous *Corpus* some systems are reclassified, but predominantly changing their status from noncompliant to compliant.

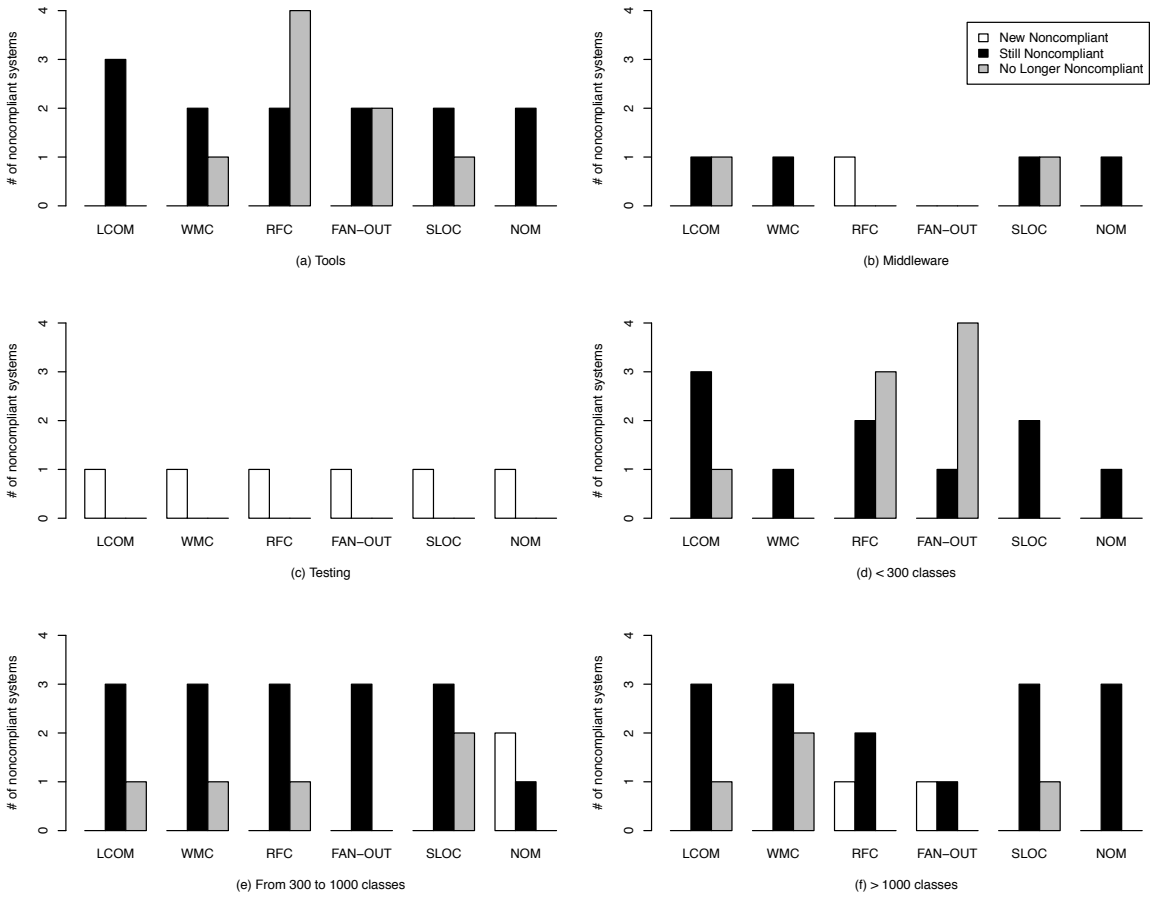


Figure 4.5. Contextual analysis for noncompliant systems

Summary of findings: The impact of context changes on the relative thresholds is limited. This impact changes is more common in k (upper limit) than p (percentage of classes that should follow the upper limit). Regarding the noncompliant systems, it is more common that noncompliants in the whole *Corpus* keep this status in the subcorpus. However, contextual changes may have a deep impact when other contextual factors are considered, *e.g.*, programming languages, proprietary systems, and number of changes.

4.7 Historical Analysis

In this section, we check how the proposed thresholds apply to different versions of the systems under analysis. Next, we verify whether classes migrate during the evolution of these systems, from a state that follow the proposed upper limits of a

relative threshold to a state that does not follow this limit and vice-versa. We also check the percentage of added and deleted classes that follow and that do not follow the proposed thresholds. Specifically, we address three research questions:

RQ #3 — Are the relative thresholds valid in different versions of the systems under analysis? Our motivation is to investigate whether the relative thresholds capture enduring design practices, which are valid in different versions of a system.

RQ #4 — Along the history of versions, do classes change their status? Our motivation is to check whether the evolution of the systems causes changes in the states of their classes. As illustrated in Figure 4.6, we analyzed two profiles of classes: (i) classes initially created not following the relative thresholds, but that no longer follow them; (ii) classes created following the relative thresholds, but that turned to do not follow them. In other words, we monitor the history of versions to check how often classes change their states.

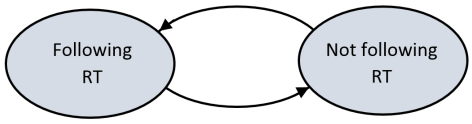


Figure 4.6. Possible states of a class: following or not the upper limit of a relative threshold

RQ #5 — What is the relation between created and deleted classes along the history of versions? Our motivation is to investigate in classes that follow and not follow the proposed thresholds which is more common: addition or deletion of classes.

4.7.1 Study Setup

To provide answers to our research questions, we consider the history of versions of five systems. Table 4.10 describes the systems, the number of classes (NOC), and the number of versions considered in this analysis. To create this historical dataset, we selected four systems that follow our thresholds for all metrics (*Lucene, Hibernate, Spring,* and *PMD*), which are systems included both in the Qualitas Corpus and in the COMETS dataset, a dataset for empirical studies on software evolution [28]. COMETS provides time series for metric values in intervals of bi-weeks. We extend this dataset to include time series on a new system (*Weka*), to also analyze a noncompliant system for all metrics. In Table 4.10, the time frame considered in the extraction ends exactly in the bi-week just before the version available in the Qualitas Corpus, *i.e.*, the version

we considered to extract the relative thresholds. The number of classes also refers to the version in the Qualitas Corpus.

Table 4.10. Systems used in the Historical Analysis

System	Period	# Versions	NOC
Lucene	01/01/2005—10/04/2008	99	946
Hibernate	06/13/2007—10/10/2010	82	1,216
Spring	12/17/2003—11/25/2009	156	1,845
PMD	06/22/2002—08/14/2009	175	1,425
Weka	11/16/2008—07/09/2010	48	1,181

4.7.2 Results

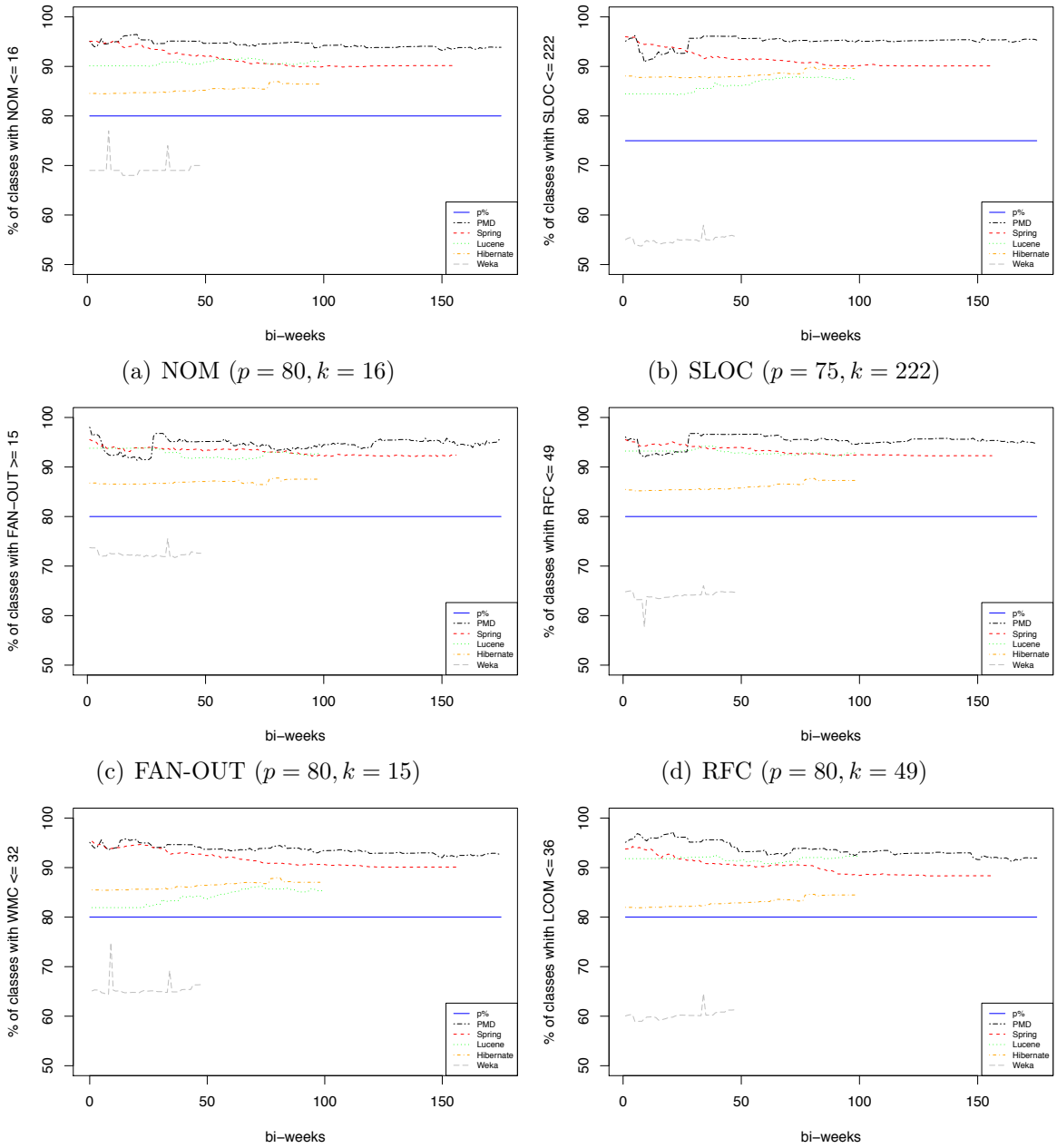
RQ #3 — Are the relative thresholds valid in different versions of the systems under analysis?

Figure 4.7 shows plots with the percentage of classes in each version and system considered in this analysis that respect the upper limit (k parameter) in the proposed relative thresholds. The four systems respecting the proposed thresholds (*PMD, Spring, Lucene*, and *Hibernate*) present the same behavior since the first considered version, for all metrics. In other words, they have never been a noncompliant system in the past. An opposite observation holds in the case of *Weka*. Along the extracted versions, *Weka* is always a noncompliant system for all metrics. Hence, we claim that the proposed thresholds are able to *capture enduring design practices* in the considered systems.

RQ #4 — Along the history of versions, do classes change their status?

To answer this RQ, we analyzed two profiles of classes: (i) classes initially created not following the upper limits of a relative threshold, but that turned to follow them in a given version; (ii) classes created following these upper limits, but turned to do not follow them.

Table 4.11 shows the percentage of classes that adhere to each of these profiles. We can observe that the percentage of classes with changes in their states tends to zero. Furthermore, when a change in a class' state occurs, it is usually towards not following the upper limit k of a relative threshold. Such changes range from 0.0% (*Hibernate*, NOM, SLOC, RFC, and LCOM) to 1.1% (*Weka*, SLOC). On the other



(a) NOM ($p = 80, k = 16$)

(b) SLOC ($p = 75, k = 222$)

(c) FAN-OUT ($p = 80, k = 15$)

(d) RFC ($p = 80, k = 49$)

(e) WMC ($p = 75, k = 32$)

(f) LCOM ($p = 80, k = 36$)

Figure 4.7. Percentage of classes following the upper limit of a relative threshold (parameter k) during the systems' evolution

side, changes in a class to make it follow the proposed upper limits are very rare. We only found these changes for FAN-OUT, in the case of three systems (*Spring*, *PMD*, and *Weka*). Finally, *Weka* (a noncompliant system) has the highest ratio of changes towards not following the proposed upper limits ($\approx 1\%$).

(a) NOM

System	ToViolate	ToFollow
Lucene	0.1	0.0
Hibernate	0.0	0.0
Spring	0.1	0.0
PMD	0.1	0.0
Weka	0.9	0.0

(b) SLOC

System	ToViolate	ToFollow
Lucene	0.2	0.0
Hibernate	0.0	0.0
Spring	0.1	0.0
PMD	0.1	0.0
Weka	1.1	0.0

(c) FAN-OUT

System	ToViolate	ToFollow
Lucene	0.1	0.0
Hibernate	0.1	0.0
Spring	0.2	0.1
PMD	0.2	0.1
Weka	0.8	0.1

(d) RFC

System	ToViolate	ToFollow
Lucene	0.1	0.0
Hibernate	0.0	0.0
Spring	0.1	0.0
PMD	0.1	0.0
Weka	0.8	0.0

(e) WMC

System	ToViolate	ToFollow
Lucene	0.2	0.0
Hibernate	0.1	0.0
Spring	0.1	0.0
PMD	0.2	0.0
Weka	0.9	0.0

(f) LCOM

System	ToViolate	ToFollow
Lucene	0.1	0.0
Hibernate	0.0	0.0
Spring	0.1	0.0
PMD	0.2	0.0
Weka	1.0	0.0

Table 4.11. Percentage of classes that changed from a state following the upper limit of a threshold to a state not following this limit (*ToViolate* column) and vice-versa (*ToFollow* column)

RQ #5 — What is the relation between created and deleted classes along the history of versions?

To provide answers to this question, we analyzed the percentage of classes that are created and deleted along the extracted versions. We analyzed this percentage in classes that follow and do not follow the proposed thresholds. Figures 4.8 and 4.9 present these results. In these figures, the percentage of created classes are represented by gray bars, while the percentage of deleted classes are represented by white bars.

Initially, we analyzed the relation between created and deleted classes that do not follow the proposed thresholds (Figure 4.8). The percentage of created classes varies from 3.2% (*Spring* for NOM) to 26% (*Weka* for NOM) and the percentage of deleted

classes varies from 3.1% (*Spring* for LOC and RFC) to 39.9% (*Weka* for NOM). We observed that in classes that do not follow the thresholds, for all systems and metrics, the percentage of deleted classes is generally greater than the percentage of created classes. There is one small exception in the case of *Spring* for NOM, LOC, FAN-OUT, RFC, and LCOM. This means that regarding classes that do not follow the thresholds is more common to delete classes than to create.

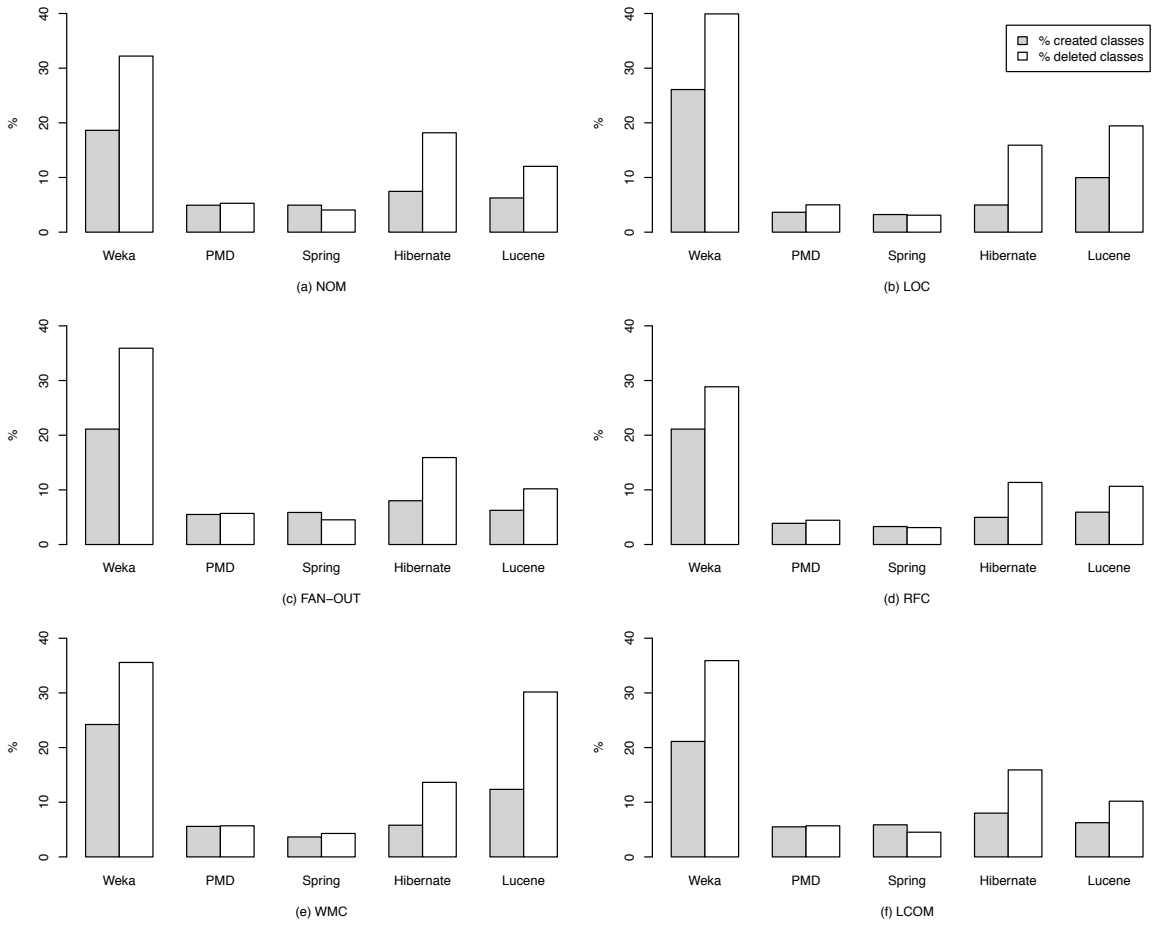


Figure 4.8. Relation between creation and deletion of classes regarding the classes that *do not follow* the relative thresholds

We also analyzed the relation between created and deleted classes among the classes that follow the proposed thresholds (Figure 4.9). We observed that for all systems and metrics, most classes are created and deleted in compliance with the proposed thresholds, as expected. The percentage of created classes varies from 73.9% (*Weka* for LOC) to 96.8% (*Spring* for LOC) and the percentage of deleted classes varies from 60.1% (*Weka* for LOC) to 97.0% (*Spring* for RFC). We observed that in classes that follow the thresholds, the percentage of created

classes is similar with the percentage of deleted classes. However, when occur differences, the percentage of deleted classes (represented by white bars) is lower than the percentage of created classes (represented by gray bars). There is one exception in the case of *Spring* for FAN-OUT, RFC, and LCOM. This means that regarding classes that follow the thresholds the percentage of delete classes is similar to create. However, in some cases to create classes is more common than delete.

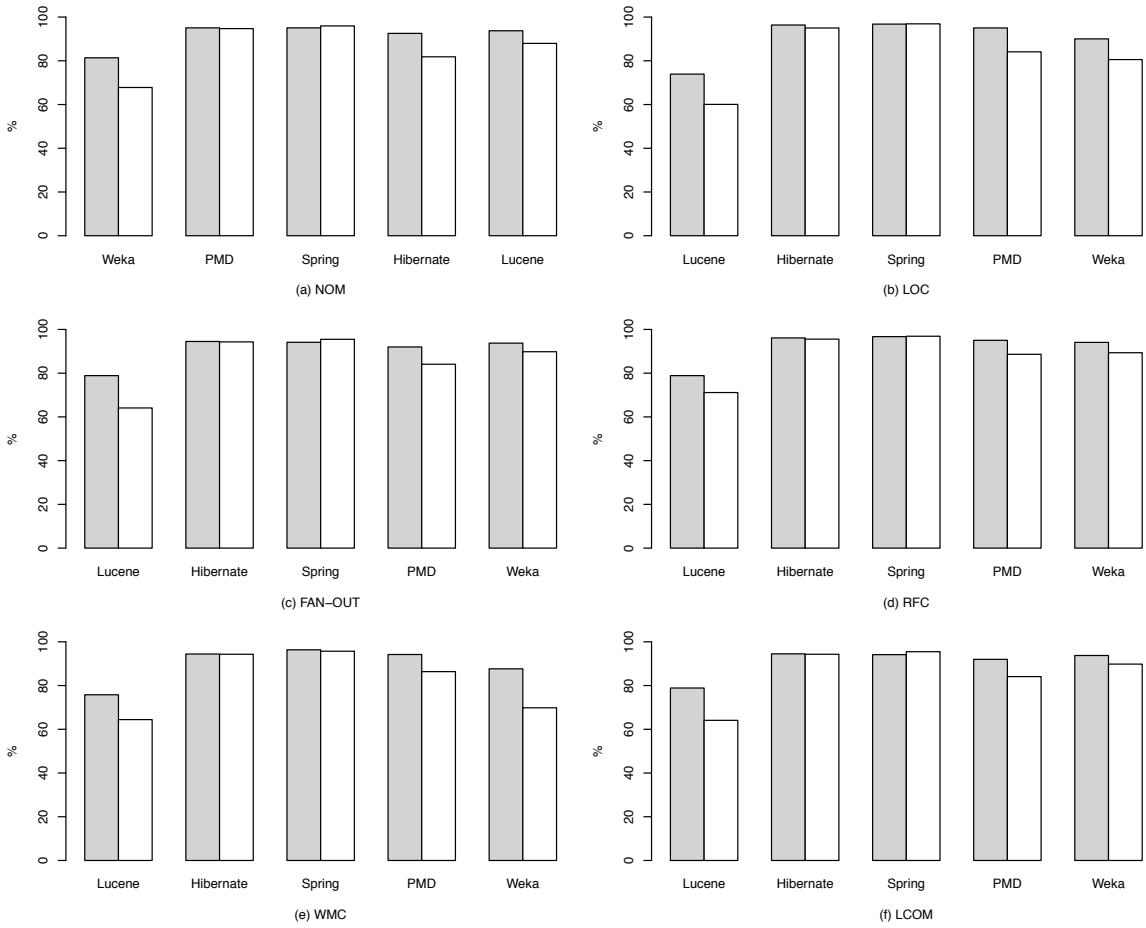


Figure 4.9. Relation between creation and deletion of classes regarding the classes that *follow* the relative thresholds. In this figure, the percentage of created classes are represented by gray bars, while the percentage of deleted classes are represented by white bars

Summary of findings: We observed that the proposed thresholds are able to *capture enduring design practices* in the considered systems. Next, we found that the percentage of classes with changes in their states tends to zero. Moreover, we found that the percentage of class deletions is generally greater among classes that do not follow the thresholds. In contrast, when we analyzed the percentage of classes that follow

the thresholds, we found that the percentage of class deletions is generally similar to percentage of created classes.

4.8 Change Analysis

In this section, we aim to check the importance of classes that do not follow the upper limit of a relative threshold, by checking how often such classes are changed. Thereby, we designed a study to address two research questions:

RQ #6 — What is the percentage of changes in classes that do not follow the upper limit of a relative threshold? Our motivation is to investigate whether these classes are important in terms of maintenance activities or if they are stagnant classes.

RQ #7 — Are changes in classes that follow and classes that do not follow the upper limit of a relative threshold proportional to their number in the evaluated systems? Our motivation is to check which type of classes have the highest rate of changes: classes that follow or classes that do not follow the proposed upper limits.

4.8.1 Study Setup

To provide answers to our research questions, we analyzed the same metrics and systems used in Section 4.7. Table 4.12 shows general information on the commits we considered in this analysis, the total number of different classes found in the extracted commit logs (column #Classes), and the total number of changes in such classes (column #Changes).

Table 4.12. Data on commits log

System	#Commits	#Classes	#Changes
Lucene	702	1,618	15,284
Hibernate	861	3,562	3,819
Spring	1,767	3,818	3,628
PMD	1,414	682	6,054
Weka	329	1,112	8,496

4.8.2 Results

RQ #6 — What is the percentage of changes in classes that do not follow the upper limit of a relative threshold?

Figure 4.10 shows plots with the percentage of changes in classes that do not follow the upper limit of a relative threshold, for each system and metric considered in this analysis. These classes concentrate a considerable percentage of maintenance activities, ranging from 20% (*PMD*, SLOC) to 73% (*Weka*, SLOC). In other words, 73% of the changes detected in the analyzed commits are in classes that have more than 222 SLOC, in the case of *Weka* (as reported in Section 4.3 the upper limit of the proposed relative threshold for SLOC is 222). This result is explained by the fact that *Weka* is a noncompliant system, and therefore it has more classes with metric values greater than the parameter k. Hence, RQ #7 analyzes changes per class for each system and metric considered.

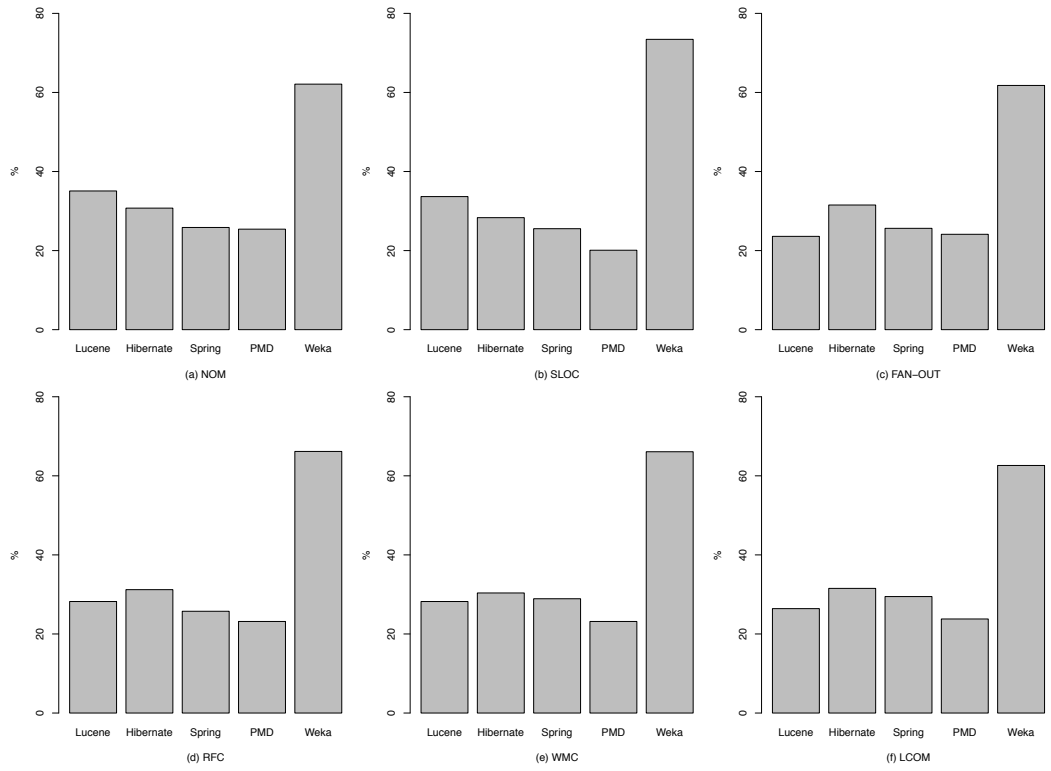


Figure 4.10. Percentage of changes in classes that do not follow the upper limits of the relative thresholds proposed for NOM, SLOC, FAN-OUT, RFC, WMC, and LCOM

RQ #7 — Are changes in classes that follow and classes that do not follow the upper

limit of a relative threshold proportional to their number in the evaluated systems?

To answer this second research question, we calculated the number of changes per classes that follow and that do not follow the upper limits of the analyzed relative thresholds, as reported in Figure 4.11. In all cases, the rate of changes in classes that do not follow the proposed upper limits is greater than the rate of changes in the other classes (the gray columns are always higher than the black ones). *PMD* is the system with the highest rate of changes in the classes that do not follow the proposed upper limits. In *PMD*, this rate ranges from 23 changes/class (LCOM) to 44 changes/class (RFC and WMC). Regarding the results for classes that follow the upper limits, *Lucene* is the system with the highest rate of changes per classes. This rate ranges from 6.82 changes/class (NOM) to 8.0 changes/class (FAN-OUT).

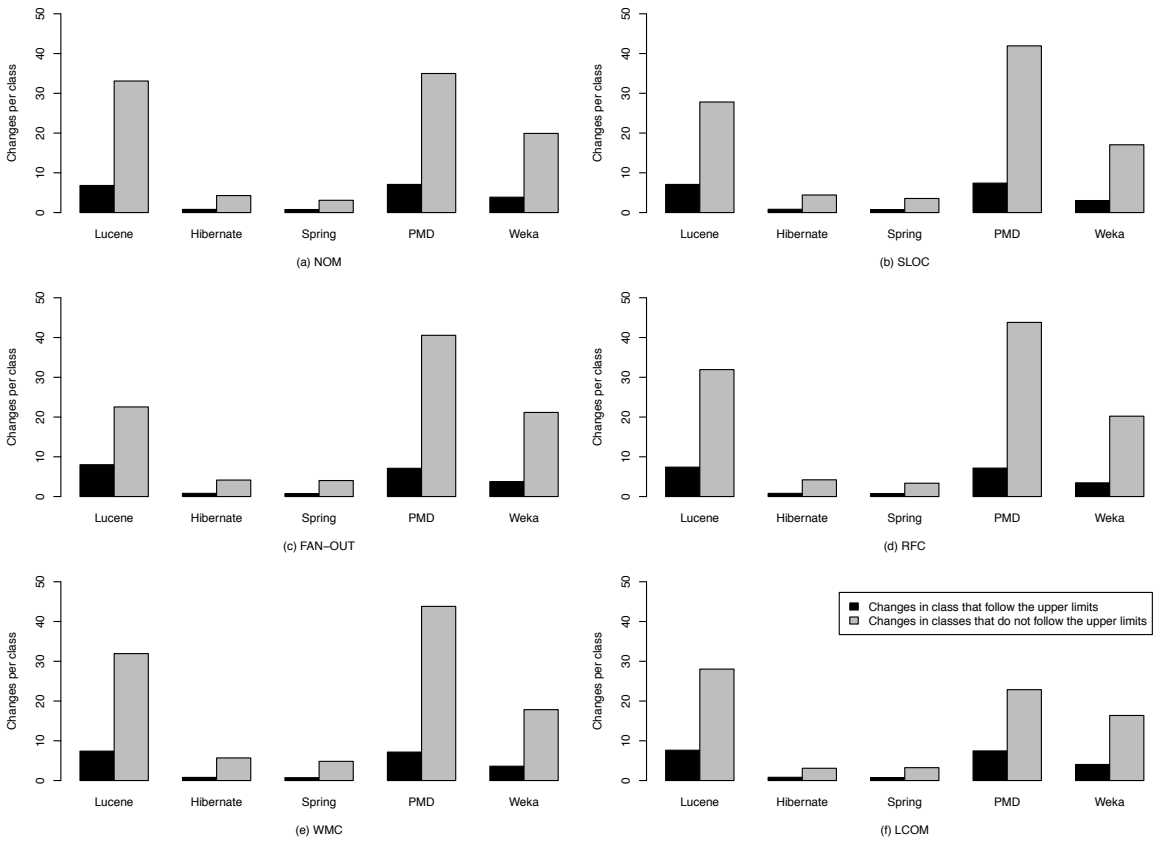


Figure 4.11. Number of changes per classes that follow and that do not follow the upper limits of the relative thresholds proposed for NOM, SLOC, FAN-OUT, RFC, WMC, and LCOM

We also inspected classes with the highest number of changes in two systems: *Lucene* and *Spring*. These systems are selected because *Lucene* has the highest

number of changes per classes (9.5) and *Spring* has the smallest number of changes per classes (1.0), as showed in Table 4.12. Tables 4.13 and 4.14 present the top-15 classes with the highest number of changes in these systems. The tables also report whether each class follow, represented by "yes", or do not follow, represented by "-", the upper limits of the relative thresholds for six metrics. The number of changes in *Lucene* and *Spring* are very different, ranging from 174 to 966 changes per classes in *Lucene*, and from 22 to 59 changes per classes in *Spring*. However, the results for both systems show that most highly-changed classes do not follow the upper limits for all metrics. For example, in *Spring* all top-15 classes do not follow the proposed upper limit for SLOC. In contrast, LCOM is the metric with the highest number of classes that follow the upper limit (four classes in *Lucene* and five classes in *Spring*).

Summary of findings: We conclude that classes that do not follow the upper limits are important in terms of maintenance activities. We also observed that the noncompliant system (*Weka*) does not have more changes per class than compliant systems.

Table 4.13. Top-15 classes with the highest number of changes in *Lucene*. The table also shows whether each class follow or not the proposed upper limits for the relative thresholds of six metrics

		Follow upper limits?					
Class	**Changes**	**NOM**	**SLOC**	**FAN-OUT**	**RFC**	**WMC**	**LCOM**
IndexWriter	966	-	-	-	-	-	-
IndexReader	472	-	-	-	-	-	-
SegmentReader	412	-	-	-	-	-	-
DocumentsWriter	304	-	-	-	-	-	yes
SegmentMerger	296	-	-	-	-	-	yes
CheckIndex	282	yes	-	-	-	-	yes
FSDirectory	262	-	-	-	-	-	-
MemoryIndex	256	yes	-	-	-	-	yes
IndexSearcher	246	-	-	-	-	-	-
BooleanQuery	220	-	yes	-	-	-	-
DirectoryReader	198	-	-	-	-	-	-
Field	182	-	-	yes	yes	-	-
QueryParser	180	-	-	-	-	-	-
SegmentInfo	180	-	-	yes	-	-	-
FieldCacheImpl	174	-	yes	-	-	yes	-

Table 4.14. Top-15 classes with the highest number of changes in *Spring*. The table also shows whether each class follow or not the proposed upper limits for the relative thresholds of six metrics

		Follow upper limits?					
Class	**Changes**	**NOM**	**SLOC**	**FAN-OUT**	**RFC**	**WMC**	**LCOM**
TypeDescriptor	59	-	-	yes	-	-	-
HandlerMethodInvoker	38	-	-	-	-	-	yes
AbstractBeanFactory	35	-	-	-	-	-	-
RestTemplate	35	-	-	-	-	-	-
AbstractApplicationContext	34	-	-	-	-	-	-
BeanWrapperImpl	33	-	-	-	-	-	-
DefaultListableBeanFactory	31	-	-	-	-	-	-
Indexer	27	yes	-	-	-	-	yes
TypeConverterDelegate	27	yes	-	-	-	-	yes
StandardTypeConverter	26	yes	-	yes	yes	yes	yes
ConstructorResolver	26	yes	-	-	-	-	yes
AutowireCapableBeanFactory	24	-	-	-	-	-	-
ExpressionState	24	-	-	yes	-	yes	-
MethodHandlerAdapter	22	-	-	-	-	-	-

4.9 Bad Smells Analysis

Duplicated code, overly complex methods, non-cohesive classes, and long parameter lists are possible signs of degradation in the design of software system [62, 71]. These signs are usually known as design flaws [70], bad smells [39], or anti-patterns [16]. To investigate the relation between the presence of bad smells in a system and its adherence to the proposed relative thresholds, we designed a study to address the following research question:

RQ #8 — Do noncompliant systems have more bad smells? Our motivation is to investigate whether the proposed relative thresholds can be used to reveal systems with high number of bad smells.

4.9.1 Study Setup

For this study, we used the *Tools* subset of the Qualitas Corpus, which has 26 systems of different size. Table 4.15 presents the noncompliant systems. To detect the presence of bad smells in this subcorpus, we use the `inCode` tool[6]. `InCode` is an Eclipse plug-in

[6]https://www.intooitus.com/products/incode, verified 11/25/2014.

that automatically detects bad smells using metric-based rules that capture deviations from good design principles [70]. We relied on a subset of Qualitas Corpus because we need to manually perform inCode in each system (from Java files). Table 4.16 shows the design and code anti-patterns (bad smells) detected by inCode for object-oriented systems. The smells are reported for classes (*e.g.*, Data Class) or methods (*e.g.*, Feature Envy)

Table 4.15. Noncompliant systems in the *Tools* subcorpus

Metrics	**Outliers Systems**
NOM	Compiere, Weka
SLOC	Compiere, Weka
FAN-OUT	JAG, JMoney
RFC	Compiere, Weka
WMC	Compiere, Weka
LCOM	Compiere, JFreeChart, Weka

Table 4.16. Evaluated bad smells

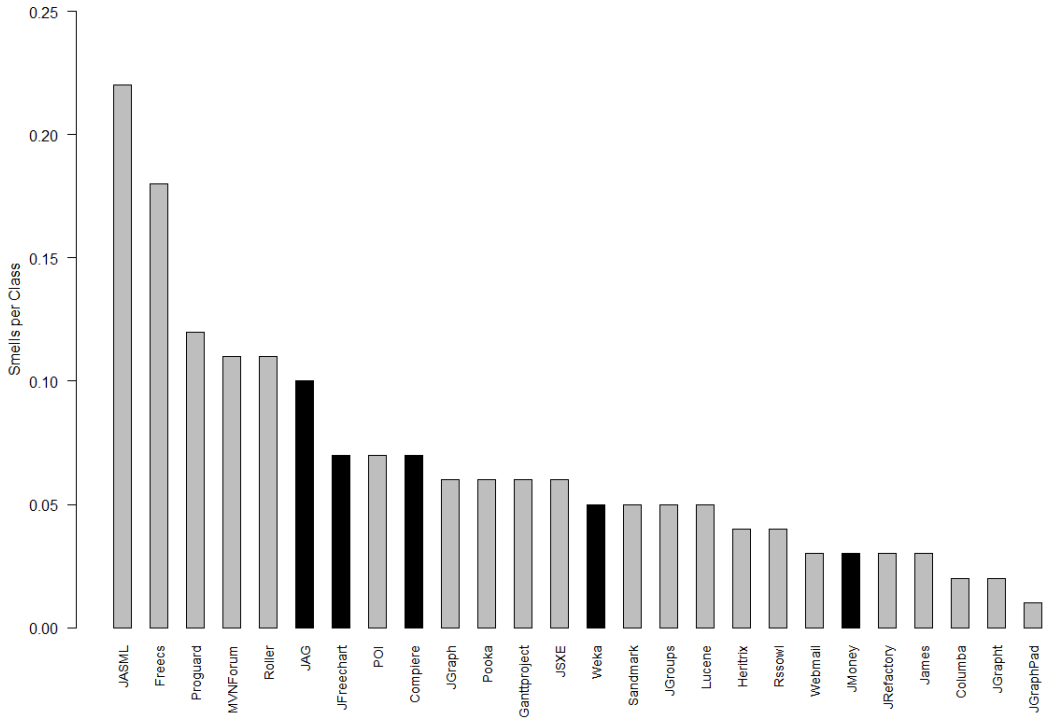
Class-Level	**Method-Level**
Data Class	Feature Envy
Tradition Breaker	Data Clumps
Schizophrenic Class	Sibling Duplication
God Class	Internal Duplication
	External Duplication
	Message Chains

4.9.2 Results

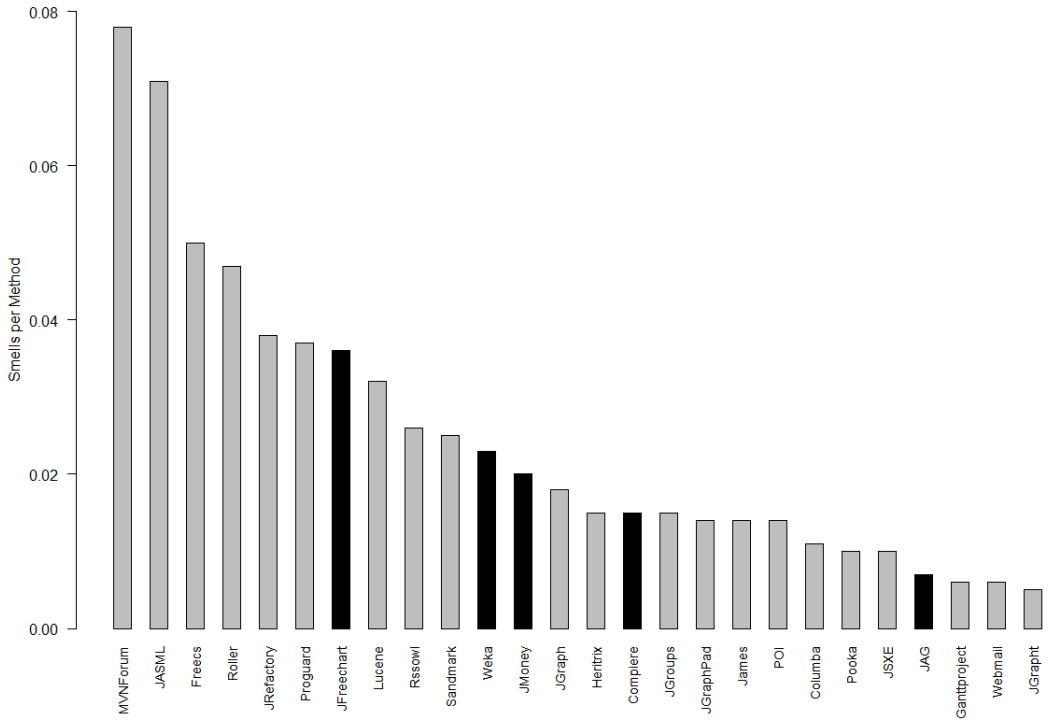
RQ #8 — Do noncompliant systems have more bad smells?

To answer this research question, Figure 4.12 presents plots with: (a) the number of class-level bad smells per classes in each system and (b) the number of method-level bad smells per methods in each system. The x-axis shows the 26 systems considered in this analysis. The y-axis represents the rate of bad smells. The black bars represent noncompliant systems and the gray bars represent compliant systems. As we can observe, the rate of bad smells per method is smaller than the rate of bad smells per classes. The former ranges from 0.01 (*JGraphPad*) to 0.22 (*JASML*) and the later ranges from 0.00 (*JGrapht*) to 0.08 (*MVNForum*). More importantly, in the considered

subcorpus there is no evidence that noncompliant systems have more density of bad smells. For example, for both class-level and method-level smells, the top-5 systems with the highest density of bad smells follow the proposed thresholds.



(a) Bad smells per classes



(b) Bad smells per methods

Figure 4.12. Rate of class-level and method-level bad smells in systems in the *Tools* subcorpus. The black bars represent noncompliant systems and the gray bars are compliant systems

4.10 Inequality Analysis

We evaluated the dispersion of the metric values in the systems respecting the proposed thresholds, using the Gini coefficient. Gini is a coefficient widely used by economists to express the inequality of income in a population [102]. The coefficient ranges between 0 (perfect equality, when everyone has exactly the same income) to 1 (perfect inequality, when a single person concentrates all the income). Gini has been applied in the context of software evolution and software metrics [100, 102], although not exactly to evaluate the reduction in inequality achieved by following metric thresholds.

In the analysis we consider the distributions of NOM values in the original *Corpus*. First, we calculated the Gini coefficient considering the whole population of classes in each system. Next, we recalculated the coefficient for the classes respecting the upper threshold of 16 methods. In both cases, we excluded the systems with a noncompliant behavior, since our goal is to reveal the degree of inequality in systems respecting our method. The boxplots in Figure 4.13 summarizes the Gini results in our systems. As we can observe, the median Gini coefficient considering the whole population of classes in each system is 0.5. By considering only classes with 16 or less methods, the median coefficient is reduced to 0.5. In fact, this reduction in dispersion is expected, since we removed the high values in the long tail.

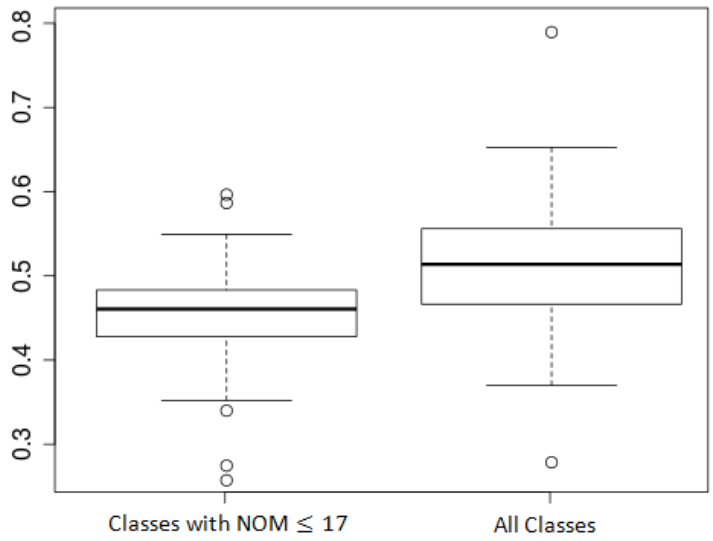


Figure 4.13. Inequality Analysis using Gini coefficients

We also analyzed the noncompliants in the sample filtered by the proposed threshold. As observed in the right boxplot in Figure 4.13, we have noncompliants due to very equal distributions (Gini < 0.4) and also noncompliants due to very unequal distributions (Gini > 0.6). For example, *JParser* is an example of the first case (Gini=0.3) and

CheckStyle is an example of the second case (Gini=0.6). Figure 4.14 shows the quantile functions for these two systems. We can see that most classes in *JParser* respecting the proposed threshold have between 5 to 10 methods, while in *CheckStyle* we have a representative number of classes with less than five methods, between 5 to 10 methods, and also with more than 10 methods.

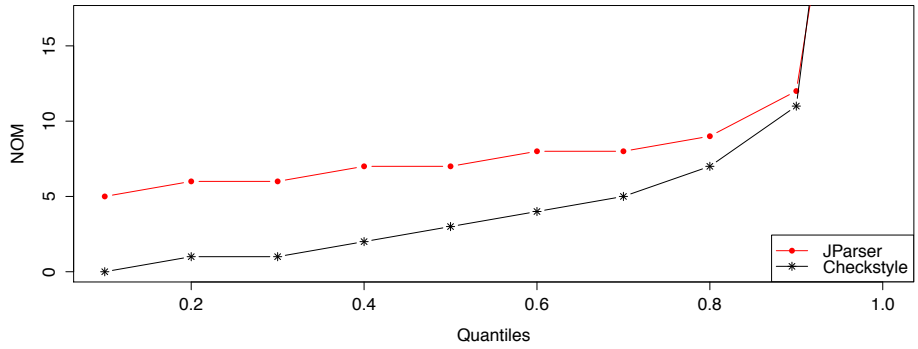


Figure 4.14. Quantile functions for noncompliants regarding the Gini values

Although *JParser* and *CheckStyle* have very different Gini coefficients, we can not claim they are noncompliants in terms of software quality. In other words, a system with classes ranging from 5 to 10 methods (*JParser*) seems to be not very different than a system having classes with 1 to 16 methods (*Checkstyle*), at least in terms of their internal software quality.

Therefore, as revealed by the Gini coefficients, the inequality analysis shows that there are different distributions of methods per class among the systems that follow the proposed thresholds. However, such differences do not seem to have major impacts in terms of software quality. More specifically, at least in our *Corpus*, we have not found degenerate distributions, both in terms of equality or inequality, *e.g.*, a system with all classes having exactly a single method or a system with half of the classes having $k-1$ methods and half of the classes having very few methods. Although such distributions may respect our thresholds, they would certainly reveal serious design problems. However, it is hard to believe that distributions like that are possible in practice.

4.11 Threats to Validity

In this section, we discuss possible threats to validity, following the usual classification in threats to external and internal validity:

Threats to External Validity: The main threat is the representativeness of our Corpora. The Qualitas Corpus may not be representative to capture relative thresholds in different domains, including systems in other programming languages, proprietary systems, etc. However, it is important to note that we do not aim to propose universal relative thresholds, but instead we just claim that the relative thresholds proposed in this paper apply at least to open-source Java systems. Moreover, we consider in our studies six metrics, covering important source code properties, like size, coupling, complexity, and cohesion. Despite these observations, we cannot guarantee—as usual in empirical software engineering—that our findings apply to other metrics.

Threats to Internal Validity: A possible internal validity threat is related to the fact that we did not inspect the systems in the Qualitas Corpus to remove for example classes generated automatically by tools like parsers generators [3]. However, our central goal with these studies is not establishing an industrial software quality benchmark, but to illustrate the use of our method in a real software corpus, including a discussion on its main properties. Moreover, most systems usually do not have many classes generated automatically.

4.12 Final Remarks

In this chapter, we initially derive relative thresholds for six source code metrics, using the Qualitas Corpus. Next, we report an extensive relative threshold analysis divided in seven studies, as follows:

1. We investigated whether 308 Java repositories, available at GitHub, follow the proposed relative thresholds (Section 4.4). We found that most popular GitHub repositories indeed follow our thresholds.

2. We compared the proposed relative thresholds with thresholds extracted using the SIG method [5] (Section 4.5). We concluded that both methods convey similar information. However, our method derives relative thresholds that can be automatically used to detect noncompliant systems.

3. We evaluated the influence of the context in our results and we concluded that the impact on relative thresholds of context changes is limited (Section 4.6).

4. We investigated how the proposed thresholds apply to different versions of the systems under analysis. In this study, we also investigated whether classes migrate

from a compliant to a noncompliant state (or vice-versa) during their evolution (Section 4.7). We observed that the proposed thresholds capture enduring design practices. Moreover, we also found that the percentage of classes with changes in their states tends to zero.

5. We investigated the importance of classes that do not follow the upper limit of a relative threshold, by checking how often such classes are changed (Section 4.8). We concluded that classes that do not follow the upper limits are important in terms of maintenance activities.

6. We investigated the relation between the presence of bad smells in a system and its adherence to the proposed relative thresholds (Section 4.9). We found that in the considered subcorpus there is no evidence that noncompliant systems have a higher density of bad smells.

7. Finally, we evaluated the dispersion of the metric values in the systems respecting the proposed thresholds, using the Gini coefficient (Section 4.10). This inequality analysis showed that there are different distributions of methods per class among the systems that follow the proposed thresholds. However, such differences do not seem to have major impacts in terms of software quality.

Chapter 5

Validating Relative Thresholds with Developers

In this chapter, we report results of a study designed to validate our method to extract relative thresholds. We extract thresholds from a benchmark of 79 Pharo/Smalltalk systems, which are validated with five Pharo experts and 25 Pharo developers. This chapter is organized as follows. Section 5.1 describes the design of our empirical study. Next, Section 5.2 reports our results. Section 5.3 presents a critical analysis and Section 5.4 reports final remarks.

5.1 Study Design

In this section we present the questions that motivated this chapter (Section 5.1.1). Next, we present the *Corpus* and the considered source code metrics (Section 5.1.2). Finally we describe the methodology and participants of our study (Section 5.1.3).

5.1.1 Research Questions

Our goal is to validate with expert developers the relative thresholds derived. To achieve this goal, we pose three research questions:

RQ #9 Do systems perceived as well-written by expert developers follow the derived relative thresholds?

RQ #10 Do systems perceived as poorly-written by expert developers do not follow the derived relative thresholds?

RQ #11 Do the noncompliant systems require more effort to maintain? By main noncompliant we refer to systems that do not respect the relative thresholds on multiple metrics.

5.1.2 Corpus and Metrics

In order to validate our notion of relative thresholds with software developers, we use a *Corpus* of 79 Pharo systems[1]. We initially select 39 systems found in the Pharo standard distribution. From these 39 systems, 18 may be considered as legacy, although they are intensively in use, covered by unit tests, and have received numerous contributions by the community. In addition, we select 40 systems from the Pharo forge[2]. These additional systems are selected based on their size, popularity, activity, and relevance for the Pharo ecosystem. Most of these systems are part of specialized distributions of Pharo (namely *Moose* and *Seaside*), confirming their relevance and maturity. Figure 5.1 describes the size of the systems in our corpus in terms of classes. We report the size after excluding unit tests (which are also implemented as classes). Tests classes are removed because they usually have a structure radically different from production code. Specifically, presence of unit test code, which usually has very little complexity, will result in lower threshold values. For this study, tests classes were removed because they usually have a structure radically different from production code.

In this study, we validate relative thresholds for the following four source code metrics computed by the Moose software analysis platform[3]: (a) Number of Attributes (NOA)—Moose computes this metric by counting all attributes in the class; (b) Number of Methods (NOM)—Moose computes this metric by counting all methods in the class, including constructors, getters, and setters; (c) Number of Provider Classes (FAN-OUT)—Moose computes this metric by considering all types of class dependencies (due to inheritance, method calls, static accesses, etc.); and (d) Weighted Method Count (WMC)—Moose computes this metric as the sum of the cyclomatic complexity of each method in a class. We selected these metrics because they covey distinct factors affecting the internal quality of object-oriented systems, such as size, coupling, and complexity.

[1] A detailed description is available at `http://aserg.labsoft.dcc.ufmg.br/pharo-dataset`.
[2] `http://smalltalkhub.com`, verified on 06/15/2015.
[3] `http://www.moosetechnology.org/`, verified on 06/15/2015.

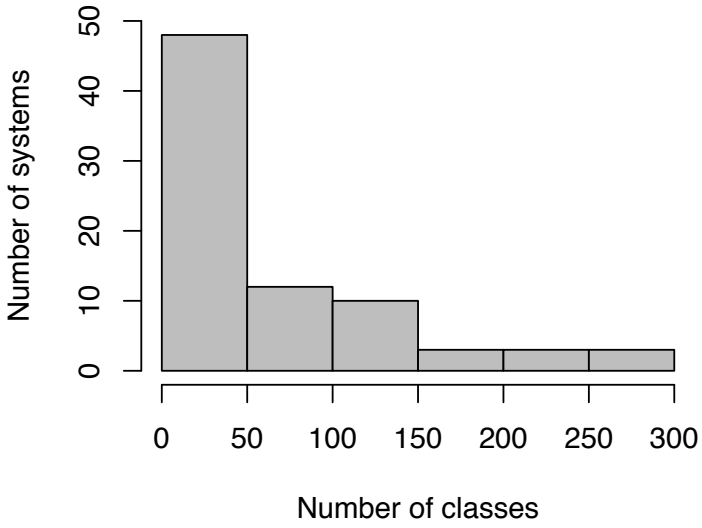


Figure 5.1. Size of the systems in the Pharo *Corpus*

5.1.3 Methodology and Participants

Recall that our goal is to validate the notion of relative thresholds with Pharo practitioners. To achieve this goal, we follow a mixed-method approach [7, 57]. We initially conducted an interview involving five Pharo experts (*i.e.*, people deeply committed to the Pharo development and success) and a broader survey with 25 Pharo maintainers (*i.e.*, people in charge of incorporating system improvements and producing new releases). A mixed-method approach is chosen to counter-balance the shortcomings of each one of the methods and to ensure that the results are valid and representative in the Pharo ecosystem.

Specifically, to answer RQ #9 and RQ #10, we asked five experts in Pharo, which are members of the Pharo board, to provide examples of systems that are "well-written" and "not well-written". The choice of these terms is based on the outcome of a pilot study, which indicated that "well-written" and "not well-written" are largely understood by practitioners as opposed to terms such as "maintainability", that practitioners had difficulty on interpreting.

To answer RQ #9, we focus on systems that do not respect the derived relative thresholds for at least two metrics[4]. We call such systems *main noncompliant* and we interviewed the top maintainers of each one. A top maintainer is a developer that has written most of the methods found in the last release of the system. Specifically, we identified the five top maintainers in each noncompliant system by ranking authors of

[4]We did not consider a single metric to avoid the "One-track metric" anti-pattern, which occurs when a single metric is used to measure software quality [15].

each system according to the number of contributed methods (*i.e.*, defined or modified methods). We asked these maintainers the following question: *Compared to other systems you work with, the system in question requires (a) more effort to maintain; (b) a comparable effort to maintain; (c) less effort to maintain.*

5.2 Results

In this section, we first present the relative thresholds for the source code metrics considered in this chapter, derived using the Pharo *Corpus* (Section 5.2.1). Next, we describe the results of our research questions (Sections 5.2.2 to 5.2.4).

5.2.1 Relative Thresholds for the Pharo Corpus

Table 5.1 presents the relative thresholds derived by our method. For each metric, the table shows the values of p and k that define a relative threshold and the number of noncompliant systems. We can observe that the upper limit k of the derived relative thresholds are valid for a large number of classes (parameter p), but not for all classes in a system. In fact, the value of p ranges from 75% to 80%. The number of noncompliant systems range from six (FAN-OUT) to 14 (WMC), *i.e.*, from 7.6% to 17.7% of the systems in the *Corpus*.

Table 5.1. Relative Thresholds for Pharo

Metrics	p	k	# noncompliant systems
NOA	75	5	9
NOM	75	29	11
FAN-OUT	80	9	6
WMC	75	46	14

Table 5.2 presents nine systems considered as main noncompliant, *i.e.*, systems that do not respect the relative thresholds for two or more metrics, as described in Section 5.1.3.

5.2.2 RQ 9: Do systems perceived as well-written by the expert developers follow the derived relative thresholds?

To answer this question, we asked five Pharo experts to indicate well-written Pharo systems. Table 5.3 presents these systems, including a brief description, and the experts that elected the system. Among the seven systems named by the experts, *Roassal* and

Table 5.2. Main noncompliant systems

Main noncompliant	Metrics			
	NOA	NOM	FAN-OUT	WMC
Collections		✓		✓
ComandShell		✓	✓	✓
Files			✓	✓
Graphics	✓	✓	✓	✓
Kernel		✓		✓
Manifest	✓	✓		✓
Morphic		✓		✓
Shout	✓	✓	✓	✓
Tools	✓	✓	✓	✓

Zinc belong to the *Corpus*. We claim that this overlap does not affect validity of our analysis. In fact, benchmark-based methods to derive thresholds depend on a balanced corpus, including both well and poorly-written systems. In other words, the overlapping shows that our *Corpus* includes well-written systems, but as expected the Pharo ecosystem also has other well-written systems.

Table 5.3. Well-written systems

Systems	Description	Voted by	Corpus
PetitParser	Parser framework	Expert #1	
PharoLauncher	Platform to manage Pharo images	Expert #2	
Pillar	Markup language and tools	Expert #2	
Roassal	Visualization engine	Expert #3	✓
Seaside	Web framework	Expert #4	
SystemLogger	Log framework	Expert #5	
Zinc	HTTP framework	Expert #5	✓

For each voted system, we evaluate their percentage of classes respecting the k-value of the proposed relative threshold for each metric. The results are summarized in Table 5.4. For instance, the relative threshold for NOA is $[75, 5]$ and the table shows that 100% of the classes of *PetitParser* have five attributes or less, *i.e.*, *PetitParser* respects the relative threshold for NOA.

As can be observed in Table 5.4, the well-written systems respect the proposed relative thresholds for all metrics with the notable exception of FAN-OUT. The only systems that respect the relative threshold for FAN-OUT are *SystemLogger* and *Zinc*. To explain this fact, we investigated the distribution of the FAN-OUT values in the *Corpus* and in the well-written systems reported by the Pharo experts. Figure 5.2 shows

Table 5.4. Percentage of classes in the well-written systems that follow the upper limit k of a relative threshold (underlined values show the cases when the thresholds are not respected).

Systems	**Metrics**			
	NOA	NOM	FAN-OUT	WMC
[p,k]	[75,5]	[75,29]	[80,9]	[75,46]
PetitParser	100	97	<u>41</u>	97
PharoLauncher	97	97	<u>74</u>	99
Pillar	97	94	<u>62</u>	95
Roassal	91	90	<u>24</u>	93
Seaside	97	96	<u>41</u>	96
SystemLogger	100	92	100	92
Zinc	91	82	81	82

the quantile functions for the FAN-OUT values, *i.e.*, the x-axis represents the quantiles and the y-axis represents the upper metric values for the classes in the quantile. The noncompliant systems for FAN-OUT, *i.e.*, *PetitParser*, *PharoLauncher*, *Pillar*, *Roassal*, and *Seaside*, are represented by dashed lines, while the remaining systems (*Corpus* and well-written systems that follow the thresholds) are represented by solid lines. We can observe that the noncompliant systems have very different distribution of FAN-OUT values than systems in our *Corpus*.

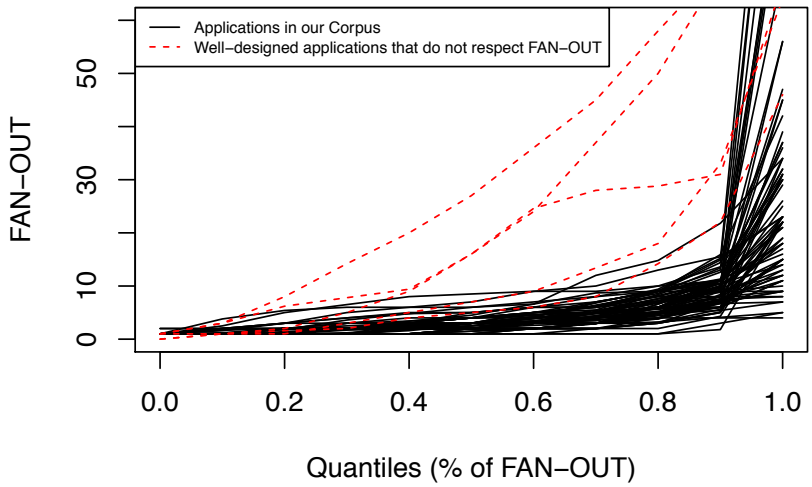


Figure 5.2. FAN-OUT quantiles—dashed lines represent *PetitParser*, *PharoLauncher*, *Pillar*, *Roassal*, and *Seaside*, which are systems perceived as well-written but that do not follow the relative threshold for FAN-OUT

Furthermore, we took a closer look at the way Moose computes FAN-OUT. When determining this metric, Moose considers all types of class dependencies introduced,

e.g., by means of inheritance, method calls, static accesses, etc. Therefore, one way for a class to have a high FAN-OUT is to be a client of an extensive inheritance hierarchy with many instances of overridden methods, as exemplified in Figure 5.3. In this figure, *ClassC* is a subclass of *ClassB*, which is a subclass of *ClassA*. The former implements a method m_1, which is overridden in *ClassB* and *ClassC*. *ClassD* has a method m_2 that calls method m_1 of *ClassA*. In this case, *ClassD* has FAN-OUT equal to three. This occurs because it is not possible to infer the exact implementation of m_1 that it is called. A preliminary inspection in the source code shows that this is exactly the case of *PetitParser*, *PharoLauncher*, *Pillar*, *Roassal*, and *Seaside*.

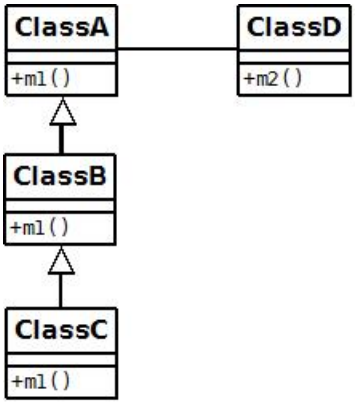


Figure 5.3. FAN-OUT example

We also evaluate the distribution of the FAN-OUT values using the Gini coefficient. Initially, we calculate the Gini coefficient considering the whole population of classes in each system, as summarized in the left boxplot in Figure 5.4. This boxplot shows that there are two noncompliant systems in the *Corpus*: *ProfStef* and *HelpSystem-Core* with Gini equals 0.24 and 0.25, respectively. Next, we calculate the coefficient for the systems with good design quality, as indicated by the Pharo experts. The results are presented in the right boxplot of Figure 5.4. We can observe that the Gini coefficients of these systems are similar to the ones of the systems in the *Corpus*. They range from 0.41 (*PetitParser*) to 0.58 (*Pillar*). In the *Corpus*, the median Gini coefficients is 0.49; for the well-designed systems it is 0.48. We should interpret such results as follows. In the *Corpus*, most classes have small FAN-OUT values. In the expert's systems, most classes have higher FAN-OUT values. Despite that the Gini coefficients are similar because this coefficient measures relative and not absolute wealth (which is represented in our case by FAN-OUT values).

Summary of findings: We observe that systems perceived as well-written by the interviewed experts follow the proposed relative thresholds for NOA, NOM, and WMC.

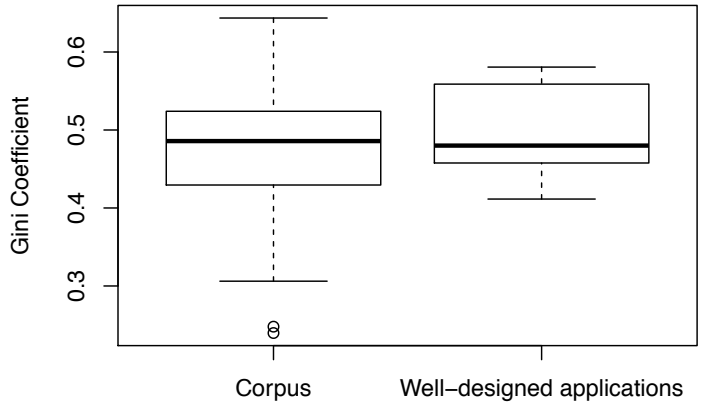


Figure 5.4. Gini coefficients

However, this does not happen for FAN-OUT, as *SystemLogger* and *Zinc* are the only systems that follow the relative threshold for this metric. The reason for "too many high FAN-OUT" values being present in five well-written systems seems to be the presence of extensive inheritance hierarchies with many instances of overridden methods. This finding stresses the importance of considering multiple metrics when determining whether a system might be problematic from the point of view of its internal quality.

5.2.3 RQ 10: Do systems perceived as poorly-written by the expert developers do not follow the derived relative thresholds?

To answer this research question, we asked the five experts to indicate poorly-written systems. This question turned out to be much more difficult, since only two experts answered it. Table 5.5 presents these systems, including a brief description, and the experts that suggested the system. This difficulty to identify poorly-written systems might be explained by the respondents being rather optimistic than pessimistic, or by them not being comfortable with admitting that some systems are not well-written.

Table 5.5. Poorly-written systems

Systems	Description	Voted by	Corpus
Metacello	Versioning system	Expert #4	✓
Morphic	Graphical interface framework	Experts #2 and #4	✓

For *Metacello* and *Morphic*, Table 5.6 shows the percentage of classes respecting the k-value of the relative thresholds proposed in this work.

Table 5.6. Percentage of classes in the poorly-written systems that follow the upper limit k of a relative threshold (underlined values show the cases when the thresholds are violated).

Systems	Metrics			
	NOA	NOM	FAN-OUT	WMC
[p,k]	[75,5]	[75,29]	[80,9]	[75,46]
Metacello	93	82	86	79
Morphic	77	<u>74</u>	83	<u>71</u>

On the one hand, we found that *Morphic* is a main noncompliant system, *i.e.*, it does not follow the proposed relative thresholds for two metrics: NOM and WMC. For example, Expert #2 made the following comments about Morphic:

"Morphic is an old system and there is no test and sparse documentation".

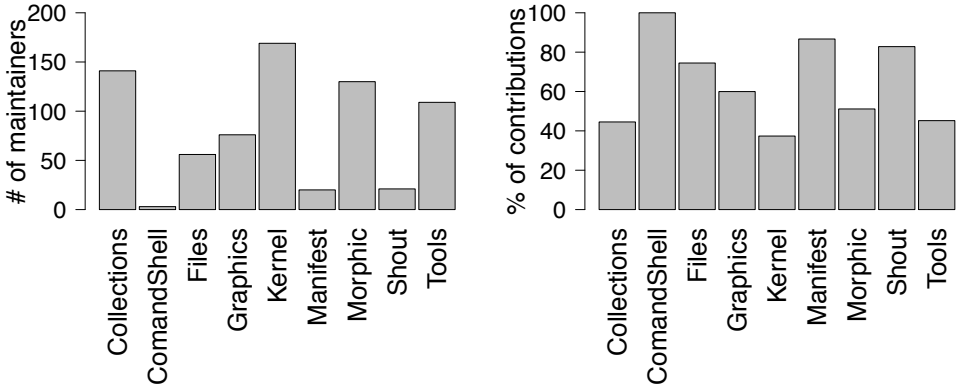
On the other hand, *Metacello* follows the relative thresholds for all metrics. This system supports a complex domain-specific language to express intricate relations between different versions of Pharo packages (*e.g.*, this language allows a developer to define that package X depends on version v_1 and v_2 of package Y, only on a platform P). It also takes care of determining cyclic dependencies and identifying proper versions required in presence of multiple dependencies. One Pharo expert argued that the complexity of the versioning domain makes *Metacello* very hard to understand, and there is an on-going effort to replace the system. Therefore, we claim that the perception of *Metacello* as poorly written is more likely to be caused by the inherent complexity of the versioning domain rather than by a problematic design.

Summary of findings: Violation of two relative thresholds in *Morphic* agrees with its design being perceived as problematic. However, this is not the case of *Metacello*. Probably, *Metacello* was cited as poorly-written due to the complexity of its domain.

5.2.4 RQ 11: Do the noncompliant systems require more effort to maintain?

Before answering this RQ, we analyze the importance of the Top-5 maintainers in the noncompliant systems. Figure 5.5(a) presents the number of maintainers of each non-compliant. We can observe that this number ranges from three (*ComandShell*) to 169 (*Kernel*). Six out of nine noncompliant systems have more than 50 maintainers, which reinforces the relevance of these systems in the Pharo Ecosystem. Figure 5.5(b) shows

the percentage of contributions by the considered top maintainers. Recall that we ranked the maintainers according to the percentage of their contributions in each non-compliant system. We can observe that the top-5 maintainers contributions range from 37% (*Kernel*) to 100% (*ComandShell*). In five out of nine systems the contributions of these maintainers exceed 60% of the total of the contributions.



(a) Number of maintainers

(b) Percentage of contributions by the top-5 maintainers

Figure 5.5. Top-5 maintainers analysis in noncompliant systems

To answer RQ #11, we sent out a survey to the 25 maintainers represented in Figure 5.5(b) and we obtained 11 answers (44%). Based on these answers we calculate the following score expressing the Effort to Maintain (EM) a system S:

$$\boxed{EM = M - L}$$

where M is the number of maintainers that answered that S is more difficult to maintain, when compared to the systems the respondent work with, and L is the number of maintainers that answered that it requires less effort to maintain.

Figure 5.6 shows the EM values for the nine noncompliant systems. Four out of nine systems require more effort to maintain ($EM > 0$). To illustrate this fact, we reproduce comments made by a Graphics developer:

"Graphics is a sum of patches over patches without a clear direction on design, with tons of duplicates and several design errors/conflicts. So is a pain to introduce any change there."

Figure 5.6 shows that three systems have a maintenance effort that is comparable to other systems our respondents work with ($EM = 0$). We also observe that $EM < 0$ for two systems. We hypothesize two reasons for these noncompliant systems require

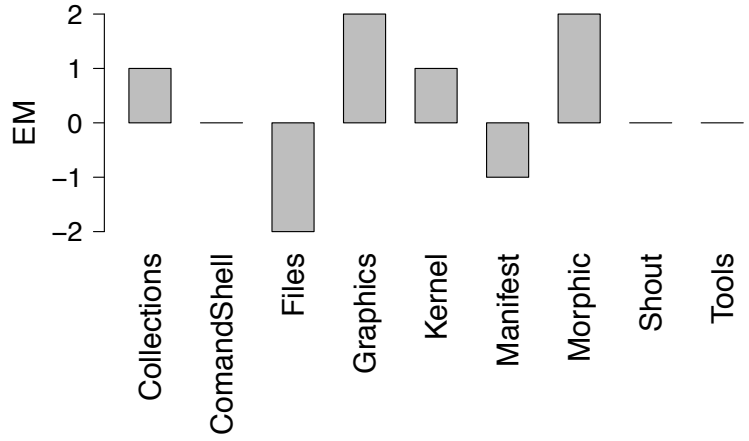


Figure 5.6. Effort to Maintain (EM)

less maintenance effort: (a) the metrics used to classify a system as noncompliant do not cover the whole spectrum of properties and requirements the maintainers considered when ranking systems in terms of internal quality; (b) maintainers are usually more wary when judging a system as presenting low quality, as we have learned when investigating the second research question.

Summary of findings: We found that four out of nine noncompliant systems are harder to maintain. Therefore, noncompliant systems are not largely viewed as requiring more effort to maintain than other systems.

5.3 Threats to Validity

In this section, we present possible threats to validity. First, our study participants might not be representative of the whole population of Pharo developers and, in more general terms, of general software developers. Anyway, we interviewed expert developers, with large experience and who are responsible for the central architectural decisions in their systems. Second, our *Corpus* and metric selections may not be representative to evaluate the quality of Pharo systems. However, we at least strive to include well-known and large Pharo systems in the *Corpus*. Moreover, the metrics used in this chapter cover important dimensions of a system implementation (size, coupling, and complexity).

5.4 Final Remarks

This chapter reported the results of an empirical study aimed at validating relative thresholds with professional developers. The study has been conducted with 79 Pharo systems and four source code metrics. The results indicate that well-designed systems mentioned by expert respect the relative thresholds. In contrast, we observed that developers usually have difficulties to indicate poorly-designed systems. We also found that four out of nine noncompliant systems are harder to maintain. Therefore, non-compliant systems are not largely viewed as requiring more effort to maintain than other systems.

Chapter 6

Conclusion

In this chapter, we summarize the outcome of this PhD thesis (Section 6.1). Next, we report on three recent works conducted by other authors that used our method to extract relative thresholds (Section 6.2). We also review our main contributions (Section 6.3). Finally, we present the further work (Section 6.4).

6.1 Summary

Source code metrics can be used to find possible problems or chances for improvements in software quality [34, 85]. A variety of metrics to measure source code properties like size, complexity, cohesion, and coupling have been proposed [1, 10, 22, 58, 61]. However, source code metrics are rarely used to support decision making because they are not easy to interpret [85, 95]. To promote the use of metrics as an effective measurement instrument, it is essential to establish credible thresholds [5, 35, 44, 92]. However, the definition of thresholds for source code metric values is not a trivial task, because these values usually follow heavy-tailed distributions [13, 64, 68, 86]. Therefore, in most systems it is "natural" to have source code entities not respecting the proposed thresholds for several reasons, including complex requirements, performance optimizations, etc.

To tackle this problem, we proposed and described an empirical method to derive relative thresholds from a *Corpus*. The proposed thresholds are relative because they should be valid for most but not for all entities in object-oriented systems. A relative thresholds is a pair $[p, k]$ such that at least $p\%$ of the classes should have $M \leq k$, where M is a given source code metric and p is the minimal percentage of classes in each system that should respect the upper limit k. Therefore, a relative threshold tolerates $(100 - p)\%$ of classes with $M > k$. We also designed a tool—called *RTTool*—that

implements our method and hence derives relative thresholds for metrics that follow heavy-tailed distributions.

We performed an extensive analysis of relative thresholds. Initially, we derive relative thresholds for six source code metrics, using the Qualitas Corpus. Next, we report seven studies using these thresholds. Specifically, we investigate whether 308 Java repositories, available at GitHub, follow the proposed relative thresholds (Section 4.4). We found that most popular GitHub repositories indeed follow our thresholds. We also compare the proposed relative thresholds with thresholds extracted using the SIG method [5] (Section 4.5). We concluded that both methods convey similar information. However, our method derives relative thresholds that can be automatically used to detect noncompliant systems. We evaluated the influence of the context in our results and we concluded that the impact on relative thresholds of context changes is limited (Section 4.6). We investigate how the proposed thresholds apply to different versions of the systems under analysis. In this study, we also investigate whether classes migrate from a compliant to a noncompliant state (or vice-versa) during their evolution (Section 4.7). We found that the proposed thresholds capture enduring design practices and we also fount that the percentage of classes with changes in their states tends to zero. We check the importance of classes that do not follow the upper limit of a relative threshold, by checking how often such classes are changed (Section 4.8). We found that classes that do not follow the upper limits are important in terms of maintenance activities. We investigated the relation between the presence of bad smells in a system and its adherence to the proposed relative thresholds (Section 4.9). We found that there is no evidence that noncompliant systems have a higher density of bad smells. Finally, we evaluated the dispersion of the metric values in the systems respecting the proposed thresholds, using Gini coefficients (Section 4.10). This inequality analysis showed that there are different distributions of methods per class among the systems that follow the proposed thresholds. However, such differences do not seem to have major impacts in terms of software quality.

Finally, we performed a study to validate our method to extract relative thresholds (Chapter 5). We extract thresholds from a *Corpus* with 79 Pharo/Smalltalk systems, which were validated with five Pharo experts and 25 Pharo developers. The results indicate that well-designed systems mentioned by expert often respect the relative thresholds. In contrast, developers usually have difficulties to indicate poorly-designed systems. We also found that four out of nine systems are harder to maintain.

6.2 Applications of Relative Thresholds

In this section, we report on three recent works conducted by other authors that used our method to extract relative thresholds. Section 6.2.1 report a comparison of methods to derive thresholds. Section 6.2.2 describes the use of the proposed method to derive thresholds for three annotation metrics. Section 6.2.3 presents experiences on using relative thresholds from performing software quality evaluations.

6.2.1 A Comparative Study on Metric Thresholds for Software Product Lines

A software product line (SPL) is a configurable set of systems that share a common, managed set of features in a particular market segment [66]. Features can be defined as modules with consistent, well-defined, independent, and combinable functions [6]. In this context, Vale *et al.* provide a comparison of methods to derive thresholds, using a *Corpus* of 33 software product lines [98]. They focus on three methods that consider the heavy-tailed distribution of source code metrics: SIG method [5], the method proposed by Ferreira *et al.* [35], and relative thresholds method. This study involved four main steps [98]:

- the authors built a benchmark of SPLs to explore the characteristics of each analyzed method. To build these benchmarks, they focus on SPLs developed using Feature-Oriented Programming (FOP) [11];
- they selected four metrics that capture different attributes of a SPL design, which are lines of code (LOC), coupling between Objects classes (CBO), weighted method per class (WMC), and number of constant refinements (NCR);
- they re-grouped the systems, based on their size, to compose two additional benchmarks and the three methods were used to derive thresholds for the four metrics in each of benchmarks;
- they verified whether the derived thresholds were appropriated, for example to detect God Classes.

The relative thresholds obtained are presented in the Table 6.1. The authors observed that there is a difference between the thresholds (among the benchmarks). The table shows that, in almost cases, the thresholds remained the same or have a slight growth. LOC was an exception, since presents a small decrease. The authors

claim that, this decrease is probably impacted by the penalties applied when deriving metric thresholds. Specifically, when the thresholds are used to identify code smells, our method outperforms SIG method both on precision (57% vs 47%, on average) and recall (90% vs 71%, on average).

Table 6.1. Relative thresholds derived by Vale *et al.*

Benchmark	LOC		CBO		WMC		NCR	
	p	*k*	*p*	*k*	*p*	*k*	*p*	*k*
1	55%	91	50%	6	70%	11	30%	1
2	80%	86	70%	13	80%	21	75%	2
3	75%	78	70%	13	80%	21	75%	2

6.2.2 Extracting Relative Thresholds for Feature Annotations Metrics

Feature annotations (*e.g.*, code fragments guarded by *ifdef* C-preprocessor directives) are widely used to control code extensions related to features [33, 59]. Feature annotations have long been said to be undesirable. For example, when maintaining features guarded by # *ifdefs*, there is a high risk of ripple effects. Also, excessive use of feature annotations may lead to code clutter, hinder program comprehension and harden maintenance [86]. To prevent such problems, developers should monitor the use of feature annotations, for example, by setting acceptable thresholds. However, little is known about how to extract such thresholds in practice, and which values are representative for feature-related metrics.

To address this issue, Queiroz *et al.* analyzed the statistical distribution of three feature-related metrics collected from a *Corpus* of 20 open source systems that use C preprocessor directives to annotate feature code [87]. The metrics they consider are [63]: (i) scattering degree of feature constants (SD); (ii) tangling degree of feature expressions (TD); and (iii) nesting depth of preprocessor annotations (ND). After collecting the metrics, the authors inspected the histograms and descriptive statistics describing the distributions of SD, TD, and ND for each system in the *Corpus*. Next, they performed a test for heavy-tailed distributions. They found 14 out of 20 systems show strong evidence that SD is heavy-tailed. They also found that TD and ND have a more uniform distribution for all subject systems, with most values equal to one. Then, the authors computed relative thresholds for SD metric, using the *RTTool* proposed in this thesis obtaining the following result:

85% of the feature constants in a system should have SD ≤ 6

According to Queiroz *et al.* all systems in the *Corpus*, with exception of *VI* and *SYLPHEED*, follow the proposed threshold for SD. *VI* and *SYLPHEED* exceed the threshold only marginally. In *VI*, they observed that 83% of the feature constants have SD ≤ 6 and, in *SYLPHEED*, this percentage was 82%. However, the proposed relative threshold holds for large and complex systems, such as the Linux Kernel, GCC, and MySQL. Figure 6.1 shows the percentile functions for the SD values of each system in the *Corpus*. The x-axis represents the percentiles and the y-axis represents the upper SD values of the feature constants matching the percentile. The plot nicely illustrates that SD values are heavy-tailed. However, there are two systems whose SD values start to grow earlier, around the 85th percentile, which are exactly *VI* and *SYLPHEED*.

The authors conclude that the proposed relative thresholds reflect the most common scattering distributions found in the *Corpus*. Moreover, they expect that different corpora would not produce radically different thresholds, because they selected a representative sample of C-preprocessor-based systems, including small, medium, and large systems.

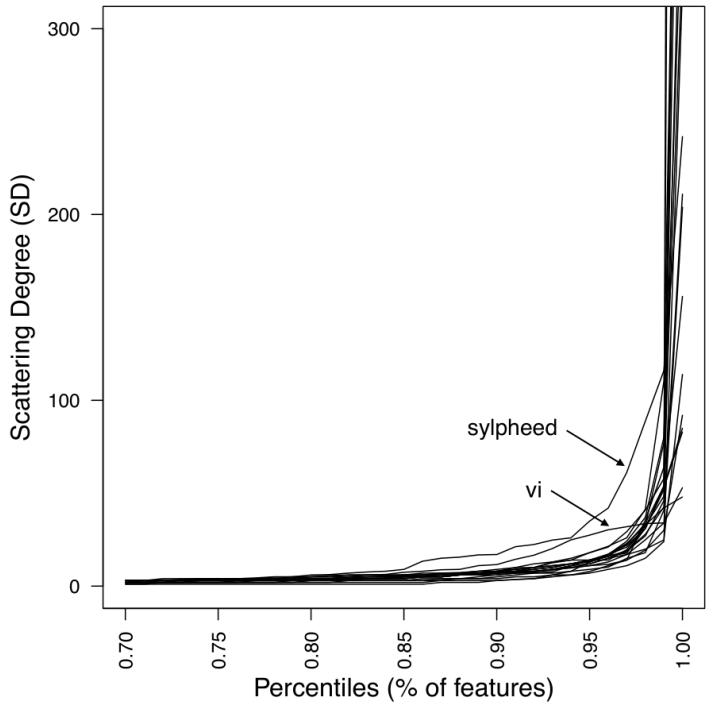


Figure 6.1. Percentile plots of scattering degrees (SD). Figure and caption originally used by Queiroz *et al.* [87]

6.2.3 Using Relative Thresholds in Industrial Context

Yamashita reports her experience on using relative thresholds in an international logistics company [107]. She describes a method and a mix-and-match of state of art research (*RTTool*), open source (*SchemaSpy*[1]) and industrial strength tools (*SonarQube*[2] and *NDepend*[3]), that can be used to perform system quality assessments. She first explores how *Corpus* from open source repositories can be created and used in order to provide more explicit and objective baselines for metrics based software quality evaluations. Next, she reinforces the advantages of benchmark-based methods (which provides a "factual and more neutral approach to assess the status quo of a system", according to one of the company employees). However, she also reports the challenges to define a curated *Corpus*, which does not include for example many libraries and testing code.

She mentioned that the evaluation assisted to disclose some development practices that in some areas were acceptable, but that in some areas was not aligned with the expectations of the company. She also reports that the study assisted the company to be in a better position for negotiating changes and improvements with the external contributors of the software.

6.3 Contributions

This PhD thesis makes five major contributions:

1. We provide a review of the state-of-the-art with respect to statistical properties of source code metrics and on methods to derive metrics thresholds (Chapter 2).

2. We introduce a novel method to derive source code metric thresholds based in a set of a systems (Chapter 3). This method derives *relative thresholds*, *i.e.*, pairs (p, k) such that $p\%$ of the classes should have $M \leq k$, where M is a source code metric and p is the minimal percentage of classes in each system that should respect the upper limit k.

3. We implemented a prototype tool called *RTTool* that implements our method. This tool is publicly available at `http://aserg.labsoft.dcc.ufmg.br/rttool` (Section 3.4).

[1] `http://schemaspy.sourceforge.net`, verified 11/11/2015.
[2] `http://www.sonarqube.org`, verified 11/11/2015.
[3] `https://www.ndepend.com`, verified 11/11/2015.

4. We evaluate the use of the proposed method in 106 real-world Java systems and in six source code metrics (Chapter 4).

5. We describe a validation study with expert developers, who are the right experts to check whether metric thresholds are indeed able to infer maintainability and design problems(Chapter 5). To the best of our knowledge, this is the first time that metric thresholds are validated with professional software developers.

6.4 Further Work

This work must be complemented by the following future work:

- By conducting in-depth interviews with at least some of the Pharo experts and maintainers considered in the study. These interviews will help to strength our findings (*e.g.*, to confirm that frameworks are usually noncompliants in terms of FAN-OUT) and also to clarify why some noncompliant systems are not perceived as being more difficult to maintain.

- By considering data from other sources, like mailing lists and bug tracking systems. These sources can help us to better asses the quality of both compliant and noncompliant systems.

- By evaluating the proposed method with other *Corpus*, possibly using the portfolio of a software development organization.

- By considering other software metrics, including non-source code based metrics, such as process metrics.

Bibliography

[1] Abreu, F. B. and Carapuça, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. In *4th International Conference of Software Quality (ICSQ)*, pages 3–5.

[2] Albert, R., Jeong, H., and Barabási, A. L. (1999). Diameter of the world-wide web. *Nature*, 401(6749):130–131.

[3] Alves, T. L. (2011). Categories of source code in industrial systems. In *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 335–338.

[4] Alves, T. L., Correia, J. P., and Visser, J. (2011). Benchmark-based aggregation of metrics to ratings. In *21th International Workshop on Software Measurement (IWSM)*, pages 20–29.

[5] Alves, T. L., Ypma, C., and Visser, J. (2010). Deriving metric thresholds from benchmark data. In *26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10.

[6] Apel, S., Kastner, C., and Lengauer, C. (2009). Featurehouse: Language-independent, automated software composition. In *31st International Conference on Software Engineering (ICSE)*, pages 221–231.

[7] Bacchelli, A. and Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering (ICSE)*, pages 712–721.

[8] Baggen, R., Correia, J. P., Schill, K., and Visser, J. (2012). Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307.

[9] Barabasi, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286:509–520.

[10] Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

[11] Batory, D., Sarvela, J. N., and Rauschmayer, A. (2003). Scaling step-wise refinement. In *25th International Conference on Software Engineering (ICSE)*, pages 187–197.

[12] Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. (2013). An empirical study on the developers' perception of software coupling. In *35th International Conference on Software Engineering (ICSE)*, pages 692–701.

[13] Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. (2006). Understanding the shape of Java software. In *21th Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 397–412.

[14] Benlarbi, S., Emam, K. E., Goel, N., and Rai, S. (2000). Thresholds for object-oriented measures. In *11th International Symposium on Software Reliability Engineering (ISSRE)*, pages 24–38.

[15] Bouwers, E., Visser, J., and van Deursen, A. (2012). Getting what you measure. *Communications of the ACM*, 55(7):54–59.

[16] Brown, W. J., Malveau, R. C., McCormick, H. W. S., and Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1 edition.

[17] Buse, R. and Weimer, W. (2010). Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558.

[18] California Institute of Technology (2010). JPL institutional coding standard for the Java programming language. Technical report.

[19] Campbell, G. A., Papapetrou, P. P., and Gaudin, O. (2013). *SonarQube in Action*. Manning Publications Company.

[20] Catal, C., Alan, O., and Balkan, K. (2011). Class noise detection based on software metrics and ROC curves. *Information Sciences*, 181(21):4867–4877.

[21] Chidamber, S. and Kemerer, C. (1991). Towards a metrics suite for object oriented design. In *6th Conference proceedings on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 197–211.

[22] Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[23] Chrissis, M., Konrad, M., and Shrum, S. (2006). *CMMI: Guidelines for Process Integration and Product Improvement*. The SEI Series in Software Engineering. Addison-Wesley, 2 edition.

[24] Clauset, A., Shalizi, C. R., and Newman, M. E. J. (2009). Power-law distributions in empirical data. *Physics*, 51(4):661–703.

[25] Coleman, D. M., Lowther, B., and Oman, P. W. (1995). The application of software maintainability models in industrial software systems. *Journal of Systems and Software*, 29(1):3–16.

[26] Concas, G., Marchesi, M., Pinna, S., and Serra, N. (2007). Power-Laws in a Large Object-Oriented Software System. *IEEE Transactions on Software Engineering*, 33(10):687–708.

[27] Correia, J. P., Kanellopoulos, Y., and Visser, J. (2009). A survey-based study of the mapping of system properties to ISO/IEC 9126 maintainability characteristics. In *25th IEEE International Conference on Software Maintenance (ICSM)*, pages 61–70.

[28] Couto, C., Maffort, C., Garcia, R., and Valente, M. T. (2013). COMETS: a dataset for empirical research on software evolution using source code metrics and time series analysis. *Software Engineering Notes*, 38(1):1–3.

[29] Cunningham, W. (1992). The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30.

[30] da Silva, B. C., Sant'Anna, C. N., and Chavez, C. v. F. (2014). An empirical study on how developers reason about module cohesion. In *13th International Conference on Modularity (MODULARITY)*, pages 121–132.

[31] Dybå, T., Sjøberg, D. I., and Cruzes, D. S. (2012). What works for whom, where, when, and why?: On the role of context in empirical software engineering. In *6th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 19–28.

[32] Erni, K. and Lewerentz, C. (1996). Applying design-metrics to object-oriented frameworks. In *3rd IEEE International Software Metrics Symposium (METRICS)*, pages 64–74.

[33] Favre, J.-M. (1996). Preprocessors from an abstract point of view. In *18st IEEE International Conference on Software Maintenance (ICSM)*, pages 329–339.

[34] Fenton, N. E. and Neil, M. (2000). Software metrics: roadmap. In *22th International Conference on Software Engineering (ICSE)*, pages 357–370.

[35] Ferreira, K., Bigonha, M., Bigonha, R., Mendes, L., and Almeida, H. (2011). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257.

[36] FilÃş, T. G. S., Bigonha, M. A. S., and Ferreira, K. A. M. (2015). A catalogue of thresholds for object-oriented software metrics. In *First International Conference on Advances and Trends in Software Engineering (SOFTENG)*, pages 48–55.

[37] Foss, S., Korshunov, D., and Zachary, S. (2011). *An Introduction to Heavy-Tailed and Subexponential Distributions*. Springer-Verlag.

[38] Foucault, M., Palyart, M., Falleri, J.-R., and Blanc, X. (2014). Computing contextual metric thresholds. In *29th ACM Symposium On Applied Computing (SAC)*, pages 1–10.

[39] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

[40] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

[41] Gao, Y., Xu, G., Yang, Y., Niu, X., and Guo, S. (2010). Empirical analysis of software coupling networks in object-oriented software systems. In *1th International Conference on Software Engineering and Service Sciences (ICSESS)*, pages 178–181.

[42] Heitlager, I., Kuipers, T., and Visser, J. (2007). A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 30–39.

[43] Henderson-Sellers, B. (1996). *Object-oriented Metrics: Measures of Complexity*. Prentice-Hall.

[44] Herbold, S., Grabowski, J., and Waack, S. (2011). Calculation and optimization of thresholds for sets of software metrics. *Journal of Empirical Software Engineering*, 16(6):812–841.

[45] Humphrey, W. S. (1995). *A Discipline for Software Engineering*. Addison-Wesley, 1 edition.

[46] Ichii, M., Matsushita, M., and Inoue, K. (2008). An exploration of power-law in use-relation of java software systems. In *19th Australian Conference on Software Engineering (ASWEC)*, pages 422–431.

[47] IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. Technical report.

[48] ISO/IEC (2001a). International standard ISO/IEC 14598: International standard for software engineering-product evaluation.

[49] ISO/IEC (2001b). International standard ISO/IEC tr 9126: Software engineering - product quality.

[50] ISO/IEC (2003). International standard ISO/IEC 15504: Information technology - process assessment.

[51] ISO/IEC (2005). International standard ISO/IEC 9000:2005 - quality management systems - fundamentals and vocabulary.

[52] ISO/IEC (2008). International standard ISO/IEC 25000:software quality requirements and evaluation standard family.

[53] Jing, L., Keqing, H., Yutao, M., and Rong, P. (2006). Scale free in software metrics. In *30th Annual International Conference on Computer Software and Applications (COMPSAC)*, volume 1, pages 229–235.

[54] Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2 edition.

[55] Kaner, C., Falk, J. L., and Nguyen, H. Q. (1999). *Testing Computer Software*. John Wiley & Sons, 2nd edition.

[56] Katzmarski, B. and Koschke, R. (2012). Program complexity metrics and programmer opinions. In *20th International Conference on Program Comprehension (ICPC)*, pages 17–26.

[57] Kerzazi, N., Khomh, F., and Adams, B. (2014). Why do automated builds break? an empirical study. In *30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–50.

[58] Kitchenham, B. (2009). What is up with software metrics?-A preliminary mapping study. *Journal of Systems and Software*, 83(1):37–51.

[59] Krone, M. and Snelting, G. (1994). On the inference of configuration structures from source code. In *16th IEEE International Conference on Software Maintenance (ICSM)*, pages 49–57.

[60] Landman, D., Serebrenik, A., and Vinju, J. J. (2014). Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods. In *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 221–230.

[61] Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

[62] Lehman, M. M. (1996). Blaws of software evolution revisited. In *5th European Workshop on Software Process Technology (EWSPT)*, pages 108–124.

[63] Liebig, J., Kästner, C., and Apel, S. (2011). Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Tenth International Conference on Aspect-oriented Software Development (AOSD)*, pages 191–202.

[64] Lin, Z. and Whitehead, J. (2015). Why power laws? an explanation from fine-grained code changes. In *The 12th Working Conference on Mining Software Repositories (MSR)*, pages 01–08.

[65] Lincke, R., Lundberg, J., and Löwe, W. (2008). Comparing software metrics tools. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 131–142.

[66] Loesch, F. and Ploedereder, E. (2007). Restructuring variability in software product lines using concept analysis of product configurations. In *11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 159–170.

[67] Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall.

[68] Louridas, P., Spinellis, D., and Vlachos, V. (2008). Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–26.

[69] Luijten, B. and Visser, J. (2010). Faster defect resolution with higher technical quality of software. In *4th International Workshop on Software Quality and Maintainability (IWSQM)*, pages 1–10.

[70] Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM)*, pages 350–359.

[71] Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):1–9.

[72] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

[73] Meyer, B. (2000). *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition.

[74] Mordal, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., and Ducasse, S. (2013). Practical software quality metrics aggregation. *Software Maintenance and Evolution: Research and Practice*, pages 1–19.

[75] Nejmeh, B. A. (1988). Npath: A measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200.

[76] Newman, M. E. J. (2005). Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46(5):223–351.

[77] Nierstrasz, O., Ducasse, S., and Gĭrba, T. (2005). The story of Moose: an agile reengineering environment. *Software Engineering Notes*, 30(5):1–10.

[78] Oliveira, P., Borges, H., Valente, M. T., and Costa, H. (2013). Metrics-based detection of similar software. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 447–450.

[79] Oliveira, P., Lima, F., Valente, M. T., and Serebrenik, A. (2014a). RTTool: Extracting relative thresholds for source code metrics. In *30th International Conference on Software Maintenance and Evolution, (ICSME)*, pages 629–632.

[80] Oliveira, P., Valente, M. T., Bergel, A., and Serebrenik, A. (2015a). Validating metric thresholds with developers: an early result. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–5.

[81] Oliveira, P., Valente, M. T., and Lima, F. (2014b). Extracting relative thresholds for source code metrics. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 254–263.

[82] Oliveira, P., Valente, M. T., and Serebrenik, A. (2015b). Corpus-based derivation of relative metrics thresholds. *Journal of Systems and Software*, pages 1–34.

[83] Oliveira, P. M., Borges, H. S., Valente, M. T., and Costa, H. A. X. (2012). Uma abordagem para verificaï£¡ï£¡o de similaridade entre sistemas orientados a objetos. In *XI Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 1–15.

[84] Potanin, A., Noble, J., Frean, M., and Biddle, R. (2005). Scale-free geometry in OO programs. *Communications of the ACM*, 48:99–103.

[85] Pressman, R. S. (2009). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science, 7 edition.

[86] Queiroz, R., Passos, L., Valente, M. T., Apel, S., and Czarnecki, K. (2014). Does feature scattering follow power-law distributions? an investigation of five pre-processor-based systems. In *6th International Workshop on Feature-Oriented Software Development (FOSD)*, pages 23–29.

[87] Queiroz, R., Passos, L., Valente, M. T., Hunsen, C., Apel, S., and Czarnecki, K. (2015). The shape of feature code: An analysis of twenty C-preprocessor-based systems. *Journal on Software and Systems Modeling*, 1(1):1–20.

[88] Rosenberg, L. H. (1998). Applying and interpreting object oriented metrics. In *10th Annual Software Technology Conference (STC)*.

[89] SEI (2006). CMMI for development, version 1.2. Technical report, Software Engineering Institute.

[90] Serebrenik, A., Roubtsov, S. A., and van den Brand, M. (2009). Dn-based architecture assessment of java open source software systems. In *17th IEEE International Conference on Program Comprehension (ICPC)*, pages 198–207.

[91] Serebrenik, A. and van den Brand, M. G. J. (2010). Theil index for aggregation of software metrics values. In *26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–9.

[92] Shatnawi, R. (2015). Deriving metrics thresholds using log transformation. *Journal of Software: Evolution and Process*, 27(2):95–113.

[93] Shatnawi, R., Li, W., Swain, J., and Newman, T. (2010). Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(1):1–16.

[94] SOFTEX (2009). *MPS.BR - Melhoria de Processo do Software Brasileiro - Guia Geral v1.2*.

[95] Sommerville, I. (2010). *Software Engineering*. Addison-Wesley, 9 edition.

[96] Taube-Schock, C., Walker, R., and Witten, I. (2011). Can we avoid high coupling? In *25th European Conference on Object-Oriented Programming (ECOOP)*, pages 204–228.

[97] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 336–345.

[98] Vale, G., Albuquerque, D., Figueiredo, E., and Garcia, A. (2015). Defining metric thresholds for software product lines: A comparative study. In *19th International Conference on Software Product Line (SPLC)*, pages 176–185.

[99] van den Brand, M. G. J., Roubtsov, S. A., and Serebrenik, A. (2009). SQuAVisiT: A flexible tool for visual software analytics. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 331–332.

[100] Vasa, R., Lumpe, M., Branchand, P., and Nierstrasz, O. (2009). Comparative analysis of evolving software systems using the Gini coefficient. In *25th IEEE International Conference on Software Maintenance (ICSM)*, pages 179–188.

[101] Vasa, R., Schneider, J. G., and Nierstrasz, O. (2007). The inevitable stability of software change. In *23rd IEEE International Conference on Software Maintenance (ICSM)*, pages 4–13.

[102] Vasilescu, B., Serebrenik, A., and van den Brand, M. (2011). You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 313–322.

[103] Walker, R. J., Rawal, S., and Sillito, J. (2012). Do crosscutting concerns cause modularity problems? In *20th International Symposium on the Foundations of Software Engineering (ACM SIGSOFT)*, pages 1–11.

[104] Weibull, W. (1951). A statistical distribution function of wide applicability. *Journal of Applied Mechanics*, 18:293–297.

[105] Wheeldon, R. and Counsell, S. (2003). Power law distributions in class relationships. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 45–54.

[106] Whitmire, S. A. (1997). *Object-Orientend Design Measurement*. John Wiley & Sons, 1 edition.

[107] Yamashita, A. (2015). Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 421–428.

[108] Yoon, K.-A., Kwon, O.-S., and Bae, D.-H. (2007). An approach to outlier detection of software measurement data using the k-means clustering method. In *1st Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 443–445.

[109] Zhang, F., Mockus, A., Zou, Y., Khomh, F., and Hassan, A. E. (2013). How does context affect the distribution of software maintainability metrics? In *29th International Conference on Software Maintainability (ICSM)*, pages 1–10.

[110] Zipf, G. K. (1949). *Human Behavior and the Principle of Least Effort*. Addison-Wesley.