Journal of the Brazilian Computer Society, 2024, 30:1, doi: 10.5753/jbcs.2024.3373

Evaluating Thresholds for Object-Oriented Software Metrics

Tarcísio G. S. Filó [**Federal University of Minas Gerais** | ***tarcisio.filo2@gmail.com***]
Mariza A. S. Bigonha [**Federal University of Minas Gerais** | ***mariza@dcc.ufmg.br***]
Kecia A. M. Ferreira [**Federal Center for Technological Education of Minas Gerais** | ***kecia@cefetmg.br***]

Federal University of Minas Gerais,Instituto de Ciências Exatas (ICEx), Departamento de Ciência da Computação, Av. Antônio Carlos, 6627, Pampulha, 31.270-901, Belo Horizonte, MG, Brazil.

Received:02 May 2023 • **Accepted:** 08 June 2024 • **Published:** 25 September 2024

Abstract *Software metrics* measure quantifiable or countable software characteristics. Researchers may apply them to provide better product understanding, evaluate the process effectiveness, and improve the software quality. A *threshold* is a value that aids the proper interpretation of software measurements; it indicates whether or not a given value represents a quality risk. Thresholds are unknown for most software metrics, inhibiting their use in a software quality assessment process. In a previous paper, we proposed a catalog with 18 object-oriented software metrics thresholds, providing a preliminary case study in proprietary software to validate them. This article evaluates these thresholds more deeply, considering significant aspects. We show a new example of threshold derivation, discussing it qualitatively. We explain these software metrics and discuss their threshold values, presenting each one's application level, definition, formula, and implications for the software design. We conduct a study with two software systems to evaluate the capacity of our thresholds to identify software quality enhancement after a restructuring process. We assess these thresholds using two case studies, comparing the evaluation provided by the thresholds with the qualitative analysis given by manual inspections. The study results indicate that the thresholds may lead to few false-positive and false-negative occurrences, i.e., the thresholds provide a proper quantitative assessment of software quality. This study contributes with empirical evidence that the metrics' thresholds proposed in our previous work provide a proper interpretation of software metrics and, hence, may aid the application of software metrics in practice.

Keywords: Object-Oriented Software Metrics, Measurement, Thresholds Catalog, Internal Software Quality, Software Engineering

1 Introduction

According to Fenton [1994], "measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules". A software metric is a clearly defined rule that assigns values to software entities, such as components, classes, methods, or attributes of development processes. Object-oriented metrics were proposed in the early 1990s and are widely used by researchers in proposals to evaluate software systems [Chidamber and Kemerer, 1994; Mishra Alok *et al.*, 2021].

Metrics provide quantitative measures to evaluate the quality of projects, allowing software engineers to make changes throughout the development process that will increase the quality of the final product. Each metric measures either an internal or an external attribute. Someone can measure internal attributes by observing only the process, product, or resource without considering its behavior. External attributes are attributes that are related to the behavior of software systems. In this work, the focus is on internal product metrics that measure source code. Some examples of internal attributes that relate to source code are size, cohesion, coupling, and inheritance. The evaluation of software quality via measurements aids in assessing qualitative attributes quantitatively. Metrics are essential in managing the quality of object-oriented software; moreover, their practical use still needs to appear in the software systems [Ferreira *et al.*, 2012; Kitchenham, 2010; Mishra Alok *et al.*, 2021; Radjenović *et al.*, 2013].

Despite their importance in software engineering, OO metrics still need to be adequately applied in the software industry. The threshold values of most metrics need to be proposed and validated since software metrics, without their proper interpretation, may be worthless. Therefore, identifying and evaluating threshold values for software metrics is vital for their application. An appropriate software metric threshold value is an orientation for the developer in maintenance, testing, and evolution tasks.

In our previous work, [Filó *et al.*, 2015], we defined thresholds for 18 OO metrics. We preliminary evaluated these thresholds using a case study with proprietary software. In that study, we compared the results of a manual evaluation performed by systems' development team with the results of the evaluation of the system by applying the thresholds. The results suggest that our catalog is an efficient way to evaluate the classes effectively. We also evaluated the correlation between the thresholds and the occurrences of bad smells in the system. For this purpose, we used JDeodorant to identify the bad smells. The results of this analysis also indicate that our proposed thresholds were useful in identifying bad smells in the software system we considered. However, our analysis was limited to a single software system. As Mishra Alok *et al*. [2021] concluded in a mapping study on software metrics thresholds, "even nowadays, many studies still need to validate metric threshold values regarding quality attributes. And still, there is a need to study a more extensive number of metrics and their thresholds."

In the present work, we aim to contribute to advancement in software metrics thresholds evaluation by evaluating a catalog with the threshold for 18 metrics of object-oriented software we proposed previously [Filó *et al*., 2015], which, to our knowledge, is the most extensive number of threshold metrics existing so far. With thresholds cataloged and validated, metrics can be effectively applied in the software industry, allowing the evaluation of software quality automatically. Another possible scenario is using thresholds to evaluate the internal software quality, allowing decision-making in software acquisition processes.

With this in mind, our overall objective described in this paper is to verify those thresholds' ability to assess the software systems' internal quality. We aim to provide evidence of their benefits for object-oriented metrics at class, method, and package levels, aiding the quantitative evaluation of internal software quality.

To fulfill our overall goal, we present a deeper evaluation of these thresholds in seven significant aspects: (1) we give a detailed description of our method to derive the 18 software metrics considered in this research and discuss their proposed thresholds. For each metric, we provide in Appendix A its application-level -*method, class,* or *package*- its definition, formula, and design implications. (2) We exhibit a new illustrative example of our method application, describing the data analysis of the *McCabe Complexity (VG)* metric and a qualitative discussion about it. We conduct a study with two software systems to evaluate the capacity of our thresholds to identify software quality enhancement after a restructuring process. (4) We evaluate these threshold metrics with two new case studies using an extensive dataset of 111 systems also applied in our previous studies on software metrics Terra *et al*. [2013]. These cases compare the software systems' quantitative analysis with the qualitative analysis of manual inspections. Case Study 1 investigates whether the thresholds may detect poor-quality methods and classes. Case Study 2 evaluates whether the thresholds may see high-quality classes and methods. (5) We show that the empirical evaluation of the results of our thresholds indicates that applying them will lead to a few false-positive and false-negative occurrences and, then, may aid a proper quantitative assessment of the software quality. (6) we describe each method's qualitative evaluation, and (7) we present the qualitative evaluation regarding each class. Items (6) and (7) both refer to Case Study 1. To perform our study, we formulated two general-purpose research questions **RQ1** and **RQ2** and proposed two specific ones, **RQ2.1** and **RQ2.2**.

RQ1. *Does refactoring improves the metric thresholds?*
Rationale. With this research question, we investigate whether the thresholds are sensitive to the changes in the system's structure before and after refactoring. As a refactoring process usually aims to improve the internal software quality, it is expected that after the refactoring process, more classes fit in the thresholds' intervals, which correspond to good internal quality (Section 7).

RQ2. *Are the proposed thresholds able to differentiate poor and good software quality?*
Rationale. We investigated this question in two different ways, according to the following specific research questions (Section 8).

RQ2.1. *Can the proposed thresholds detect poor-quality methods and classes?*
Rationale. To answer RQ2.1, we conducted Case Study 1, which consists of comparing the results of a qualitative evaluation given by a manual inspection of methods and classes, with the results of the quantitative evaluation provided by the threshold (Section 8.1).

RQ2.2 *Can the proposed thresholds detect high-quality methods and classes?*
Rationale. To answer RQ2.2, we have conducted Case Study 2 to verify how the proposed thresholds behave in the evaluation of a well-designed software system (Section 8.2).

We organized the remainder of this paper as follows. Section 2 shows how our work relates to existing literature efforts. Section 3 presents the dataset, the method applied to collect the metric data, and the dataset preparation for defining the thresholds. For completeness, Section 4 briefly presents our previous work, presenting the method applied to extract the thresholds, its preliminary evaluation, and exhibits the proposed thresholds catalog. Section 5 shows our method used for the VG metric, illustrating a new example and discussing its threshold derivation main aspects. Section 6 discusses the technique used to derive the thresholds conducted in this research. Section 7 shows an experiment to evaluate the thresholds proposed for package metrics as software quality indicators in actual object-oriented restructuring processes. Section 8 presents two case studies to assess the proposed thresholds. Section 9 discusses the threats to the validity of this study. Section 10 shows this research's possible future directions and presents the final remarks. Appendix A presents a detailed description of each metric and its thresholds. Appendices B and C, respectively, give the qualitative evaluation of each method (Section 8.1.1) and each class (Section 8.1.2), both considered in Case Study 1. Appendix D presents the metrics' description.

2 Related Work

Deriving software metrics thresholds is an active research field [Mishra Alok *et al*., 2021]. This section discusses the most relevant works that aimed to derive software metrics thresholds by applying statistical properties. Section 2.1 shows the most used metrics by the researchers, and Section 2.2 exhibits some recent studies regarding metrics and thresholds.

2.1 Object-Oriented Metrics

The software metrics may be classified into traditional and object-oriented. Our work focuses on object-oriented metrics. Several metrics with different categories are suggested in the literature to evaluate OO software systems. Li and Henry [1993] proposed the following metrics: MPC (number of send statements defined in a class), DAC (number of ADTs defined in a class), NOM (number of local methods), SIZE1 (number of semicolons in a class), and SIZE2 (number of attributes + number of local methods). Chidamber and Kemerer [1994] proposed six CK metrics to measure the complexity of object-oriented software: NOC (Number of Children), DIT (Depth of Inheritance Tree), CBO (Coupling between Objects), WMC (Weight Method per Class), LCOM (Lack of Cohesion in Methods), and RFC (Response for a class). Li [1998] proposed metrics for class inheritance and method complexity. NAC (Number of Ancestor Classes), NLM (Number of Local Methods), CMC (Class Method Complexity), NDC (Number of Descendent Classes), CTA (Coupling Through Abstract Data Type), and CTM (Coupling Through Message Passing). Lorenz and Kidd [1994] defined eleven metrics to calculate the static aspects of software design at the class and method level, considering size: NPM, NM, NPV NV, NCV, NCM; inheritance: NMI, NMO, NNA; and class internal: APM and SIX. Among these, the most frequently used are the CK metrics. Our catalog of thresholds considers 18 of these software metrics defined in the literature.

2.2 Object-Oriented Metrics Thresholds

Over the years, researchers have been investigating software metrics in a wide range of approaches, and in many cases, they used software metrics to assess software evolution and quality. Also, the studies have shown methods to derive, identify, or evaluate software metrics thresholds for various categories of O.O. metrics. "Metric Threshold" refers to a limit for the metric value that may demand an action or change in the software system. For example, considering the metric MLOC (Method Lines of Code), Filó *et al.* [2015] identified ten as the threshold considered good and 30 as considered regular. This threshold means that a method should have up to ten lines preferably, or up to 30 lines, i.e., if a method has more than 30 lines, the developer should verify the need for refactoring the method. In tables 1 and 2, the column "Threshold" exhibits how the previous work has defined these limits for software metrics.

Mishra Alok *et al.* [2021] state software metrics' thresholds might be classified into programmer experience, statistical properties, and quality-related properties. Considering this classification, we exhibit examples of the many available research studies.

Statistical properties. Many works derive thresholds by analyzing statistical properties of the software metrics gathered from a corpus of software systems. Some examples of works in this category: Alves *et al.* [2010], Aniche *et al.* [2016], Benlarbi *et al.* [2000], Ferreira *et al.* [2012], Filó *et al.* [2015], Gil and Lalouche [2016], Herbold *et al.* [2011], Lochmann [2012], Veado *et al.* [2016], Malhotra and Bansal [2015], Mei *et al.* [2023], Mori [2018], Oliveira *et al.* [2014b], Padhy *et al.* [2021], Shatnawi [2010], Shatnawi [2015], Shatnawi [2018], Shatnawi and Althebyan [2013], Singh and Kaur [2014], Stojkovski [2017], Sultan [2021], Vale *et al.* [2015], Vale *et al.* [2019], Vale and Figueiredo [2015].

Programmer experience. Some works derive software metric thresholds by analyzing the relation between the metric values and aspects related to the programmer experience, e.g., software degradation and occurrences of bad smells in the source code. Some examples in this category: Beranič and Heričko [2017]; Fan *et al.* [2020]; Sodiya *et al.* [2012].

Quality related. This category may derive a threshold by observing the relationship between the software metric values and quality attributes. According to Mishra Alok *et al.* [2021], researchers have addressed different quality attributes for threshold calculation. Some examples of works in this category: Al Dallal [2011], Al Dallal [2012], Al Dallal and Briand [2010], Arar and Ayan [2016], Bigonha *et al.* [2019], Boucher and Badri [2016], Boucher and Mourad [2018], Fontana *et al.* [2015], Hussain *et al.* [2016], Lavazza and Morasca [2016], Malhotra and Bansal [2015], Malhotra and Bansal [2017], Mohammed *et al.* [2019], Mohović *et al.* [2018], Oliveira *et al.* [2014a], Shatnawi *et al.* [2009], Vale *et al.* [2015]. Mishra Alok *et al.* [2021] divided quality attributes of threshold calculation methods into five categories: fault detection, bad smell detection, reuse proneness, and other design problems. These quality attributes are described as follows.

Fault detection refers to the ability of a method or approach to identify potential faults or defects in software code. Detecting faults early in the development process is important for improving software reliability and reducing the cost of fixing defects later on. Examples of works in this category includes: Arar and Ayan [2016], Bigonha *et al.* [2019], Boucher and Badri [2016], Boucher and Mourad [2018], Hussain *et al.* [2016], Lavazza and Morasca [2016], Malhotra and Bansal [2015], Malhotra and Bansal [2017], Mohović *et al.* [2018], Mori [2018], Oliveira *et al.* [2014a], Rosenberg *et al.* [1999], Shatnawi [2010], Shatnawi [2015], Shatnawi [2018], Shatnawi *et al.* [2009], Vale *et al.* [2015].

Bad smells are specific structures in the code that may indicate deeper problems. They are usually not bugs but indicators of poor design or implementation choices. Detecting and addressing them improves code maintainability, readability, and extensibility. Examples of works in this category: Al Dallal [2012], Bigonha *et al.* [2019], Feng *et al.* [2019], Filó *et al.* [2015], Fontana *et al.* [2015], Sousa *et al.* [2017], Souza *et al.* [2017], Vale *et al.* [2019], Vale and Figueiredo [2015].

Reuse proneness refers to how likely a software artifact, such as a method, class, or module, will be reused in other parts of the software or other projects. Promoting it helps to reduce code duplication, improve consistency across the software, and speed up development by leveraging existing, tested components. An example in this category is: Padhy *et al.* [2021].

Design problems encompasses other aspects of software design that are critical for its quality but do not directly fall under fault detection, bad smell detection, or reuse-proneness. These could include performance bottlenecks due to poor design decisions or issues related to scalability and security, etc. Examples of works in this category are: Al Dallal [2011], Al Dallal and Briand [2010], Benlarbi *et al.* [2000], Herbold *et al.* [2011], Malhotra and Bansal [2015], Malhotra and Bansal [2017], Shatnawi and Althebyan [2013].

Considering the categorization of Mishra Alok *et al.* [2021], our work is classified as statistical. In the sequel, we discuss some of the main works that also applied statistical methods to derive software metrics thresholds.

Benlarbi *et al.* [2000] empirically derived thresholds for the CK metrics, without LCOM, by applying logistic regression and considering two telecommunication systems written in C++ to investigate the faults and software metrics connection.

Alves *et al.* [2010] proposed a method to identify metric thresholds from a statistical analysis of a benchmark of software systems. They showed thresholds for three metrics at the method level and two at the class level. They used four percentiles to define ranges of values to the metric thresholds: low 70%, moderate 80%, high 90%, and very high >90%.

Shatnawi [2010] used a statistical model derived from the logistic regression to identify threshold values for the CK metrics. He analyzed three versions of Eclipse software using the error data from Bugzilla. The author divided the errors into Low, Medium, and High categories. He identified a risk level for any arbitrary threshold value.

Al Dallal [2011] proposed values for 12 cohesion metrics to classes that exhibit exceptional cases, such as having no attributes, no parameter types for the methods, or fewer than two methods in the class. He performed an empirical analysis using classes in four open-source systems from Java. However, their study did not consider factors that could affect the study's results, such as inheritance.

Herbold *et al.* [2011] defined an approach to calculate thresholds for a software metric set to evaluate quality attributes. Their approach is purely data-driven and uses machine learning and data mining techniques. Considering that a single metric cannot cover all internal attributes such as structure, size, and complexity, they applied 11 metrics, four for methods and seven for classes, and did four case studies.

Ferreira *et al.* [2012] used the graphical analysis of software metric distributions to derive the thresholds. In their method, when the metric has a distribution with an expected average value, like the *Poisson* distribution, this value is taken for that metric; otherwise, they proposed three ranges for the metric values: *Good*, *Regular*, and *Bad*. The *Good* corresponds to values with high frequency. The *Bad* refers to values with relatively low frequency, and the *Regular* refers to values that are not too frequent nor have a shallow frequency. They applied this method to 40 Java software systems and defined thresholds for six software metrics. They analyzed the entire dataset's software system sizes, application domains, and type of software and found no significant differences in the suggested thresholds among these applications.

Lochmann [2012] used a benchmark approach where the calculation of the threshold value for a metric is based on the metric values of a set of systems, the benchmark base. To see if its use influences the software quality assessment result, he conducted a quality assessment of a series of test systems for each. He found that considering a randomly generated benchmark base of sufficient size, neither the selected systems nor the size of the systems had a significant influence.

Shatnawi and Althebyan [2013] proposed a statistical assessment of the behavior of software systems conducted on metrics for five systems divided into two contexts, four systems of different sizes, and twelve releases resulting from the evolution of an open-source system. They found out that the six metrics showed a potential to follow a power law distribution, and two metrics did not follow a power law distribution.

Oliveira *et al.* [2014b] derived thresholds for eight metrics at the class level. They used the concept of relative thresholds for evaluating metric data that follows a heavy-tailed distribution, where the values in the "long tail" correspond to the range. This concept advocates that most source code entities within a software system should follow a metric threshold, although some "long tail" entities may not follow them.

Singh and Kaur [2014] proposed a study to derive a threshold metrics value against the bad smell using risk analysis at five different levels, seven metrics, and three versions of Firefox as a dataset to validate their study. Results showed the practical threshold values only for five metrics for predicting the smelly classes in the three releases of Mozilla Firefox. They also found that the threshold values derived to predict smelly classes do not indicate whether the class is faulty.

Abílio *et al.* [2015] conducted exploratory research using the work of Lanza and Marinescu [2010] and derived the threshold for eight SPLs developed with AHEAD and 26 participants. They want to propose means to detect three bad smells in FOP (Feature-Oriented Programming) source code.

Shatnawi [2015] proposed data transformation to improve two techniques of software quality assessment: derive threshold values and fault classification, using the derived CK metrics to reduce the effect of skewness in data, and 11 different open-source Java projects. He used the natural log function to transform all metrics. The threshold values are then derived for all log metrics. To evaluate the effectiveness of the results after transformation, He used the derived thresholds to classify classes into either faulty or not faulty otherwise.

Arar and Ayan [2016] replicated Shatnawi [2010]'s study but expanded it to 37 versions from ten open-source software systems as datasets and 20 metrics. To investigate whether or not there are effective threshold values for each metric that support a defining risk categorization of each module, they defined three risk categories: non-fault-prone, fault-prone, and three-or-more-fault-prone, and then executed two case studies to derive two thresholds that support the specified categories of modules. Using ten datasets as training data by the model, they created a single set by merging logistic regression and a previous technique proposed by Bender [1999]. Seven metrics gave good results in the first case study. In the second, 11 metrics gave acceptable results.

Malhotra and Bansal [2017] made an empirical study to identify metric threshold values using the Receiver Operating Characteristic (ROC) Curves for fifteen metrics. They used five releases of an open-source, Linux-based OS, Android, written in Java, calculated threshold values to identify change-prone classes for those releases, and validated them on their immediate successor release. They compared their study with the traditional methodology based on logistic regression.

Bigonha *et al.* [2019] evaluated the applicability of the software metric thresholds of Filó *et al.* [2015] in the identification of the bad smells in Java-based projects: Large Class, Long Method, Data Class, Feature Envy, and Refused Bequest. Results indicate that these thresholds support the prediction of software faults and effectively detect bad smells.

Sultan [2021] conducted an empirical study using 36 defect-prediction datasets and six metrics, whose objective was to build predictive models that estimate thresholds based on system size and assess this approach's feasibility as a threshold estimation method. The size–threshold relationships are then examined by analyzing their correlation for each metric. If such a relationship exists for a metric, a linear regression model is fitted and used to estimate its thresholds given only system sizes. The feasibility of this approach is then assessed based on the accuracy with which the estimated thresholds identify fault-prone classes.

Mei *et al.* [2023] empirically examined whether there are practical threshold values of OO metrics by analyzing threshold derivation methods with large-scale meta-analysis. Based on five representative threshold derivation methods and 3268 releases from 65 Java projects, the authors first employed statistical meta-analysis and sensitivity analysis techniques to derive thresholds for 62 OO metrics on the training data. Then, they investigated the predictive performance of five candidate thresholds for each metric on the validation data to explore which candidate thresholds served as the threshold. Finally, they evaluated their predictive performance on the test data. The experimental results showed that 26 of 62 metrics have the threshold effect, and the derived thresholds by meta-analysis achieved promising results in almost all five representative baseline thresholds.

To automate the threshold identification process or extract the metrics, the tool support is necessary. Oliveira *et al.* [2014a] described RTTOOL, an open-source tool capable of extracting relative thresholds for software metrics based on benchmark collections. The authors illustrated their usage by deriving relative thresholds for four metrics based on 106 open-source Java systems from the Qualitas Corpus E. Tempero and Noble [2010]. Sousa *et al.* [2017] proposed FindSmells, a tool for bad smell detection in software systems based on software metrics and their thresholds. Veado *et al.* [2016] proposed and implemented the Threshold Derivation Tool (TDTool), an open-source tool to provide threshold derivation for software metrics using four different methods: Ferreira *et al.* [2012]; Alves *et al.* [2010]; Oliveira *et al.* [2014b] and Vale and Figueiredo [2015].

Our preliminary study [Filó *et al.*, 2015] is preceded by others [Abílio *et al.*, 2015] [Al Dallal, 2011] [Alves *et al.*, 2010] [Benlarbi *et al.*, 2000] [Ferreira *et al.*, 2012] [Herbold *et al.*, 2011] [Lochmann, 2012] [Oliveira *et al.*, 2014b] [Shatnawi, 2010] [Shatnawi, 2015] [Shatnawi and Althebyan, 2013] [Singh and Kaur, 2014]. However, our previous work differs from most of its precedents in many aspects. The work of Alves *et al.* [2010] is similar to ours; however, they fixed percentiles that only sometimes work for all metrics, and the validity of their technique is done merely by visual inspection of plots instead of rigorous statistical tests. The CK metrics are the most reported in these studies. Most methods are not reproducible, and they have used only a few datasets to validate their thresholds. The method we applied to derive the thresholds in our previous work [Filó *et al.*, 2015] is based on the method proposed by [Ferreira *et al.*, 2012]. However, we applied a rigorous statistical approach and based the derived thresholds on analyzing a more extensive and diverse dataset, 111. We also used 18 metrics and derived thresholds for all of them.

Comparing the recent studies with our present work, we found that some are similar to our first approach Filó *et al.* [2015] in certain aspects, while others are not. Furthermore, similar to our recent and previous work, most works presented in this paper took the work of Ferreira *et al.* [2012] as a baseline method. Ferreira *et al.* [2012] applied their method to 40 software systems and defined thresholds for six software metrics. Aniche *et al.* [2016] proposed SATT (Software Architecture Tailored Thresholds), an approach that detects whether an architectural role is considerably different from others in the system regarding code metrics. They concluded that their approach tends to return doubtful results for architectural roles with metric value distributions significantly different when compared with other classes. Stojkovski [2017] proposed thresholds for software quality metrics in open-source Android projects. However, he derived thresholds for only five metrics, four of which are from the CK suite, and he still needs to validate thresholds on quality attributes.

Mohammed *et al.* [2019] compared their method for extracting thresholds with the one used by Ferreira *et al.* [2012] and concluded that they are close in the range for each category. Compared with our present work, even though they developed a web application to extract the threshold easily, there is no guarantee that it does not have errors. They considered only the class level of the 16 open-source Java-based projects. Besides that, they claim that they performed statistical and comparative analyses to evaluate and validate the effectiveness of the derived thresholds, but they used only three metrics. Another example is the work of Beranič and Heričko [2019], who adopted the technique described by Ferreira *et al.* [2012] in their study. They presented the results of an empirical study comparing systematically obtained threshold values for nine software metrics in four OO programming languages: Java, C++, C#, and Python. The authors claim that the definitions of software metrics present a limitation within the research and that they need to validate the software metrics in their study. Hussain *et al.* [2016] followed the [Bender, 1999] approach and proposed a model to derive the threshold values to 13 metrics. Still, they stated that their proposed model only applies to derive significant metrics thresholds for some studied systems. Lavazza and Morasca [2016], Morasca and Lavazza [2016], and Morasca and Lavazza [2017] proposed and evaluated metric thresholds based on fault-proneness, whereas Filó *et al.* [2015] and Ferreira *et al.* [2012] methods are based in benchmark data, so, ours. Lavazza and Morasca [2016] investigated the consequences of defining thresholds on internal measures without considering the external measures that quantify qualities of practical interest. They found a set of statistically significant fault-proneness models for measured 11 metrics but did not report them explicitly in their study. Malhotra and Bansal [2017], in their work, confirmed the derivation and validation of thresholds on only five versions of the Android operating system. The validity of Mohović *et al.* [2018] research is vital because they used a robust standard sampling and statistical analysis approach. Still, their conclusions are based on a small-scale case study, limiting their external validity.

Mishra Alok *et al.* [2021] concluded in their Systematic Mapping Study that, even nowadays, many studies still need to validate metric threshold values regarding quality attributes. Still, there is a need to study a more extensive number of metrics and their thresholds.

Even though previous studies propose different techniques to derive thresholds for software metrics, most cover only a few metrics, except [Arar and Ayan, 2016] and [Mei *et al.*, 2023]. Nevertheless, in Arar and Ayan [2016], only seven metrics gave satisfactory results in their first case study and eleven in the second case study. In the end, they have 18 threshold values like us. Besides that, they defined thresholds only for modules and used only ten datasets as training data for the model. We used a larger sample with software systems from different companies and institutions. Instead, our approach considered methods, classes, packages, a dataset of 111 systems, and 18 metrics and evaluated all the metrics thresholds. Mei *et al.* [2023] obtained their metrics by the Perl script; however, given that some OO cohesion metrics are not defined in the absence of methods in a class, information was lost, so the authors excluded classes in which the OO metric contains undefined values. Besides that, they also have problems on the prediction performance results on the test datasets.

Highlighting existing methods and tools to derive thresholds, Mori [2018] proposed a method to derive domain-sensitive thresholds that respect metric statistics based on benchmarks of systems from the same domain with the support of a software tool. Nevertheless, he does not consider the intrinsic characteristics of software systems in each domain or provide an analysis of thresholds for software domains. They ignore, for instance, that systems from different environments may have various degrees of complexity and size. Besides, they restricted his research results due to the fewer metrics selected, only eight. Even though they have presented a large-size study, it is necessary to do additional replications to see if researchers can generalize their findings to other domains and datasets. Another example is the work of Vale *et al.* [2019], who proposed a benchmark-based approach, Vale's method, for deriving thresholds for eight metrics with the help of a TDTtool [Veado *et al.*, 2016]. However, they evaluated their method using a 103 Java open-source software system benchmark.

In our preliminary work [Filó *et al.*, 2015], we do not propose a new method to derive the thresholds. Instead, we improved the empirical method defined by Ferreira *et al.* [2012], as in most of the works shown in this paper. Unlike the other works, we assessed the thresholds in a proprietary software system, evaluating the outside of the open-source universe.

In the present work, described in this article, when we compare the contributions of our whole study with the results of Ferreira *et al.* [2012] and all other studies, we may spot more than four significant differences. (1) We evaluated the 18 object-oriented software metrics threshold, a large number of thresholds compared with previous and recent studies. (2) The proposed thresholds provide a benchmark for *quantitatively* evaluating the software systems' internal quality, considering metrics at the class, method, and package levels, while most of the related works described in this paper considered only the class level on the open-source Java-based projects. (3) To evaluate the thresholds for object-oriented package metrics, we experimented to verify these values' ability to assess the quality improvement of actual software restructuring processes. (4) We provide a *qualitative* discussion about the proposed thresholds, presenting two case studies to assess the thresholds' ability to evaluate the internal quality of classes and methods.

Tables 1 and 2, appearing on the next page, summarize some works, according to the categories exhibited in Section 2.2, highlighting the statistical categories.

3 Data Collection

The software metric thresholds proposed in our research are based on the metrics values found in real data. While defining the thresholds, we considered each software metric's statistical distribution characteristics. This section describes Qualitas.*class* Corpus [E. Tempero and Noble, 2010], the set of systems used, the data preparation, and the statistical data generation of the metrics necessary for this research. Qualitas.*class* Corpus provides compiled Eclipse Java projects for the 111 systems included in Qualitas Corpus [Terra *et al.*, 2013], and the aim of Qualitas.*class* Corpus is to assist researchers by removing the compilation effort when conducting empirical studies. It contains over 18 million LOC, 200,000 compiled classes, and 1.5 million compiled methods. Qualitas.*class* Corpus relies on Metrics 1.3.8 [Metrics, 2014]. It provides XML files with values for the 18 metrics, distributed as follows. (1) *Basic metrics* - Number of Classes (NOC), Number of Methods (NOM), Number of Fields (NOF), Number of Overridden Methods (NORM), Number of Parameters (PAR), Number of Static Methods (NSM), and Number of Static Fields (NSF). (2) *Complexity metrics* - Method Lines of Code (MLOC), Specialization Index (SIX), Normalized Distance (RMD), McCabe Cyclomatic Complexity (VG), and Nested Block Depth (NBD). (3) *CK metrics* - Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NSC), and Lack of Cohesion in Methods (LCOM). And (4) *Coupling metrics* - Afferent/Efferent Coupling (AC/EC).

We developed a tool, RafTool [Filó *et al.*, 2014], using JAXB[1] to read the available XML files at Qualitas.*class* Corpus. JAXB provides an easy way to read XML documents, doing the *unmarshall* process to convert the XML document to a tree of Java objects. RafTool then generates text files containing all the measurements for each metric and populates a *MySql* database with all the measures and identifications of the following artifacts: method, class, or package. We used R[2] to generate the cumulative relative frequency graph, which summarizes the frequency below a given level and generates the histogram in the logarithmic. To aid researchers with software metrics works, we generated a statistical dataset of object-oriented software metrics Filó *et al.* [2014], which also provides a MySQL database dump that establishes the measures for the metrics and facilitates the task of data manipulation, such as filtering and aggregation.

[1] http://docs.oracle.com/javase/tutorial/jaxb
[2] https://www.r-project.org/

Table 1. Related Work - some basic information, software metrics, and corresponding thresholds (Adapted from Beranič and Heričko [2017])

Papers	**Prog. Language**	**Purpose of metric derivation**	**Threshold calculation method**	**Metrics**	**Threshold**	**Number of software projects used as benchmark**
Benlarbi et al.[2000]	C++	Fault prediction	Cognitive theory	CK metrics, without LCOM	Yes (numeric)	2
Alves et al.[2010]	Java, C#	Measurement data concerning statistical prop. metrics	Distribution-based statistical methods	LOC, MCCabe, NOM, FAN-IN,NPM	Yes (risk intervals low, moderate, high, very high	100
Shatnawi [2010]	Java	Error risk level	Statistical logistic regression	CK metrics	Yes (numeric)	1 Eclipse Proj. V.2.0 e V.2.1
AlDallal [2011]	Java	Predicting faulty classes	Logistic regression data	TCC, LCC, CC, DCi, SCOM, CAMC, DCd, LCOM1, LCOM2, LCOM5, NHD, Coh	Yes (interval min, mean, med, max)	4
Herbold et al.[2011]	Java clas. C++, C# methods, C func.	Evaluate quality attribute	Driven machine learning	NFC, NORM VG, WMC, CBO, NOM, NST, LOC, NBD, RFC, NSM	Yes (numeric)	4 studies on open source soft. metric sets
Ferreira et al.[2012]	Java	Detecting design flaws	Statistical analysis of data	LCOM, DIT, COF, AC, NPM, NPF	Yes (bad, regular, good)	40
Lochmann [2012]	Java	Quality assessment	Benchmarking approach	CBO, RFC, WMC, LCOM, DIT, NOC	Yes (numeric)	2041
Shatnawi et al.[2013]	Java	Fault-proneness models	Power law distribution	NOC, WMC, NOM, CBO, NOV, RFC, SLOC	Yes (numeric)	5 open-source systems
Oliveira et al[2014a]	Java	Internal quality	Analysis of a software corpus	NOM, LOC, NOA, LCOM, FAN-OUT	Yes (numeric)	106
Sing et al.[2014]	C++	Bad smell	Risk analysis logistic regression	CK metrics, NOA, NOOM, NOAM, EncF, Puf, Co,	Yes (low, medium high)	3 versions Mozilla Firefox
Abílio et al.[2015]	AHEAD	Detect code smells in Soft. Product Lines (SPL)	categorization and detection of smells in AHEAD-based	NOF, NCR, NMR, TNCt, TNR, TNMR, TNRC, TNRM	Yes (low, average, high)	8 AHEAD systems
Shatnawi [2015]	Java	Reduce data skewness	Log transformation	CK metrics	Yes (minimum maximum median, mean)	11 products 41 releases
Filó et al.[2015]	Java	Cumulative relative frequency graph	Graphical analysis of software metric distribution	AC, CE, MLOC, NOF NOM, NSM, VG, SIX WMC, LCOM, NOC, DIT, PAR, RMD, NBD	Yes (interval bad-uncommon regular/casual good/common)	111 software systems
Arar et al.[2016]	Java	Fault prediction	Statistical logistic regression	WMC, CBO, RFC, LCOM, LOC, NPM, AMC, MAX CC, AVG CC, CA, CE	Yes (Interval min, max, standard)	10
Lavazza et al.[2016]	Java	Fault prone	Distribution-based methods	WMC, DIT, CBO, CAM, AMC, LOC, RFC, CA, CE, maxCC, avgCC	Yes (interval tref: refer to Fault-proneness value, tmin 16% tmax 84%)	Datasets from PROMISE repository
Stojkovski [2017]	Java	Internal quality	Visual analysis of frequency chart, tables	NOM, RFC, CBO, NOC, DIT	Yes (numeric)	2 Source Forge and Android

Table 2. Related Work - some basic information, software metrics, and corresponding thresholds (Adapted from Beranič and Heričko [2017])

Papers	**Prog. Language**	**Purpose of metric derivation**	**Threshold calculation method**	**Metrics**	**Threshold**	**Number of software projects used as benchmark**
Malhotra et al.[2017]	Java	Change-prone classes	Receiver operating characteristic curves	CK metrics: LOC, NOM, NIM, NIV, NIV, NOA, NPM, NPROM, NPRM	Yes (Numeric)	5 releases of open source from ANDROID
Mori [2018]	Java	Assess the quality of software syst. from	Benchmarking approach	LOC, NOA, NOM, WMC, LCOM, CBO, DIT, NOC	Yes (Interval very low, low moderate high, very high)	3,107 soft. syst. from 15 desktop domains
Mohovic et al.[2018]	Java	Code fault prone	Binary univariate logistic regression model	CK metrics	Yes (numeric)	5 releases of 2 open-source dataset of Eclipse proj.
Mohammed et al.[2019]	Java	Internal quality	Machine learning based framework	LCOM, CBO CFC, LOC	Yes (standard, deviation, mean)	16 open-source projects
Bigonha et al.[2019]	Java	Bad smells detection, and Fault-prone	Evaluate the usefulness of the Filó 18 thresholds	DIT,NOF,NSF,SIX, NOM, NORM, NSC, NSM, LCOM, WMC	Yes (interval bad, regular, good)	12 systems
Sutan et al.[2019]	Java	Investigate the relation-ship between system size & threshold	Predictive small models to estimate threshold based sys. size	CBO, WMC, DCC, NOM, Export coupling, Import coupling	Yes (numeric)	36 defect-prediction dataset of dif. version of 13 open source syst.
Vale et al.[2019]	AHEAD	To compose detection strategies for 2 code smells	Benchmark-based method	CK metrics, CK metrics, LOC, NCR	Yes (very low, low, moderate, high, very high)	103 open source soft. system
Mei et al.[2023]	Java	Defect-proneness analysis of classes	Statistical meta analysis technique	LCOM1 - LCOM4, Co, NewCo, NewLCOM5, LCC, ICH, OCC, PCC, DCd, DCi, CAMC, NHD, SNHD, ACAIC, ACMIC, AMMIC, DMMEC, OCAEC, OCAIC, OCMEC, OCMIC, OMMEC, CBI, CBO DAC, ICP, MPC, IHICP, NIHICP, RFC, AID, CLD, DIT, DP, DPA, DPD NMA, NMI, NMO, NOA, NOC, NOD, NOP, SIX, SP, SPA, SPD, NA NMIMP, SLOC NumPara, Stmts	Yes (numeric)	65 project

Table 3. Catalog of Thresholds (extracted from Filó *et al.* [2015])

Metrics	*Good/Common*	*Regular/Casual*	*Bad/Uncommon*
AC	$m \leq 7$	$7 < m \leq 39$	$m > 39$
EC	$m \leq 6$	$6 < m \leq 16$	$m > 16$
DIT	$m \leq 2$	$2 < m \leq 4$	$m > 4$
LCOM	$m \leq 0.167$	$0.167 < m \leq 0.725$	$m > 0.725$
MLOC	$m \leq 10$	$10 < m \leq 30$	$m > 30$
NBD	$m \leq 1$	$1 < m \leq 3$	$m > 3$
NOC	$m \leq 11$	$11 < m \leq 28$	$m > 28$
NOF	$m \leq 3$	$3 < m \leq 8$	$m > 8$
NOM	$m \leq 6$	$6 < m \leq 14$	$m > 14$
NORM	$m \leq 2$	$2 < m \leq 4$	$m > 4$
NSC	$m \leq 1$	$1 < m \leq 3$	$m > 3$
NSF	$m \leq 1$	$1 < m \leq 5$	$m > 5$
NSM	$m \leq 1$	$1 < m \leq 3$	$m > 3$
PAR	$m \leq 2$	$2 < m \leq 4$	$m > 4$
RMD	$m \leq 0.467$	$0.467 < m \leq 0.750$	$m > 0.750$
SIX	$m \leq 0.019$	$0.019 < m \leq 1.333$	$m > 1.333$
VG	$m \leq 2$	$2 < m \leq 4$	$m > 4$
WMC	$m \leq 11$	$11 < m \leq 34$	$m > 34$

4 Thresholds Identification

For completeness, this section briefly describes how we identified the software metrics thresholds and made their first evaluation in the previous work, Filó *et al.* [2015].

In the Ferreira *et al.* [2012] work (Section 2), when a metric has a distribution with an expected average value, as the *Poisson* distribution, this value is taken as typical for that metric; otherwise, they identified three ranges for the metric values: *Good*, *Regular*, and *Bad*. The authors used the graphical analysis of software metric distributions to derive the thresholds. We proposed two improvements to this method in Filó *et al.* [2015], (1) we modified the ranges names to *Good/Common*, *Regular/Casual*, and *Bad/Uncommon* to highlight the importance of the frequency concept in the suggested thresholds. (2) We established two percentiles rather than the values based on a visual analysis of the graphical views and the threshold's frequency concept. These percentiles separate the metric values on the three ranges of values mentioned. We found that using predefined percentiles improves the method since it allows obtaining the values directly from the data set, making the method application more reproducible. To illustrate that, we provided illustrative examples of the application of our approach for two metrics: *Number of Methods (NOM)* and *Depth of Inheritance Tree (DIT)*.

4.1 The Proposed Catalog of Thresholds

Besides our improvements in the Ferreira *et al.* [2012] work, our previous work proposed a catalog with thresholds for 18 software metrics and briefly described the method used to derive them. Table 3 exhibits the thresholds catalog for each metric analyzed in this study. Our catalog reflects a pattern followed by most of the software systems in Qualitas.*class* Corpus, which may be helpful in some scenarios of Software Engineering.

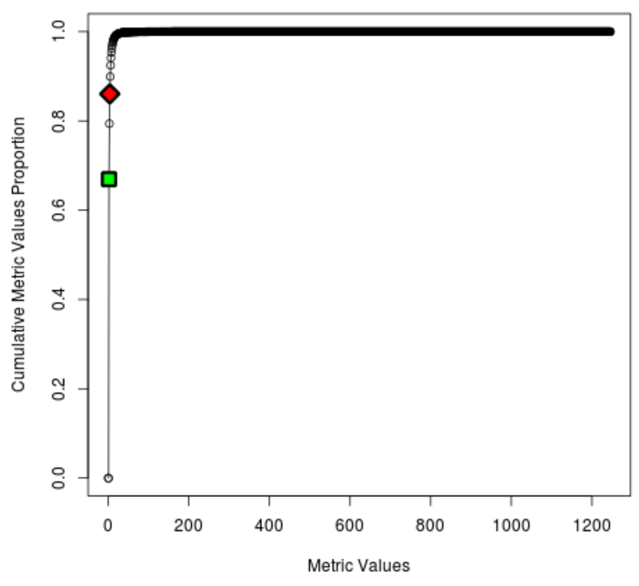
This paper details each metric and its thresholds in Appendix A.

4.2 Case Study of Proprietary Software with Bad Internal Quality

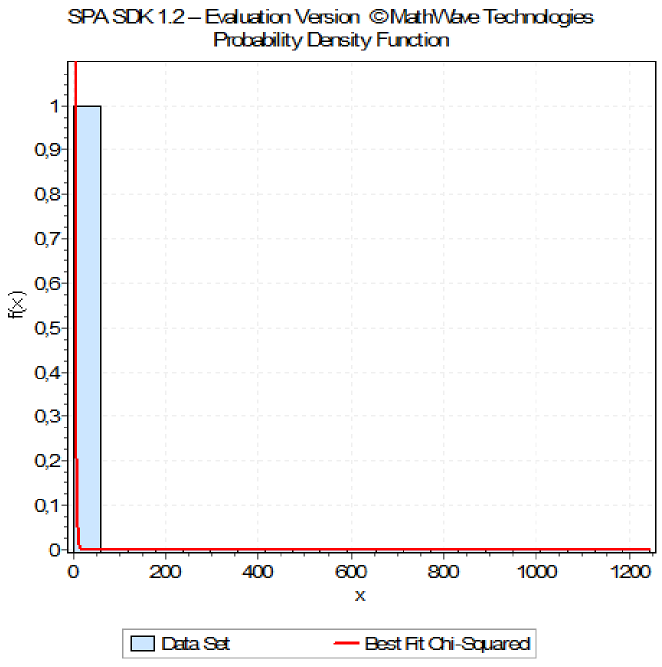
To evaluate our catalog, in that work, we handled a case study to assess proprietary software from a public organization with a bad internal quality to verify the proposed thresholds' ability to indicate it. We divided this study into three items to assess the metrics of class and methods and the correlation of bad smell occurrence with our threshold evaluation. In such analysis, we identified fewer methods evaluated as *Good/Common* and a high number estimated as *Bad/Uncommon* compared with most of the systems in the Qualitas.*class* Corpus. This result agrees with the established qualitative design about the low quality of this system, and the thresholds for method metrics reflected, quantitatively, the scenario of the low quality of this system. Regarding the class evaluation, we considered a set of eight poor-quality classes. They all had at least three classifications out of range *Good/Common*. Based on this evaluation, we concluded that we should not use a single metric of our catalog to define class quality. This conclusion is consistent with the work of Rosenberg *et al.* [1999]. Therefore, the results of our previous work suggest that our catalog is an efficient way to evaluate the methods and classes effectively. It will not show quality where there are problems. More essentially, it suggests the utility of the proposed thresholds outside the open-source universe [Filó *et al.*, 2015].

5 New Threshold Derivation Example

Section 5 shows our method applied to the *McCabe Complexity (VG)* metric, illustrating a new example of threshold derivation and discussing its main aspects. Figure 1 shows in



(a)



(b)

Figure 1. VG: a Cumulative Relative Frequency Graph b *pdf* fitted to the Chi-Squared

a the Cumulative Relative Frequency Graph and in b the *pdf* fitted to the Chi-Squared. Figure 1 a suggests a heavy-tail distribution for the VG metric because the approximation of 100% o cumulative relative frequency along the x-axis, metric values, occurs in a drastically faster way, i.e., there is a nearly instantaneous approximation of 100% of the measures. This fact means that there are many methods with few independent paths in their execution graph and a few methods with a large number of independent paths in their execution graph.

Figure 1 b shows that the data set of VG is best-fitted by *Chi-Squared* distribution, with parameters $\nu = 1.000$ and $\gamma = 1.000$. This distribution has heavy-tail characteristics. If this is the case, we may not use the sample mean and variance as estimators of the population because basing any conclusion on sample means without fully understanding the distribution would be misleading. According to [Baxter *et al.*, 2006] the mean value is not representative.

Figure 2 exhibits the data set on a *log-log* scale. This graph shows a straight leaning to the left, a power-law feature [Ferreira *et al.*, 2012; Baxter *et al.*, 2006]. This pattern enhances the features already mentioned; most classes have few methods, and the mean is not representative. We chose the 70° and 90° percentiles by a visual analysis of the Cumulative Relative Frequency Graph shown in Figure 1 a. The points marked with green/square shape and red/diamond shape represent the regions identified in this analysis. Furthermore, the choice of these percentiles is also based on the concepts of *Good/Common*, *Regular/Casual* and *Bad/Uncommon* ranges since our thresholds reflect the standard that has commonly applied in software development. Thus: (1) based on visual analysis, (2) on the established concept for the ranges, and (3) inspired by Alves *et al.* [2010] –who used percentiles to statistically part quality metric profiles in the identification

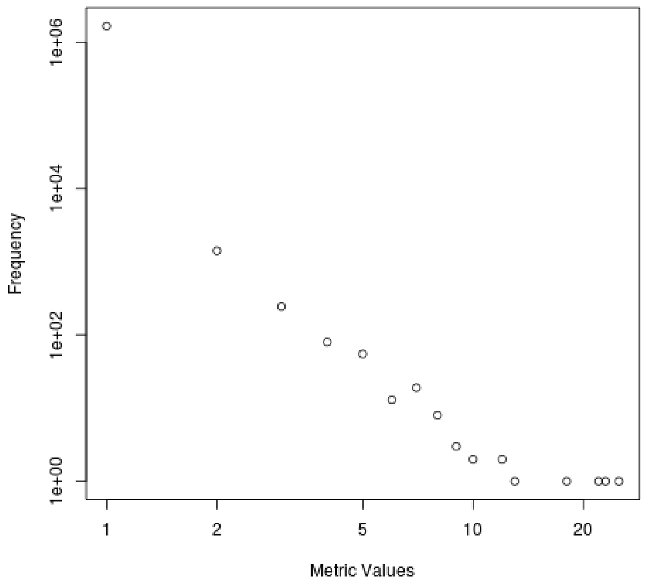


Figure 2. Histogram *log-log* scale.

of thresholds– we tried to apply 70° and 90° percentiles in most of the metrics to identify the measures to separate the three suggested ranges, which correspond to the values 2 and 4.

We observed variations that follow the distribution curve features when they are taller or flattened or depending on a higher or lower metric value (x- axis) to reach a higher cumulative frequency, in which the 70° and 90° percentiles do not have significance. In such cases, we relied mainly on the visual analysis and distribution features to identify the regions that may separate the three suggested ranges. The value two is at the 70° percentile. So, two is equal to or greater than 70% of the measures in the dataset. The value four is at the 90° percentile. Then, four is equal to or greater than 90%

of the values. Therefore, 2 and 4 allow us to separate the VG metric in: *Good/Common* ($VG \leq 2$), *Regular/Casual* ($2 < VG \leq 4$), and *Bad/Uncommon* ($VG > 4$).

6 Discussion About the Technique Used to Derive the Thresholds

The proposed thresholds allow identifying methods, classes, and packages with anomalous measures compared to the quality pattern commonly applied in software development. Identifying anomalous measurements may aid software quality assessment, even though they do not necessarily mean a problem, but it suggests that there might be a problem with the artifact's structure. This section discusses the main aspects of deriving the thresholds in this research.

6.1 Treatment of Metric Values

Bouwers and Van [2012] say that conducting software changes to improve metric values is a purely "cosmetic" activity. Value handling leads to refactorings that "please the metrics", which severely wastes resources. To illustrate this situation, they consider a project in which a method is identified as having too many parameters; this may indicate that the method is implementing several features. If this method is split into smaller ones so that each resulting method implements a single functionality, it will make the program easier to understand. A second problem that may be occurring with this method is the need for an object that groups parameters commonly used together. For instance, consider a method that receives as a parameter two objects of type Date, *startDate* and *endDate*. These names suggest that these two parameters may form an object of type *DatePeriod*, in which *startDate* must precede *endDate*. When multiple methods receive them as input, the introduction of the object *DatePeriod* into the domain may be of great value to the quality of the project.

These two situations exemplify that quantitative evaluation may imply a real improvement in the project, making it more readable and easier to maintain. It reaches the organization's goal of having higher-quality software. However, the programmer may directly treat the value of a metric. For instance, he may move the method parameters to class attributes or even replace them with an object of type *Map*, which is a list of elements of type *key-value*, where *key* represents the parameter with its value, and *value* represents its value.

All those strategies may reduce the number of parameters of the methods. However, the ultimate goal is to improve code readability and minimize future maintenance efforts. In this case, solutions that address symptom management rather than the root causes of the problem will not effectively enhance the quality of the software. This situation appears because the developers may need to learn that there are more important goals when applying the metrics to manage the internal software quality.

Looking at the practical application of the proposed threshold catalog, it is not intended that tools that use them give developers results in absolute numbers. It is essential that the programmer adequately interprets the method, class, or package as having structural problems according to a given quality attribute mapped by the metric. Moreover, the programmer must know the quality goal sought by that control [Bouwers and Van, 2012]. Using an ordinal variable that reflects the qualitative classification allows the programmer to be guided not by a value but by seeking quality. The value should not be the ultimate goal; it should be a way to reach the ultimate goal, which is to increase the software quality.

6.2 Unidirectionality of the Proposed Thresholds in Relation Quality

The approach of this work is unidirectional concerning the quality since it assumes a clear quality orientation for lower metric values. This presumption exists. However, observe that this problem starts with the metrics definition itself. All the metrics, even those with no thresholds, have a more precise definition regarding deteriorated quality for high values than for low ones. A classic example of this ”pattern” adopted is where the higher the metric value, the worse it is, as in the metric LCOM. A class must have high cohesion. However, to maintain this sense of orientation that high values are not good, the metric is defined as a lack of cohesion [Chidamber and Kemerer, 1994]. The questions in the sequel are, in fact, much more problematic in both definitions -metrics and thresholds- What is the minimum coupling value a package should have? What is the minimum number of attributes a class should have? Moreover, What is the minimum number of class methods?

As such, *Good/Common* range reflects what is consensual in quality because it represents the practice in software development. Of course, there are problems related to this definition, but they are much more related to how to apply the threshold catalog than their values. An example of this scenario is the DIT metric, which has a clear orientation that high values characterize a deep inheritance hierarchy, meaning that methods will be inherited and the class will become more complex. In addition, large distances between child classes and the root class in the inheritance hierarchy characterize a more complicated project and, therefore, prone to errors. This orientation is evident because inheritance trees should not be deep. In the metric definition, Chidamber and Kemerer [1994] suggest that many classes with very small DIT look like a ”top-heavy” project, indicating that the project may not be taking advantage of reusing methods via inheritance. However, the authors of the DIT metric say that this may occur in function of the application and, itself, is a conclusion that transcends the metrics' responsibility in evaluating the inheritance tree depth of the class and, consequently, the threshold proposed for it.

In this scenario, if all system classes have $DIT = 0$, they would be classified as *Good/Common*. However, the project would be "top-heavy," suggesting a sub-optimal use of inheritance. This assumption is probably correct, but it contradicts what is proposed as a threshold for DIT, as shown by the good results in this example. This conclusion emerges from a general observation of the software design and has no relation to the Filó et al. threshold having classified classes as Good/Common. A class with a shallow inheritance tree

($DIT = 0$) does not display the problems listed by DIT authors when defining this metric. One might think that the issue of all classes being well evaluated is related to the threshold definition. After all, the project does not use inheritance properly. This fact would be a problem in understanding the metrics definition and the application of this catalog; this happens because none of the metrics for which we proposed the threshold evaluate the use of the corresponding resource within the software. These considerations highlight issues related to using the thresholds catalog and interpreting their results. The scenarios to which they should be applied must be evaluated by the user to minimize misinterpretation.

6.3 Sensitivity Analysis of the Proposed Method to Derive Thresholds to the Corpus Choice

Ferreira *et al.* [2012] conducted the sensitivity evaluation of the method used to derive the thresholds regarding the Qualitas Corpus [E. Tempero and Noble, 2010] with a high level of sensitivity in the method definition itself. They applied the method across the systems and grouped it by application, domain, and type. In doing so, they do not identify relevant differences between these approaches' thresholds. Instead, considering this research's primary goal, evaluating the thresholds catalog for object-oriented software metrics, we applied our method to the whole dataset considered in this work.

6.4 Implementation of the Metrics

The metrics analyzed in this research were collected using the Metrics plugin. Differences in the implementation of software metrics due to different interpretations between the extraction tools are common due to the need for specific definitions. Implementation details are present on the plugin Metrics[3].

7 Thresholds Evaluation in Real Restructuring Processes

This experiment aims to investigate RQ1: *Does refactoring improves the metric thresholds?*. To Anquetil and Laval [2011], identified thresholds should detect improvements in a software quality system. Therefore, we evaluate if we may use Filó's et al. metrics thresholds at the *package level* as software quality indicators in actual object-oriented restructuring processes, i.e., if the metrics concerning a package change its range in Filó's et al. metrics thresholds when a refactoring is performed in the package. We analyzed the following software metrics: NOC (number of children, AC (afferent coupling), EC (efferent coupling), and RMD (normalized distance). We considered these metrics because they may be impacted when changes of classes within the packages occur, e.g., including/excluding classes in/from a package.

[3]http://metrics2.sourceforge.net/

7.1 Study Design

In this section, we describe the method we applied to the experiment. First, we justify the criterion when selecting the project subjects for analysis (Section 7.1.1).

7.1.1 Projets Selection

The context of this experiment is restructured object-oriented software packages; therefore, we defined two ideal situations regarding the samples.

- The restructuring process must be a genuine effort and of limited duration [Anquetil and Laval, 2011]. This allows for identifying versions before and after the restructuring once they do not get lost in other natural changes during software evolution. However, this is hard to find because real systems need to evolve.
- We derived the identified thresholds from measurements gathered with the *Metrics* plugin. This fact imposes one restriction: projects must be able to be compiled in *Eclipse* so that the values collected for the experiment on the target systems were collected in the same way that the values used to define the metrics were collected.

This experiment used *JHotDraw* and *Eclipse* software systems that meet these requirements. Instances of restructuring processes in them appear in Section 7.1.3.

7.1.2 Data Analysis Process

For a pair of subsequent versions, v1 and v2, of the system, we computed the following data: base version packages (number of packages of v1), removed packages, added packages, and restructured packages. We also collected NOC, AC, EC, and RMD metrics in v1 and v2. Then, we identified the threshold in which these metrics of the package fall and computed the percentage of packages whose metric values fall in each range of the corresponding thresholds.

With those data, we analyzed whether there is a relationship between the restructuring and the increase in quality indicated by the threshold. We consider this relationship exists when the percentage of packages with the metric in the *Bad/Uncommon* range in v2 is lower than the percentage of *Regular/Casual* and *Good/Common* in v1, as well as the percentage of the packages with the metric in the *Regular/Casual* range in v2 is lower than *Good/ Common* range in v1.

The percentage was used because, in absolute terms, the number of packages in the *Bad/Uncommon* range may increase in the restructured version. However, in relative terms, it is expected to decrease. For example, a package classified as *Bad/Uncommon* may be divided into ten packages, where two are *Bad/Uncommon*, and the rest are in the *Good/Common*. In this case, there is an absolute increase of *Bad/Uncommon* packages in the restructured version; however, there is a considerable relative decrease, from 100% to 20%. As it seeks to evaluate pure restructuring efforts, i.e., those in which there is no implementation of new functionalities or improvements, in this situation, in global terms, the restructuring increased the quality of the software packages. The threshold identified can indicate this. Therefore, it is not assumed that the restructuring will necessarily result only in packages classified as *Good/Common* or Regular/Casual, but it is expected that the overall quality of the restructured portion of the system will increase and that the thresholds will measure this increase.

7.1.3 Types of Restructuring Processes

A restructuring process occurs when a change is performed in the system to improve its internal quality. In this experiment, we considered global and local restructuring. A global restructuring affects the entire system and, therefore, must improve the whole system's quality [Anquetil and Laval, 2011]. In this case, the analysis considered all the system packages. A local restructuring may only consider some of the system's packages; after all, despite the increased package quality related to the restructuring, the system's quality may deteriorate due to other modifications. In this case, we considered only the packages associated with the context of the restructuring in the base and restructured versions, i.e., only the restructured part of the system. Local restructuring involves the possibilities related to packages as follows.

- **New.** When a new package appears on the restructured version but contains classes at some base version, we considered part of the context's restructuring in the new package along with all regrouped classes from the source packages.
- **Moved Class.** When a class moves from an existing package to another, we consider that both packages' quality improved [Anquetil and Laval, 2011]. Hence, we may say that moving the classes among packages corresponds to a restructuring process, and when this happens, we expect the class to move to a more suitable package. Also, we assume the class is best placed after the restructuring, so the package where the class has moved improves its quality. But, note that a restructuring process does not necessarily result in packages with the best possible grouping of classes, but only that class improved the quality of the package.

7.1.4 Operation

After defining the scope and planning the experiment, it is necessary to prepare the system for it. We downloaded the projects considered in this experiment and imported them to *Eclipse*, previously installed with the *Metrics* plugin [Metrics, 2014]. For the experiment, we run the *Metrics* plugin to collect the metrics from the projects. Then, we imported these data to a *Mysql* database to facilitate their analysis and validation.

7.2 Target Systems

This section describes the instances of restructuring identified in the target systems, *JHotDraw* and *Eclipse*.

7.2.1 *JHotDraw*

JHotDraw[4] is a *framework* developed in Java for technical and graphic design. The Anquetil and Laval [2011] work identified JHotDraw versions restructured, identifying three version changes with local restructuring, thorough documentation of the versions, and analysis of the project's source code. Such instances of restructuring are the objects of this experiment. They are:

1. The following packages, org.JHotDraw.draw and org.JHotDraw.app.action, Version 7.4.1, had part of their classes regrouped in 11 and five sub-packages, respectively. In this research, we considered that the restructured and the added packages resulting from the restructuring are part of our local restructuring.
2. Package org.JHotDraw.gui.plaf.palette, version 7.5.1, had some of its classes regrouped in a new subpackage; and classes of org.JHotDraw.gui.event package moved to org.JHotDraw.draw.event package.
3. Package org.JHotDraw.gui.plaf.palette, Version 7.6, had some classes regrouped in a new subpackage, and classes of org.JHotDraw.gui.event package moved to org.JHotDraw.draw.event package.

Table 4 summarizes, quantitatively, the identified local restructuring instances. The *Total Restructured Packages* line in this table includes the amount of restructured packages in the base version plus the added packages in the restructured version. So, suppose two packages were restructured in the base version, and the restructuring process added 16 packages. In that case, the total restructured packages are 18, just like the restructuring identified in column JHotDraw 7.3.1/7.4.1.

7.2.2 *Eclipse*

Eclipse[5] is one of the most popular *IDE* for Java development and other programming languages. To Anquetil and Laval [2011] and the Eclipse documentation, [6] the IDE was restructured from Version 2.1.3 to 3.0, evolving *Eclipse* from the concept of an extensible *IDE* to an *Rich Client Platform*. Anquetil N. et al. considered *Eclipse* restructuring globally because it impacted many packages.

7.3 Experiment Results

This experiment does package restructuring and shows the result for package metrics assessed: NOC, AC, EC, and RMD.

7.3.1 *JHotDraw*

Considering the metric NOC, Table 5 shows that from version 7.3.1 to 7.4.1, the percentage of packages in the *Bad/Uncommon* range decreased from 100% to 11%. From version 7.4.1 to 7.5.1, we observed that the percentage of

[4]http://www.JHotDraw.org/
[5]http://www.eclipse.org
[6]http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/11_Edgar.pdf

Table 4. Local Restructuring - JHotDraw

JHotDraw	Versions 7.3.1/7.4.1	Versions 7.4.1/7.5.1	Versions 7.5.1/7.6
a) Base Version Packages	46	62	65
b) Removed Packages	2	0	1
c) Added Packages	16	3	1
d) Restructured Packages	2	3	1
e) (e=c+d) *Total Restructured Packages*	18	6	2
f) (f=a-b+c)Restructured Version Packages	60	65	65

Table 5. Results of JHotDraw -Restructuring- NOC, AC, EC, and RMD

Metrics	**Versions**	**Packages**	*Bad/Uncommon*	*Regular/Casual*	*Good/Common*
NOC	**From** 7.3.1	2	2 (100%)	-	-
	to 7.4.1	18	2 (11%)	4 (22%)	12 (67%)
	From 7.4.1	3	1 (33%)	2 (67%)	-
	to 7.5.1	4	1 (25%)	1 (25%)	2 (50%)
	From 7.5.1	1	1 (100%)	-	-
	to 7.6	2	1 (50%)	-	1 (50%)
AC	**From** 7.3.1	2	2 (100%)	-	-
	to 7.4.1	18	5 (28%)	8 (44%)	5 (28%)
	From 7.4.1	3	1 (33%)	2 (67%)	-
	to 7.5.1	4	1 (25%)	1 (25%)	2 (50%)
	From 7.5.1	1	1 (100%)	-	-
	to 7.6	2	1 (50%)	-	1 (50%)
EC	**From** 7.3.1	2	2 (100%)	-	-
	to 7.4.1	18	4 (22%)	7 (39%)	7 (39%)
	From 7.4.1	3	1 (33%)	2 (67%)	-
	to 7.5.1	4	1 (25%)	2 (50%)	1 (25%)
	From 7.5.1	1	-	1 (100%)	-
	to 7.6	2	-	1 (50%)	1 (50%)
RMD	**From** 7.3.1	2	-	-	2 (100%)
	to 7.4.1	18	1 (5.55%)	9 (50%)	8 (44.45%)
	From 7.4.1	3	-	3 (100%)	-
	to 7.5.1	4	-	2 (50%)	2 (50%)
	From 7.5.1	1	-	1 (100%)	-
	to 7.6	2	-	2 (100%)	-

packages in the *Bad/Uncommon* range also decreased from 33% to 25%, and from version 7.5.1 to 7.6, the percentage of packages in the *Bad/Uncommon* range decreased from 100% to 50%.

For AC metric, the percentage of packages from versions 7.3.1 to 7.4.1 and versions 7.4.1 to 7.5.1 decreased. The percentage of packages *Regular/Casual* also decreased in versions 7.4.1 to 7.5.1, from 67% to 25%. For versions 7.5.1 to 7.6, Version 7.5.1, in the restructuring context, has one package classified as *Bad/Uncommon* 100%. The restructuring process resulted in two packages in Version 7.6, one *Bad/Uncommon* and one *Good/Common*. Therefore, the percentage of packages in *Bad/Uncommon* range in the restructured version decreased from 100% to 50%.

Table 5 shows for EC metric that from versions 7.3.1 to 7.4.1, the percentage of packages in the *Bad/Uncommon* range decreased from 100% to 22%. From versions 7.4.1 to 7.5.1, the percentage of packages in *Bad/Uncommon* also decreased, from 33% to 25%. The same occurred with the percentage of packages in the *Regular/Casual* range, which decreased from 67% to 50%. From versions 7.5.1 to 7.6, no preexisting package went from a better range to a worse one, and the percentage of packages in the *Regular/Casual* range in the restructured version decreased from 100% to 50%.

Considering the RMD metric, we observed that from versions 7.3.1 to 7.4.1, the percentage of packages in the *Bad/Uncommon* increased from 0% to 5.55%, and the rate of packages in *Good/Common* range decreased from 100% to 44.45%. Besides that, the threshold could not identify the lousy quality of the restructured packages because it classified the two initial packages as *Good/Common*. From versions 7.4.1 to 7.5.1, the percentage of packages classified as *Regular/Casual* decreased from 100% to 50%. From versions 7.5.1 to 7.6, the percentage of packages classified in *Regular/Casual* range was equal in the restructured version. So, the threshold was not able to measure quality improvement.

7.3.2 *Eclipse*

Considering Eclipse's global restructuring, there was an increase of 282 packages. Table 6 shows the absolute and relative number of packages for each package metric and thresh-

Table 6. Global Restructuring - Eclipse

Metrics	Good/Common	Regular/Casual	Bad/Uncommon
NOC	202 (45.5%) → 330 (49.4%)	146 (33.03%) → 217 (32.49%)	94 (21.27%) → 121 (18.11%)
AC	191 (43.21%) → 290 (43.41%)	141 (31.9%) → 223 (33.38%)	110 (24.89%) → 155 (23.2%)
EC	181 (40.95%) → 301 (45.06%)	138 (31.22%) → 206 (30.84%)	123 (27.83%) → 161 (24.1%)
RMD	297 (67.19%) → 450 (67.37%)	101 (22.85%) → 155 (23.2%)	44 (9.95%) → 63 (9.43%)

old range in the base version and the restructured one. The results are analyzed as follows.

- **Number of Classes (NOC)**. The percentage of packages classified in the *Bad/Uncommon* range decreased from 21.27% to 18.11%. If only those 282 packages are considered, only 24 (8.51%) were classified as *Bad/Uncommon*, while 171 (60.64%) were classified as *Good/Common*.
- **Afferent Coupling (AC)**. The percentage of packages in the *Bad/Uncommon* range decreased from 24.89% to 23.2%. Considering only those 282 packages, only 26 (9.22%) would be classified as *Bad/Uncommon*, while 166 (58.87%) would be classified as *Good/Common*.
- **Efferent Coupling (EC)**. The percentage of packages classified as *Bad/Uncommon* decreased from 27.83% to 24.1%. This fact is significant, considering that there was an increase of 282 packages in the restructured version, and only 35 (12.41%) were classified as *Bad/Uncommon*. Also, 159 (56.38%) were classified as *Good/Common*.
- **Normalized Distance (RMD)**. The percentage of packages at the *Bad/Uncommon* range slightly decreased from 9.95% to 9.43%.

7.3.3 Summary

This result indicates that the thresholds of NOC, AC, and EC can capture possible improvements brought by the refactoring process. For RMD, the results do not ensure that the threshold captures the quality increase. Moreover, no preexisting package went from a better range to a worse in the restructuring context at the base and the restructured version.

7.4 Discussion about Thresholds Evaluation in Real Restructuring Processes

This section provides a synthesis of our experiment and a discussion of the results.

It is essential to report that the threshold values were able to indicate, quantitatively, a deteriorated quality in a base software version in favor of increased quality in a restructured version, in relative terms, having been well explained that, in absolute terms, the number of artifacts classified as Bad could have even increased, even though that was not the case. The restructuring effort does not necessarily guarantee that the software quality will increase. However, it is expected, even given the context and wide use of the JHotDraw software for this type of analysis, some improvement concerning the baseline version, which our threshold values were able to indicate.

As we seek to evaluate pure restructuring efforts, that is, those in which no new functionalities or improvements are implemented, in this situation, in global terms, the restructuring increased the quality of the software packages, and the thresholds were able to indicate that.

Therefore, we did not assume that the restructuring would necessarily result only in packages classified as Good/Frequent or Regular/Occasional. Still, the overall quality of the restructured portion of the system is expected to increase, and the thresholds will be able to measure this addition.

A deeper analysis in this regard would require a qualitative study carried out by experts to cross the qualitative analysis with the quantitative analysis. However, as stated by Anquetil and Laval [2011], "if we accept that the restructurings were studied reasonably successful - which is a plausible hypothesis given the continuing success of the Eclipse platform six years later - we remain with various possible explanations."

7.4.1 Answering the Research Question RQ1

We start the discussion of the findings of the experiment at hand by answering the proposed research question (RQ1.) *Does refactoring improves the metric thresholds?*. The goal of studying this research question is to investigate whether the thresholds are sensitive to improvements resulting from refactoring.

The answer to RQ1 is YES. To evaluate thresholds for object-oriented package metrics, we experimented with local restructuring and global restructuring in two widely known software systems *JHotDraw* and *Eclipse,* to verify the ability of these values to measure the quality increased on restructuring processes. The results of this experiment suggest that the proposed thresholds could detect a deteriorated quality at the software base version in favor of increased quality at a restructured version. Specifically, it suggested the efficacy of the threshold for the NOC, AC, and EC metrics and the ineffectiveness of the recommended threshold for the RMD. For the RMD, in the Eclipse, the threshold could not indicate deteriorated quality in the scenario before refactoring, nor the quality increasing in the restructured version. Therefore, we may not reject the null hypothesis. Assuming that the RMD metric is valid and appropriate for evaluating software quality from a packaging perspective, this result would suggest that the threshold value is invalid. However, the derivation process for identifying the threshold for RMD is the same for the metrics NOC, AC, and EC, which had suggestive results in defense of the thresholds. Therefore, this result doubts the validity of this metric in evaluating the quality of object-oriented software packages.

8 Thresholds Catalog Evaluation by Case Studies

Our previous work evaluated the threshold catalog through a case study on a public organization's proprietary software system. According to the developers who maintain it, the system has a deteriorated internal quality. We investigated whether the proposed thresholds followed the developers' evaluation, and the case study results showed our catalog's effectiveness in that context.

In this article, we improved the evaluation of our thresholds. For this purpose, besides the assessment described in Section 7 and the case study shown in our previous work [Filó *et al.*, 2015], we conducted two other case studies and presented them in this section. They aim to verify whether the derived thresholds can assess the internal quality of software systems. We defined the research question **RQ2 to show that we achieved our purpose.** *Are the proposed thresholds able to differentiate poor and good software quality?*- and propose the specific ones, **RQ2.1** and **RQ2.2** in Section 1, which will be answering, respectively, here.

We did Case Study 1 (Section 8.1) to answer **RQ2.1** *Can the proposed thresholds detect poor-quality methods and classes?*This case inspected the methods and classes classified as *Bad/Uncommon* and *Regular/Casual* ranges. Then, we verified whether the qualitative evaluation follows the thresholds, i.e., if the classes with metrics in the ranges *Bad/Uncommon* and *Regular/Casual* have design problems indeed.

The second Case Study (Section 8.2) investigated **RQ2.2** *Can the proposed thresholds detect high-quality methods and classes?*Case Study 2 analyzed *JHotDraw*. As described in Section 8.2, JHotDraw is considered a well-designed project. Therefore, in this case study, we verified whether the JHotDraw quantitative evaluation based on our thresholds catalog corresponds to the so-called high-quality JHotDraw.

These case studies considered all methods and classes from the dataset used in the thresholds derivation process, Qualitas.*class* Corpus [Terra *et al.*, 2013].

8.1 Case Study 1

The purpose of Case Study 1 is to evaluate the ability of the suggested thresholds to detect methods and classes with design problems. We used the database of measures created in the derivation process, which stores the measurements. Using SQL, we filtered the methods and the classes whose measurements fall in the *Bad/Regular* using SQL. We then performed a manual inspection on them and compared the results of this evaluation with the quantitative assessment.

8.1.1 Inspection of the Methods

The Qualitas.class Corpus contains 1,663,526 methods, which makes it challenging to select a viable sample to conduct the inspection, even when using a metric composition strategy. Therefore, we applied the most restrictive filter possible, which selected all methods evaluated in the Bad/Uncommon range by the method metrics present in the catalog: Lines of Code per Method (MLOC), which resulted in 80,061 methods; of which McCabe Complexity (VG), which resulted in 63,030; Number of Parameters (PAR), which resulted in 5,531; and Depth of Nested Blocks (NBD), which resulted in 3,858. Even with the use of all defined threshold values, it was not possible to achieve a reasonable number of methods for manual inspection; after all, 3,858 remains a very high number, despite this quantity representing, relatively speaking, only 0.23% of the total number of sample methods.

In summary, we used a combination of all metrics, MLOC, VG, PAR, and NBD, in the Bad/Uncommon threshold to result in 3,858 methods. We describe the process we adopted to make this selection in the sequel.

1. We sorted the query result by the method name, column *name*. The ordering has no relation to the absolute value of any metric used in the filter.
2. We used a random number generation algorithm to generate ten numbers between 0 and 3,858. The generated numbers are: 92; 205; 963; 1,122; 1,376; 2,683; 2,882; 3,063; 3,417; and 3,727.
3. We also used the random numbers to select the methods for manual inspection, i.e., we considered the n-th method of the ordered list, where n is the random value.

Through this criterion, the selected methods may be from any of the 111 systems in the Qualitas.*class* Corpus. To perform the manual inspection, we imported into *Eclipse* the systems where the selected methods belong. Table 7 shows the chosen methods, exhibiting their name, class, and system. Appendix B describes the manual qualitative evaluation for each method.

8.1.2 Inspection of the Classes

This study aims to evaluate the ability of threshold values to indicate methods and classes with deteriorated quality. To conduct this experiment, we considered all methods and classes of the systems dataset used in the derivation process as targets. To do that, we used the database created in the derivation process, which stores all the measurements. The study consisted of inspecting the methods and classes classified in the Poor/Regular range to compare the qualitative assessment with the quantitative assessment. The Qualitas.class Corpus contains 247,395 classes. First, we considered all ten metrics for the class level by filtering the classes whose metric values fit the Bad/Uncommon range. This first filter did not return to any class. Hence, we tested a second filter without the metric Number of Static Methods (NSM) because we considered this metric less relevant compared to the others. However, the filter remained very restrictive and returned only one class. Subsequently, we excluded the Number of Children (NOC) metric of the filter, given that it does not directly impact the quality of the class but instead on the risk of alteration and failure of the inheritance hierarchy design. Given these two exclusions, we obtained 19 classes, which we considered viable for manual inspection.

From these resulting sets, we excluded three classes. Two are anonymous classes, defined more than once within the classes where the resource is used. The third one is an inner class defined inside several methods of the outer class.

Table 7. Inspected Methods.

Methods	Classes	Systems
addGetterSetterChanges	SelfEncapsulateFieldRefactoring	eclipse-3.7.1
analyseAssignment	QualifiedNameReference	eclipse-3.7.1
diffList	CasualDiff	netbeans-7.3
drawSubCategoryLabels	SubCategoryAxis	jfreechart-1.0.13
findLocalMethods	CompletionEngine	aspectj-1.6.9
prepMinion	GenericStatement	derby-10.9.1.0
readAj5ClassAttributes	AtAjAttributes	aspectj-1.6.9
REPTree.Tree#buildTree	REPTree	weka-3-6-9
startFileInternal	FSNamesystem	hadoop-1.1.2
visitBranchInstruction	CodeSubroutineInliner	proguard-4.9

Table 8. Inspected Classes.

Classes/Systems	Packages
CloneableEditor/netbeans-7.3	org.openide.text
CompilationUnitEditor/eclipse-3.7.1	org.eclipse.jdt.internal.ui.javaeditor
CtxHelpTreesubsection/eclipse-3.7.1	org.eclipse.pde.internal.ua.ui.editor.ctxhelp
DependencyManagementsubsection/eclipse-3.7.1	org.eclipse.pde.internal.ui.editor.plugin
DeploymentDisplay/megamek-0.35.18	megamek.client.ui.swing
GTKFileChooserUI/jre-1.6.0	com.sun.java.swing.plaf.gtk
InfoProduct/compiere-330	org.compiere.apps.search
JarPackageWizardPage/eclipse-3.7.1	org.eclipse.jdt.internal.ui.jarpackager
JoinNode/derby-10.9.1.0	org.apache.derby.impl.sql.compile
JRCTXVisualView/iReport-3.7.5.	com.jaspersoft.ireport.designer.jrctx
JSVGCanvas/batik-1.7	org.apache.batik.swing
JTitledPanel/netbeans-7.3.	org.netbeans.lib.profiler.ui.components
SAX2DTM2/jre-1.6.0.	com.sun.org.apache.xml.internal.dtm.ref.sax2dtm
SAX2DTM2/xalan-2.7.1.	org.apache.xml.dtm.ref.sax2dtm
SimpleCSMasterTreesubsection/eclipse-3.7.1	org.eclipse.pde.internal.ua.ui.editor.cheatsheet.simple
SVGFlowRootElementBridge/batik-1.7	org.apache.batik.bridge.svg12

We selected the 16 classes evaluated through the combination of eight suggested threshold values, which assess different aspects of software quality in object-oriented software classes. In these classes, we conducted a manual inspection to evaluate them qualitatively, which indicated that they all have internal quality problems. They are poorly readable, not very cohesive, and challenging to understand, making them complex, difficult to reuse or extend, and difficult to modify and maintain. The result of the qualitative inspection is in accordance with the threshold values used in the quantitative identification of these classes. These classes can be a risk to the project, as they make the tasks inherent to software maintenance and evolution more complex due to the limited understanding accompanying this type of class. Table 8 exhibits the selected classes, showing their names, packages, and systems. Appendix C presents the qualitative analysis of each class.

8.1.3 Discussion about the Results of Case Study 1

The ten methods evaluated in this case study have relevant complexity, size, and many parameters. They are problematic entities within software systems due to their internal structure, which has characteristics against the fundamental principles of object-oriented programming. These methods have clear opportunities for improvement, and identifying such opportunities does not require much knowledge about the software domain. The results of this study showed that the proposed thresholds correctly pointed out these methods as having metrics in the worst ranges of the metrics. Therefore, we conclude that the proposed thresholds help identify methods with design problems.

The results of the manual inspection for the 16 classes selected in this study indicate that all 16 classes have problems regarding their internal quality. They are less legible, less cohesive, challenging to understand, complex, complicated to reuse or extend, and difficult to modify and maintain. The result of the qualitative inspection agrees with the thresholds used to select these classes. So, we concluded that the proposed thresholds might identify classes that risk the project since they make software maintenance and evolution more complex.

Some studies show that software development professionals spend at least 30% to 50% of the time understanding the code before they start maintenance [Paige and Meyer, 2008]. Avoiding expensive manual inspections is a valuable opportunity to improve software quality during the construction, maintenance, and evolution phases. Using metrics and their thresholds will be helpful in automatically identifying pieces of software with design problems. We should carefully consider the software with design problems when performing

Table 9. Absolute and relative number of classes classified in the ranges suggested

Metrics	+1	0	+1 OR 0	**-1**
NOF	892 (84.07%)	133 (12.54%)	1,025 (96.61%)	36 (3.39%)
NOM	710 (66.92%)	226 (21.30%)	936 (88.22%)	125 (11.78%)
WMC	668 (62.96%)	273 (25.73%)	941 (88.69%)	120 (11.31%)
DIT	715 (67.39%)	222 (20.92%)	937 (88.31%)	124 (11.69%)
LCOM	805 (75.87%)	154 (14.51%)	969 (90.39%)	102 (9.61%)
NSM	990 (93.31%)	19 (1.79%)	109 (95.10%)	52 (4.90%)
NSF	961 (90.57%)	54 (5.09%)	1,015 (95.66%)	46 (4.34%)
SIX	649 (61.17%)	252 (23.75%)	901 (84.92%)	160 (15.08%)
NORM	910 (85.77%)	78 (7.35%)	988 (93.12%)	73 (6.88%)
NSC	969 (91.33%)	48 (4.52%)	1,017 (96.85%)	44 (4.15%)

maintenance and are candidates for refactoring.

8.2 Case Study 2

The Case Study 2 analyzed the framework *JHotDraw*. We chose *JHotDraw* for this case study because previous works have attested to its high design quality [Riehle, 2000]. *JHotDraw* was implemented based on design patterns. Christensen and Henrik [2004] used it in a case study in the context of teaching design patterns. Their choice was because they consider *JHotDraw* a high-quality software system. Hegedűs *et al*. [2012] analyzed *JHotDraw* to investigate the impact of design patterns on maintainability. They found that the use of design patterns contributes to software maintainability.

Assuming that *JHotDraw* is well-designed, Case Study 2 aims to verify how our thresholds behave in evaluating a well-designed software system. We expect that most measures of *JHotDraw* evaluate as having good internal quality. Such a result would suggest that the thresholds can assess the internal quality of a software system as good as it is. So, we could conclude that a measure in the *Good/Common* range is most reliable concerning the qualitative expectation about software quality and minimizes the false-positive evaluations problem. A false-positive appears when the class has no structural issues, but the thresholds indicate the opposite. We expected in Case Study 2 that most classes do not have structural problems, and the thresholds do not reveal them. This conclusion concerns false-negative occurrences when the class has problems, and the thresholds do not indicate them. In this case, it will also be minimized.

We evaluated *JHotDraw* 7.5.1 in the compiled version for *Eclipse* available at Qualitas.*class* Corpus. According to the collected data, this version has 66 packages and 1,061 classes. Table 9 shows the evaluation of the classes of *JHotDraw* considering the proposed thresholds. Columns two to five show the number of classes in absolute and relative values. Column +1 represents *Good/Common* range; Columns 0 and -1 represent *Regular/Casual* and *Bad/Uncommon* ranges, respectively, and Column +1 OR 0 with measure not falling in the *Bad/Uncommon* range.

Observing the **+1 OR 0** column, we may see that five out of 10 of the proposed thresholds evaluate more than 95% of the classes with good or regular quality, while less than 5% are considered bad. The remaining thresholds evaluate at least 85% of the classes with good or regular quality. The result of Table 9 shows another essential fact related to the excellent evaluation obtained by the *JHotDraw* metrics: in column **+1 OR 0**, 91.56% of classes are not evaluated as poor.

8.2.1 Discussion of the Results of Case Study 2

For all the evaluated metrics, most of the classes from *JHotDraw* were classified in the *Good/Common* range, a large amount fell in the *Regular/Casual* range, and just a few fell in the *Bad/Uncommon* range. Considering that the metrics' statistical distributions have such characteristics, one might think that this result is not surprising. But, the case of *JHotDraw* is significant because its measures are expressive in pointing out the system's high quality. The *Good/Common* and *Regular/Casual* were evaluated in conjunction because it is challenging to differentiate accurately whether a class is of good or regular quality by employing manual inspection.

As noted by Ferreira *et al*. [2012], the *Regular/Casual* range may be considered a recommendation that the structure can be improved, although it is not necessarily an anomaly occurrence. Therefore, problems related to the boundary established for the *Good/Common* and the *Regular/Casual* ranges do not have representative consequences for the quantitative evaluation of the software. However, the boundary between the *Regular/Casual* and the *Bad/Uncommon* ranges is subtle. It is worth noting that the thresholds do not dispense evaluations made by experts. Another important consideration is that, as indicated in Section 8, the evaluation of method, class, or package by a single threshold is not recommended in the quality assessment because a broader evaluation provides quantitative results of greater breadth of accuracy.

We use *RAFTool* [Filó *et al*., 2014][7] to conduct some tests on *JHotDraw. RAFTool* allows the possibility of establishing a sorting criterion such that, by choosing, for instance, classes classified by the number of fields as Regular/Casual, the specialist may be aware of which class, even though not being poorly evaluated, shows the highest value of the metric. This assists in decision-making when using the threshold.

8.3 Answering RQ2.1, RQ2.2 and RQ2

RQ2.1 *Can the proposed thresholds detect poor-quality methods and classes?* The answer to this question is YES.

[7]RAFTool - Risk Artifacts Filter Tool.

The results we have obtained in Case Study 1 indicate that the *Bad/Uncommon* ranges defined in the metrics of our threshold catalog are reliable. We observed a strong association between the quantitative and qualitative results; therefore, the study results show that applying the proposed thresholds leads to a minimal "false positive" occurrence.

RQ2.2 *Can the proposed thresholds detect high-quality methods and classes?* The answer to this question is YES. The results we have obtained in Case Study 2 suggest that the *Good/Common* ranges are reliable, i.e., the thresholds may adequately detect the good structural quality of a piece of software. Therefore, we may conclude that applying the thresholds will lead to few "false-negative" occurrences. The results also suggest that the occurrences of "false positive", i.e., when the class does not have a design problem, but the thresholds indicate the opposite, will also be minimized. This fact happens due to the high internal quality of *JHotDraw*. We expected that most classes would not have design problems and that applying the thresholds would reflect this. The result of Case Study 2 confirmed this assumption.

8.3.1 Summary of the Results

Considering the general research question, **RQ2** *Are the proposed thresholds able to differentiate poor and good software quality?* The answer to this general research question is YES. The thresholds evaluated through these case studies suggest they may be applied to assess the quality of methods and classes within object-oriented software. Their application could lower the cost of software quality evaluation since they can reduce the amount of software code that should be inspected. However, it is essential to note that manual inspection done by a software engineer remains necessary for quality management. When a quantitative evaluation points to design problems in a piece of software, that piece should be the subject of analysis by an expert. Therefore, the thresholds provide a way for quantitative and qualitative evaluations to complement each other, leading to a more efficient quality assessment of object-oriented software systems.

9 Threats to Validity

Regarding representativeness, the sample is intrinsically a threat to the validity of any empirical study. To minimize it, we used a large dataset, Qualitas.*class* Corpus. This dataset applies to software systems equivalent to or greater than those used in other empirical studies on software metrics, having more than 16,000 packages, 247,000 classes, and 1,600,000 methods.

When analyzing RQ1, we considered refactoring performed in two software systems: JHotDraw and Eclipse. Although the analysis performed with these systems indicates that the thresholds are sensitive to the refactoring processes at the package level, we can not claim that the results may be generalized to other contexts.

In Case Study 1, we applied the thresholds to identify whether the range *Bad/Regular* may aid in identifying methods and classes with poor design. The analysis was carried out via manual inspection, which might be biased. To mitigate bias in the analysis, we discussed the analysis between the three authors of this paper.

The sample also may impact the generality of the results. The example used in this study comprises various types and domains of software systems. However, the data deriving the thresholds are from open-source Java-based projects. Hence, it is impossible to claim generality to other categories of software systems. Besides, other languages might exhibit different results.

Different software metric tools may implement the same metric differently. So, the tools may report different results of a software metric for the same software system. Qualitas.class Corpus [Terra *et al.*, 2013] relies on Metrics, an Eclipse plugin. Therefore, we may not ensure that the identified thresholds will apply to tools that do not implement the software metrics in the same Metrics pattern. Implementation details are described in *Metrics*[8].

The method to derive the metric thresholds is based on a previous one defined by Ferreira *et al.* [2012]. Such a prior approach is based on manual inspection of graphics, which is an essential threat to the results. Therefore, to avoid this threat, our method does not use manual inspection of graphics to define the thresholds.

10 Final Remarks and Future Work

Knowledge of the thresholds is of fundamental importance to effectively using software metrics to manage the internal quality of software systems. In our previous research, [Filó *et al.*, 2015], we improved the method proposed by Ferreira *et al.* [2012] to derive software metric thresholds. We based our strategy on deriving metric thresholds by analyzing the statistical distribution of measures found in practice. We observed that our metrics fit a heavy-tailed or skewed-right distribution. Exploring the properties of the corresponding data distribution of each metric, we proposed three ranges of values for the metric thresholds: *Good/Common*, *Regular/Casual*, and *Bad/Uncommon*. We applied the resulting approach, defining a catalog of thresholds for 18 object-oriented software metrics and assessed them in a proprietary project. The evaluation results showed that these thresholds effectively detect when a software system has a high internal quality and does not show quality where there is no. The metrics encompass the quantitative evaluation of methods, classes, and packages. Although they do not necessarily express the best design principles established for Software Engineering, they reflect the quality that most software systems follow and may be considered a benchmark for object-oriented software systems.

In this article, we conducted a deeper evaluation of the proposed 18 thresholds to investigate, in a broader context, whether they can evaluate the internal software quality. We summarized our contributions as follows.

- We present a new illustrative example of the application of our method describing the data analysis of the *McCabe Complexity (VG)* metric.

[8]http://metrics2.sourceforge.net/

- We detailed the 18 software metrics considered in this research and discussed their thresholds. For each metric, we provide its application-level -*method, class,* or *package*- its definition formula (when necessary), and design implications.
- We experiment with real-based package restructuring processes to evaluate the ability of our thresholds to identify software quality enhancement.
- We conducted two case studies to evaluate whether the proposed thresholds can differentiate pieces of software well designed and pieces of software with design flaws.
- We showed a qualitative and quantitative evaluation of the thresholds, with the help of a system expert, considering the metrics at class, method, and package levels.
- We show evidence that applying thresholds may lead to few false-positive and false-negative occurrences.
- We provide evidence of the benefits of using thresholds of object-oriented metrics to assist developers and practitioners in the proper quantitative evaluation of internal software quality.

To the best of our knowledge, our metric threshold catalog is the largest one described and empirically evaluated in the literature so far. The proposed thresholds have been used in other studies that the co-authors of this paper have conducted with other researchers. Souza *et al.* [2017] have applied these thresholds to define detection strategies for five bad smells: Large Class, Long Method, Data Class, Feature Envy, and Refused Bequest. Their results show that the strategy based on the thresholds defined in our catalog is significantly helpful. Sousa *et al.* [2017] developed a tool for detecting bad smells in software systems based on software metric thresholds. The authors have evaluated their tool using our proposed threshold catalog. Also, Beranič and Heričko [2019] in their paper said: "Filó *et al.* [2015] introduced two main improvements. The first is connected to the identification of thresholds. Instead of visual identification, Filó *et al.* [2015] introduced the use of two percentiles that divide the values into three areas. The second improvement is connected to threshold naming. They complement existing names to understand each defined threshold better". They presented "derivation approaches resulting in concrete threshold values used for evaluating software projects."

We identify the following main future work: (i) it is essential to evaluate the effectiveness of our catalog to aid maintenance or refactoring tasks along the life cycle of software systems in open-source and proprietary projects; (ii) investigating the correlation between these thresholds and faults in software systems; (iii) deriving thresholds for other software metrics; (iv)evaluating the thresholds with systems different from the ones in Qualitas Corpus; and (v) evaluating whether the thresholds may be impacted regarding software systems' characteristics such as application domain, type, size, and programming language.

Declarations

Acknowledgements

The authors thank the Programming Language Laboratory participants who volunteered to help with the tests.

Funding

This research was supported by UFMG/DCC, Programming Language Research Group, UFMG Graduate Program in Computer Science, and CEFET-MG.

Authors' Contributions

Tarcísio G. S. Filó: Conceptualisation; Investigation; Methodology; Visualisation; Writing – original draft; Data curation; Software; Formal Analysis; Validation;
Mariza A. S. Bigonha: Project administration; Conceptualisation; Metothodoly; Formal Analysis; Supervision; Visualization; Validation; Writing – review & editing; Resources;
Kecia A. M. Ferreira: Conceptualisation; Methodology; Formal Analysis; Supervision; Visualization; Validation; Writing – review & editing.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Besides the material in the Appendices, data analysis of the threshold derivation method and other analyzes are available from the first author upon request.

References

Abílio, R., Padilha, J., Figueiredo, E., and Costa, H. (2015). Detecting Code Smells in Software Product Lines – An Exploratory Study. In *12th International Conference on Information Technology - New Generations*, pages 433–438, New York, NY, USA. DOI: 10.1109/ITNG.2015.76.

Al Dallal, J. (2011). Improving the applicability of object-oriented class cohesion metrics. *Information and Software Technology*, 53:914–928. DOI: 10.1016/j.infsof.2011.03.004.

Al Dallal, J. (2012). Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, 54:1125–1141. DOI: 10.1016/j.infsof.2012.04.004.

Al Dallal, J. and Briand, L. (2010). An object-oriented high-level design-based class cohesion metric. *Information and Software Technology*, 52:1346–1361. DOI: 10.1016/j.infsof.2010.08.006.

Alshayeb, M. (2009). Refactoring effect on cohesion metrics. In *Computing, Engineering and Information, ICC*, pages 3–7. DOI: 10.1109/ICC.2009.12.

Alves, T., Ypma, C., and Visser, M. (2010). Deriving metric thresholds from benchmark data. In *Proc. of the IEEE Int.*

Conference on Software Maintenance, pages 1–10. DOI: 10.1109/ICSM.2010.5609747.

Aniche, M., Treude, C., Zaidman, A., Van Deursen, A., and Gerosa, M. A. (2016). SATT: Tailoring Code Metric Thresholds for Different Software Architectures. In *IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 41–50. DOI: 10.1109/SCAM.2016.19.

Anquetil, N. and Laval, J. (2011). Legacy software restructuring: Analyzing a concrete case. In *15th European Conf. on Soft. Maint. and Reengineering*, pages 279–286. DOI: 10.1109/CSMR.2011.34.

Arar, O. and Ayan, K. (2016). Deriving Thresholds of Software Metrics to Predict Faults on Open Source Software: Replicated Case Studies. *Expert Systems with Applications*, 61:106–121. DOI: 10.1016/j.eswa.2016.05.018.

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 *OOPSLA*, pages 397–412. ACM. DOI: 10.1145/1167473.1167507.

Bender, R. (1999). Quantitative risk assessment in epidemiological studies investigating threshold effects. *Biometrical Journal*, 41:305–319. DOI: 10.1002/(SICI)1521-4036(199906)41:3<305::AID-BIMJ305>3.0.CO;2-Y.

Benlarbi, S., Khaled, E., Nishith, G., and Shesh, R. (2000). Thresholds for Object-Oriented Measures. In *Proc. 11th International Symposium on Software Reliability Engineering*, pages 24–38. IEEE. DOI: 10.1109/ISSRE.2000.885858.

Beranič, T. and Heričko, M. (2017). Approaches for Software Metrics Threshold Derivation: A Preliminary Review. In *Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Application*, pages 1–8. Available at:https://ceur-ws.org/Vol-1938/paper-ber.pdf.

Beranič, T. and Heričko, M. (2019). Comparison of systematically derived software metrics thresholds for object-oriented programming languages. *Computer Science and Information Systems*, 17(1):181–203. Available at:https://www.researchgate.net/publication/337589114_Comparison_of_systematically_derived_software_metrics_thresholds_for_object-oriented_programming_languages.

Bigonha, M., Ferreira, K., Souza, P., Sousa, B., Januário, M., and Lima, D. (2019). The Usefulness of Software Metric Thresholds for Detection of Bad Smells and Fault Prediction. *Information and Software Technology*, 115:79–92. DOI: 10.1016/j.infsof.2019.08.005.

Bloch, J. (2008). *Effective Java*. 2nd Ed.Prentice Hall. Book.

Boucher, A. and Badri, M. (2016). Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software. In *4th Intl Conf on Applied Computing and Info. Technology/3rd Int. Conf on Computational Science/Int. and Applied Inf./1st Intl Conf on Big Data, Cloud Computing, Data Sc. Eng.*, pages 169–176. DOI: 10.1109/ACIT-CSII-BCD.2016.042.

Boucher, A. and Mourad, B. (2018). Software metrics thresholds calculation techniques to predict faultproneness: An empirical comparison. *Inf. and Soft. Technology*, 96:38–67. DOI: 10.1016/j.infsof.2017.11.005.

Bouwers, E., V. J. and Van, D. A. (2012). Getting what you measure. *Queue ACM*, 10(5):54–59. DOI: 10.1145/2209249.2209266.

Chhikara, A., Chhillar, R. S., and Khatri, S. (2011). Evaluating the impact of different types of inheritance on the object oriented software metrics. *Int. Journal of Enterprise Comp. and Business Syst.*, 1(2). Available at:https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=ac28a2f416fc55d657b98a4a752a2b2b4144ab0f.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493. DOI: 10.1109/32.295895.

Christensen and Henrik, B. (2004). Frameworks: Putting design patterns into perspective. In *Proc. SIGCSE Conf. on Innovation and technology in Comp. Science Education*, pages 142–145. DOI: 10.1145/1007996.1008035.

E. Tempero, C. Anslow, J. D. T. H. J. L. M. L. H. M. and Noble, J. (2010). The qualitas corpus: A curated collection of java code for empirical studies. In *Proceedings of the 17th Asia Pacific Software Engineering Conference*, pages 336–345, Sidney, Australia. DOI: 10.1109/APSEC.2010.46.

Fan, Z., Feng, Z., Xue, X., Chen, S., and Wu, H. (2020). Ecosystem evolution analysis and trend prediction of projects in Android application framework. In *Proc. 35th IEEE/ACM Int. Conf. on Automated Soft.Eng. Workshops*, pages 67–72. DOI: 10.1145/3417113.3422185.

Feng, Q., Cai, Y., Kazman, R., Cui, D., Liu, T., and Fang, H. (2019). Active Hotspot: An Issue-Oriented Model to Monitor Software Evolution and Degradation. In *34th Int. Conf. on Automated Soft. Engineering*, page 986–997, New York, NY, USA. IEEE/ACM. DOI: 10.1109/ASE.2019.00095.

Fenton, N. (1994). Software Measurement: A Necessary Scientific Basis. *Soft. Eng., IEEE Transactions on*, 20:199 – 206. DOI: 10.1109/32.268921.

Ferreira, K., Bigonha, M., Bigonha, R., Mendes, L., and Almeida, H. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257. DOI: 10.1016/j.jss.2011.05.044.

Filó, T., Bigonha, M., and Ferreira, K. (2014). Raftoolferramenta de filtragem de métodos, classes e pacotes com medições incomuns de métricas de software. In *Proc. WAMPS-SOFTEX*, pages 42–48. written in portuguese.

Filó, T., Bigonha, M., and Ferreira, K. (2014). Statistical dataset on software metrics in object-oriented systems. *ACM SIGSOFT Software Engineering Notes*, 39(5):1–6. DOI: 10.1145/2659118.2659130.

Filó, T., Bigonha, M., and Ferreira, K. (2015). A Catalog of Thresholds for Object-Oriented Software Metrics. In *Proc. of the 1st SOFTENG*, pages 48–55. Available at:https://llp.dcc.ufmg.br/Publications/Art2015/2015-softeng-Catalogo-Tarcisio.pdf.

Fontana, F. A., Ferme, V., Zanoni, M., and Yamashita, A. (2015). Automatic Metric Thresholds Derivation for Code Smell Detection. In *IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, pages 44–

53, Florence, Italy. DOI: 10.1109/WETSoM.2015.14.

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

Gil, J. A. and Lalouche, G. (2016). When do software complexity metrics mean nothing? – when examined out of context. *The Journal of Object Technology*, 15:2–1. Available at:https://www.jot.fm/issues/issue_2016_01/article2.pdf.

Hegedűs, P., Bán, D., Ferenc, R., and Gyimóthy, T. (2012). Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability. In *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity Communications in Computer and Information Science*, volume 340 Springer, pages 138–145. DOI: 10.1007/978-3-642-35267-6$_1$8.

Herbold, S., Grabowski, J., and Waack, S. (2011). Calculation and optimization of thresholds for sets of software metrics. *Empirical Software Engineering*, 16:812–841. DOI: 10.1007/s10664-011-9162-z.

Hevery, M. (2008). *Google testing: Static Methods are death to testability*. Available at: https://testing.googleblog.com/2008/12/static-methods-are-death-to-testability.html.

Hussain, S., Keung, J., Khan, A. A., and Bennin, K. E. (2016). Detection of Fault-Prone Classes Using Logistic Regression Based Object-Oriented Metrics Thresholds. In *IEEE Int. Conf. on Soft. Quality, Reliability and Security Companion (QRS-C)*, pages 93–100. DOI: 10.1109/QRS-C.2016.16.

Kaur, S., Singh, S., and Kaur, H. (2013). A Quantitative Investigation Of Software Metrics Threshold Values At Acceptable Risk Level. *Int. Journal of Engineering Research & Technology*, 2(3):1–7. Available at:https://www.ijert.org/research/a-quantitative-investigation-of-software-metrics-threshold-values-at-acceptable-risk-level-IJERTV2IS3218.pdf.

Kitchenham, B. (2010). Whats up with software metrics? A preliminary mapping study. *JSS*, 83(1):37–51. DOI: 10.1016/j.jss.2009.06.041.

Lanza, M. and Marinescu, R. (2010). *Object-Oriented Metrics in Practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Publishing Company. Book.

Lavazza, L. and Morasca, S. (2016). An empirical evaluation of distribution-based thresholds for internal software measures. In *Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, page 6. IEEE. DOI: 10.1145/2972958.2972965.

Li, W. (1998). Another Metric Suite for Object Oriented Programming. *J. of System and Software*, 44:155–162. DOI: 10.1016/S0164-1212(98)10052-3.

Li, W. and Henry, S. (1993). Object-oriented that predict maintainability. *J. of System and Software*, 23:111–122. DOI: 10.1016/0164-1212(93)90077-B.

Lochmann, K. (2012). A benchmarking-inspired approach to determine threshold values for metrics. *SIGSOFT Softw. Eng. Notes*, 37(6). DOI: 10.1145/2382756.2382782.

Lopez, M. and Habra, N. (2005). Relevance of the cyclomatic complexity threshold for the java programming language. In *Proceedings of the 2nd Software Measurement European Forum*, pages 195–202. Available at:https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=261586522dd02a93ac48e9baedecae22e3751545#page=203.

Lorenz, M. and Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, USA. Book.

Malhotra, R. and Bansal, A. (2015). Fault prediction considering threshold effects of object-oriented metrics. *Expert Systems*, 32:203–219. DOI: 10.1111/exsy.12078.

Malhotra, R. and Bansal, A. (2017). Identifying threshold values of an open source software using Receiver Operating Characteristics curve (ROC). *Journal of Information and Optimization Sciences*, 38:39–69. DOI: 10.1080/02522667.2015.1135592.

Martin, R. (1994). OO Design Quality Metrics - An Analysis of Dependencies. In *Proc. of Works. on Pragmatic and Theoretical Directions in Object-Oriented Soft. Metrics*. Available at:https://linux.ime.usp.br/~joaomm/mac499/arquivos/referencias/oodmetrics.pdf.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practice*. Prentice Hall, Upper Saddle River, New Jersy, USA. Book.

McCabe, T. (1976). A complexity measure. *Software Engineering IEEE Transactions*, 2(4):308–320. DOI: 10.1109/TSE.1976.233837.

Mei, Y., Rong, Y., Liu, S., Guo, Z., Yang, Y., Lu, H., Tang, Y., and Zhou, Y. (2023). Deriving Thresholds of Object-Oriented Metrics to Predict Defect-Proneness of Classes: A Large-Scale Meta-Analysis. *International Journal of Software Engineering and Knowledge Engineering*, 33:651–695. DOI: 10.1142/S0218194023500110.

Metrics (2014). Eclipse Metrics plugin 1.3.8. URL:. Available at:http://metrics2.sourceforge.net. Accessed in: 2014 12-30.

Mishra Alok, S. R., Cagatay, C., and Akhan, A. (2021). Techniques for Calculating Software Product Metrics Threshold Values: A Systematic Mapping Study. *Applied Sciences*, 11(23). DOI: 10.3390/app112311377.

Mohammed, A., Mohammed, A., and Lahouari, G. (2019). Threshold Extraction Framework for Software Metrics. *J. of Comp. Sci. and Technology*, 34(5):1063–1078. DOI: 10.1007/s11390-019-1960-6.

Mohović, M., Mausa, G., and Galinac Grbac, T. (2018). Using Threshold Derivation of Software Metrics for Building Classifiers in Defect Prediction. In *Proc. of the SQAMIA 2018: 7th Works. of Soft. Quality, Analysis, Monitoring, Improvement, and Applications. Also published online by CEUR Works. Proc.*, pages 11–1 – 11–9, Novi Sad, Serbia. Available at:https://ceur-ws.org/Vol-2217/paper-moh.pdf.

Morasca, S. and Lavazza, L. (2016). Slope-based fault-proneness thresholds for software engineering measures. In *Proc. of the 20th Int. Conf. on Evaluation and Assessment in Soft. Engineering*, pages 1–10. DOI: 10.1145/2915970.2915997.

Morasca, S. and Lavazza, L. (2017). Risk-averse slope-based thresholds: Definition and empirical evaluation. *Information and Software Technology*, 89:37 – 63. DOI: 10.1016/j.infsof.2017.03.005.

Mori, A. (2018). Design and Evaluation of a Method to Derive Domain Metric Thresholds. Master's thesis, UFMG, Belo Horizonte, Minas Gerais. Available at:https://repositorio.ufmg.br/handle/1843/ESBF-B5UMFW.

Oliveira, P., Lima, F. P., Valente, M. T. O., and Serebrenik, A. (2014a). RTTool: A Tool for Extracting Relative Thresholds for Source Code Metrics. In *2014 IEEE Int. Conf. on Soft. Maint. and Evolution*, pages 629–632. DOI: 10.1109/ICSME.2014.112.

Oliveira, P., Valente, M., and Lima, F. (2014b). Extracting relative thresholds for source code metrics. In *Conf. on Soft. Evolution Week*, pages 254–263, Antwerp, Belgium. CSMR-WCRE IEEE. DOI: 10.1109/CSMR-WCRE.2014.6747177.

Oracle (2014). Javase technical documentation. Available at: http://docs.oracle.cp,/javase/. Accessed: 2014-12-30.

Padhy, N., Panigrahi, R., and Neeraja, K. (2021). Threshold estimation from software metrics by using evolutionary techniques and its proposed algorithms, models. *Evolutionary Intelligence - Special Issue*, 14:315–329. DOI: 10.1007/s12065-019-00201-0.

Paige, R. and Meyer, B. (2008). Objects, Components, Models, and Patterns. In *Proc. 46th Int. Conference, TOOLS EUROPE*. DOI: 10.1007/978-3-540-69824-1.

Radjenović, D., Hericko, M., Torkar, R., and Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55:1397–1418. DOI: 10.1016/j.infsof.2013.02.009.

Riehle, D. (2000). Framework design: A role modeling approach. *Software Technik-Trends*, 20(4). Available at:https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/144452/1/eth-23315-01.pdf.

Rosenberg, L., Ruth, S., and Gallo, A. (1999). Risk-based Object Oriented Testing. In *Proc.24th S.E. Workshop*, pages 1–6. NASA, S.E.Lab. Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0cadbbdc244f38e4dfbae022b5e5e3fdc8249dd0.

Shatnawi, R. (2010). A Quantitative Investigation of the Acceptable Risk levels of Object-Oriented Metrics in Open-Source Systems. *IEEE Transactions on Software Engineering*, 36:216–225. DOI: 10.1109/TSE.2010.9.

Shatnawi, R. (2015). Deriving metrics threshold using log transformation. *J. of Soft. and Evol. Process*, 27:95–113. DOI: 10.1002/smr.1702.

Shatnawi, R. (2018). Identifying Threshold Values of Change-Prone Modules. In *International Conference on E-business and Mobile Commerce*, pages 39–43, Chengdu, China. ACM. DOI: 10.1145/3230467.3230477.

Shatnawi, R. and Althebyan, Q. (2013). An Empirical Study of the Effect of Power Law Distribution on the Interpretation of OO Metrics. *ISRN Software Engineering*, 2013:18. DOI: 10.1155/2013/198937.

Shatnawi, R., Li, W., Swain, J., and T., N. (2009). Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(1):1–16. DOI: 10.1002/smr.404.

Singh, S. and Kaur, K. S. (2014). Object oriented software metrics threshold values at acceptable risk level. *Int. Journal CSIT*, 2:191–205. DOI: 10.1007/s40012-014-0057-1.

Sodiya, A. S., Aborisade, O., and Ikuomola, A. (2012). A survivability model for object-oriented software systems. In *Fourth International Conference on Computational Aspects of Social Networks (CASoN)*, pages 283–290. DOI: 10.1109/CASoN.2012.6412416.

Sommerville, I. (2012). *Software Engineering*. Pearson, 9th edition. Book.

Sousa, B. L., Souza, P. P., Fernandes, E., Ferreira, K., and Bigonha, M. A. S. (2017). Findsmells: flexible composition of bad smell detection strategies. In ICPC, editor, *Proceedings of the 25th International Conference on Program Comprehension*, pages 360–363. DOI: 10.1109/ICPC.2017.8.

Souza, P. P., Sousa, B. L., Ferreira, K. A. M., and Bigonha, M. A. S. (2017). Applying software metric thresholds for detection of bad smells. In *Proc. of the 11th SBCARS*, pages 6–1–6–10. ACM. DOI: 10.1145/3132498.3134268.

Stojkovski, M. (2017). Thresholds for Software Quality Metrics in Open Source Android Projects. Norwegian Univ. of Science and Technology Comp. Sc. Department. Available at: http://hdl.handle.net/11250/2479193.

Sultan, A. (2021). Predicting Relative Thresholds for Object Oriented Metrics. In *IEEE/ACM International Conf. on Technical Debt (TechDebt)*, pages 55–63. IEEE. DOI: 10.1109/TechDebt52882.2021.00015.

Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. (2013). Qualitas.Class corpus: A compiled version of the qualitas corpus. *Soft. Eng. Notes*, 38(5):1–4. DOI: 10.1145/2507288.2507314.

Vale, G., A. D., Figueiredo, E., and A., G. (2015). Defining metric thresholds for software product lines: A comparative study. In *Proc. of the 19th Int. Conf. on Soft. Product Line*, pages 175–185. DOI: 10.1145/2791060.2791078.

Vale, G., Fernandes, E., and Figueiredo, E. (2019). On the proposal and evaluation of a benchmark-based threshold derivation method. *Software Quality Journal*, 27(1):275–306. DOI: 10.1007/s11219-018-9405-y.

Vale, G. and Figueiredo, G. (2015). A method to derive metric thresholds for software product lines. In *29th Brazilian Symp. on Soft. Eng.(CBSOFT)*, pages 110–119. DOI: 10.1109/SBES.2015.9.

Veado, L., Vale, G., Fernandes, E., and Figueiredo, E. (2016). TDTool: threshold derivation tool. In *Proc. 20th Inter. Conference on Evaluation and Assessment in Software Engineering*, pages 1–5. DOI: 10.1145/2915970.2916014.

A An Outline and a Detailed Description of each Metric and their Thresholds

In this appendix, we describe the *software metrics* considered in this research and discuss their thresholds. For each metric, we provide the following information: (1) level to which the metric applies (method, class, or package); (2) definition; (3) formula - if necessary; and (4) implications for design.

1. Afferent Coupling (AC)
1. Level. Package.
2. Definition. The number of classes outside a package that depends on classes inside the package Metrics Metrics [2014].
4. The implication for design. The higher the AC value, the greater the responsibility of that package and its relevance within the software architecture. A package with many external dependencies may be a risk artifact since when it is necessary to make any change, this may directly impact many classes and indirectly have consequences on the stability of the entire application. Packages with high AC values should receive a more significant effort of testing and monitoring, as any change may become critical to the software. The threshold of AC identified in this work helps indicate a high AC value. Lanza and Marinescu [2010] reported that responsibilities within object-oriented software must be uniformly distributed in software artifacts. Knowing that an AC value is high allows us to divide the responsibilities of an overly influential package into more cohesive packages.

2. Efferent Coupling (EC)
1. Level. Package.
2. Definition. The number of classes inside a package depends on classes outside the package [Metrics, 2014].
4. The implication for design. The higher the EC value, the higher the number of classes the package depends on. A high value of EC indicates that the package could be more unstable since a change outside them would impact the package classes at a high level. Keeping the minimum EC value means obtaining a package with greater self-sufficiency. The thresholds derived suggest that most of the packages in object-oriented software systems depend upon no more than six classes. Occasionally, the packages depend upon up to 16 classes, but rarely do they depend upon more than that. A possible decision about a software package with a high EC value (*Bad/Uncommon*) would be to divide it into smaller and possibly more cohesive packages.

3. Method Lines of Code (MLOC)
1. Level. Method.
2. Definition. Count and sum non-blank and non-comment lines inside method bodies Metrics [2014].
4. The implication for design. For [Fowler *et al.*, 1999], since the early days of programming, it has been realized that the larger a procedure, the more difficult it is to understand. Fowler suggests that many problems in object-oriented software systems are due to long methods, which are problematic because they have much information besides complexity. Smaller methods are more natural to understand, reuse, and overwrite, making the maintenance and evolution of the system less complicated and cheaper. Sommerville [2012] says that code size is one of the most reliable predictive-error-prediction metrics. It is verified here, with an empirical study, that most methods have just a few numbers of code lines. This observation and our evaluation suggest that methods should have at most ten code lines.

4. Number of Classes Per Package (NOC)
1. Level. Package.
2. Definition. The total number of classes in the selected scope Metrics [2014].
4. The implication for design. A package is a grouping of classes and interfaces related to the same purpose, providing access protection and ease of localization Oracle [2014]. If we add more classes to a given package, the classes and interfaces grouping tend to be less interrelated. A higher value of NOC suggests a possible split into smaller packages, making it easier to find related classes and understand the package domain more quickly. To identify packages that do not follow the pattern of most ones developed in object-oriented software, we also suggest using these thresholds to evaluate possible adaptations within the project.

5. Number of Fields (NOF)
1. Level. Class.
2. Definition. The total number of fields is defined in the selected scope [Metrics, 2014].
4. The implication for design. According to [Fowler *et al.*, 1999], a class with many attributes indicates that its modeling may contain some problems and should be restructured by grouping related attributes into new objects. These classes may become problematic because they may store many states, carrying unnecessary complexity. In this way, we may use the proposed thresholds to identify candidate classes to undergo a refactoring process, which may raise the quality level of the software. According to the threshold values suggested in this work, a class with more than eight attributes is a candidate for refactoring.

6. Number of Methods (NOM)
1. Level. Class.

2. Definition. The total number of methods is defined in the selected scope [Metrics, 2014].

4. The implication for design. According to Lanza and Marinescu [2010], we must uniformly distribute the system's logic between classes in a good object-oriented design. A class with many methods tends to be more complex. It has many responsibilities, breaking one of the main principles of object-oriented systems, which suggests that a class should take only one responsibility [Martin, 2003]. In this direction, the derived threshold indicates that we may construct the classes with, at most, six methods. Classes with up to 14 methods occur occasionally and should be candidates for refactoring.

7. Number of Overridden Methods (NORM)
1. Level. Class.
2. Definition. Substitution of the total number of methods in the selected scope to an ancestor class Metrics [2014]. The

calculation of NORM excludes *toString*, equals, and *hashCode* methods.

4. Implication for design. A high value of this metric indicates that the superclass may not be an appropriate parent class for the subclass, suggesting problems in the project's inheritance hierarchy Sommerville [2012]. Therefore, the threshold identified as *Good/Common* suggests that, typically, child classes override up to two methods of the parent classes. This threshold indicates that child classes reuse parent-class behaviors by overwriting a reduced number of methods.

8. Number of Children (NSC)

1. Level. Class.

2. Definition. The total number of direct subclasses of a class Metrics [2014]. A Class implementing an interface counts as a direct child of that interface.

4. Implication for design. Classes with many children are difficult to modify and will require more testing on the system because a change in the parent class may affect all of its child classes Chidamber and Kemerer [1994]. Besides, there is a higher likelihood of misusing the parent class abstraction due to this more significant number of children. Such a situation indicates that the designer has improperly applied the inheritance in the project. The thresholds we suggested establish that a class with four or more children may be a software maintenance risk. This risk is due to the impact a change in the Class may have on the system. The suggested thresholds follow other works, Benlarbi *et al.* [2000], which have derived values two and three. A result is under the *Regular/Occasional* range set forth herein. Since we use three ranges of values, the *Good/Frequent* and *Regular/Occasional* ranges are also distinct.

9. Number of Static Fields (NSF)

1. Level. Class.

2. Definition. The total number of static fields in a class.

4. The implication for design. A static field creates a field belonging to the Class instead of being associated with the class instance Oracle [2014]. All class instances share the static field in a fixed place in memory. Any changes in the value of these fields reflect on all instances of the Class. Static fields greatly benefit object-oriented software projects developed on the Java platform Bloch [2008]. An example is the implementation of the *Singleton* design pattern, which guarantees the existence of only one instance of a given class, providing global access to that object. Static attributes may also be exposed via *public static final* modifiers, assuming that constants form a cohesive part of the abstraction the Class provides. However, classes that overuse this feature have acquired a bad reputation because it prevents developers from thinking about objects. An example of misuse is the *constant interface* Bloch [2008], which are interfaces made up of attributes *static final*, each representing the exportation of a constant. Classes implement these constants only to avoid the need to qualify the *interface* name. We expose the details using these attributes while implementing class functionalities, and a class suffers the risk of hiding information. If the *interface* is changed, the behavior of the inherited Class may also be changed, and there is no good project encapsulation. An alternative that best translates object-oriented programming, in this case, is the use of *enumerated types* (*enum*). *Enum* consists of a fixed set of constants, which translate cohesive groups of constants possibly necessary for the applications. The proposed threshold may indicate a high value for NSF, which allows identifying classes in the software using this feature excessively or even creating a simple mechanism of detecting the *constant interface*.

10. Number of Static Methods (NSM)

1. Level. Class.

2. Definition. The total number of static methods in a class.

4. The implication for design. The static method handles only static attributes. We invoke it with the class name without instantiating an object. This kind of mechanism breaks the concept of object-oriented programming, making it somewhat equivalent to procedural programming. However, it has practical utility in object-oriented software development on the Java platform Bloch [2008]. One of the scenarios in which this kind of mechanism is valuable is a simple static method that returns an instance of the Class rather than using the constructor. Doing this allows the Class to be *instance-controlled*, which causes, for example, a class to enforce *Singleton* behavior, using the constructor private. Another possible use of this feature is utility classes, clusters of static methods whose instantiation would not be justifiable. However, its use also has disadvantages. The breakdown of the object-oriented paradigm represented by this type of mechanism negatively affects several aspects related to object-oriented programming software quality. Static methods worsen software testability once unit tests are based on the concept that a method may run and be evaluated independently Hevery [2008]. The dependencies of the method tested are simulated with the possible results and how the evaluated code will respond, regardless of the good or bad functioning of the dependent code. Any method that depends on some static method has its testability impaired, as the static method will consistently execute, violating the idea of unit tests. The problem maximizes when this method is very complicated. Another downside of static methods is that they make the software less flexible since they may not overwrite. The thresholds suggested values may discriminate, given the way software is developed; what is a *Good/Common*, *Regular/Casual* and *Bad/Uncommon* for the NSM metrics, giving a quantitative means of distinguishing what is a high or low value to evaluate the use of static methods.

11. Number of Parameters (PAR)

1. Level. Method.

2. Definition. The total number of parameters in a method.

4. The implication for design. For [Fowler *et al.*, 1999], it is tough to understand a long parameter list, and they become inconsistent and confusing. The parameter list does not need to be regularly changed every time more data is or is no longer needed if the parameter is an object instead of a primitive type. This fact reduces the number of object-oriented programming parameters relative to traditional paradigms. The derived threshold translates this observation into statistical terms. Methods with up to two parameters are the most frequent and represent the level of quality that software developers usually work, providing a reference for evaluating the methods through these metrics. The low value indicates that object-oriented programming tends to have a small parameter list, and the *Regular/Occasional* range, with a value of four, is also low.

12. Specialization Index (SIX)

1. Level. Class.

2. Definition. [Lorenz and Kidd, 1994] have defined SIX metrics as the ratio of the number of overridden methods in the evaluated Class weighted by the Class's DIT metric over the Class's number of methods ($SIX = NORM * DIT/NOM$).

3. Formula. The specialization index's average is NORM * DIT / NOM [28].

4. The implication for design. This metric aims to evaluate how much a given class overrides the behavior of its superclasses. According to the metric authors, the higher the specialization index value, the greater the execution of specific behaviors at run time. This indicates a more complex class, suggesting that it receives more testing effort. Also, when a class has a high degree of Specialization Index, its child classes may not conform to the abstraction of the parent class. Thus, identifying a high value for SIX is significant because it allows for the identification of classes with high specialization, which contributes significantly to the change of behavior of the software at run time. This fact makes the software more prone to complex errors and makes it challenging to test. Therefore, the thresholds may help identify classes out of compliance with the projected inheritance hierarchy.

13. *McCabe* Complexity (VG)

1. Level. Method.

2. Definition. The VG metric aims to assess the complexity of a program McCabe [1976]. VG represents the code execution flow employing a graph, where the nodes represent program commands, and a directed edge from node A to B represents that the program flow may go from A to B. VG counts the number of independent paths in the program execution graph.

3. Formula. The metric is given by $McCabe = E - N + P$, where, given a graph, E is the number of edges in the graph, N is the number of nodes, and P is the number of connected components.

4. The implication for design. The metric *McCabe* Complexity has been used to evaluate the complexity of methods in object-oriented software. According to [Sommerville, 2012], this metric is related to the facility of understanding the software at the method level. Keeping methods with reasonable cyclomatic complexity, as suggested by the identified thresholds, We expect the code to be easier to understand, less error-prone, and easier to change and reuse. [Lopez and Habra, 2005] evaluate the threshold value, 10, suggested by the metric authors themselves, in object-oriented programming. The authors conclude this value cannot sufficiently discriminate sophisticated methods in object-oriented software projects. More than 90% of the evaluated methods have a complexity of less than five, and only 2% of the methods have a complexity more significant than 10. This fact makes the value 10, suggested by the metric authors for functional programming, inefficient in the context of object-oriented programming. These results are consistent with the thresholds suggested in this work, two and four to separate the *Good/Common*, *Regular/Casual*, and *Bad/Uncommon* ranges.

14. Weighted Methods per Class (WMC)

1. Level. Class.

2. Definition. The WMC metric consists of the sum of the complexities of the methods that compose a class. The authors of this metric leave the definition of complexity open. In Metrics, the tool used to collect metrics, WMC is the sum of the McCabe Cyclomatic Complexity of the methods within the Class.

4. The implication for design. Chidamber and Kemerer [1994], consider this metric an indicator of the development and maintenance effort of the Class in the analysis. The higher the number of methods in a class, the more significant the tendency of the Class is less specific, thus limiting its potential for reuse and impairing cohesion. For Sommerville [2012], the larger the metric value, the more complex the object will be. Complex objects are more challenging to understand and, therefore, difficult to maintain. These objects are also more challenging to reuse since the high complexity impairs class cohesion. Therefore, by keeping the WMC values within the patterns identified in this work, it is expected that software quality attributes such as ease of maintenance, ease of comprehension, and reusability will be improved. Besides, the value 11, suggested in this paper to classify the classes in the *Good/Common*, is close to or within the acceptance range established in other works [Benlarbi *et al.*, 2000; Chhikara *et al.*, 2011; Rosenberg *et al.*, 1999; Kaur *et al.*, 2013; Shatnawi, 2010].

15. Depth of Inheritance Tree (DIT)

1. Level. Class.

2. Definition. DIT metric represents the distance from a given class to the root class in the inheritance tree [Chidamber and Kemerer, 1994].

4. Implication for design. The result we have obtained agrees with the thresholds proposed by Ferreira *et al.* [2012], who found the mean value of DIT to be two. They consider that classes with DIT greater than two may indicate a bad use of inheritance in the system. However, in this research, the ranges *Regular/Casual* and *Bad/Uncommon* have also been identified. We may explain this action because the sample used in this study is significantly big and has higher metric values, even if infrequent but significant in tail formation. For this reason, we use another distribution as the best fit for the values of the metric, which has asymmetry on the right, and, consequently, the typical value is not the average. This fact allows for identifying values in the *Good/Common*, *Regular/Casual*, and *Bad/Uncommon* ranges. However, although we do not consider the mean value as a representative, the value found in this work for the *Good/Common* range is the same one found as typical by [Ferreira *et al.*, 2012]. According to Sommerville [2012], the higher the depth of the inheritance tree, the more complex the project will be. The deeper a class is in a class hierarchy, the more methods it inherits, making the class more complicated. The derived threshold for the *Good/Common* range translates into a shallow inheritance tree, with no more than two levels, which holds the child classes close to the root classes in the inheritance hierarchy.

16. Lack of Cohesion in Methods (LCOM)

1. Level. Class.

2. Definition. LCOM measures the number of pairs of methods of a given class where the similarity is zero, subtracted from the number of pairs of methods where the similarity is nonzero. The similarity between the two methods occurs in frequently using variables in an instance of the class [Chidamber and Kemerer, 1994].

3. Formula. If *m(A)* is the number of methods accessing an attribute *A*, calculate the average of *m(A)* for all attributes, subtract the number of methods *m* and divide the result by *(1-m)*. A low value indicates a cohesive class, and a value close to one indicates a lack of cohesion and suggests the class might better split into several (sub)classes [Metrics, 2014].

4. Implication for design. Cohesion facilitates the understanding, reuse, and maintenance of the code [Chidamber and Kemerer, 1994; Fowler *et al.*, 1999; Sommerville, 2012]. LCOM measures the absence of cohesion, so the higher the value of the metric, the less cohesive the class. A high value of LCOM indicates that the class has many different tasks; it is better to use more specific objects, splitting a non-cohesive class into smaller classes with higher cohesion. Alshayeb [2009] has evaluated the impact of refactoring on cohesion metrics, including LCOM. The results show that refactoring leads to better class cohesion. The suggested thresholds, based on the quality standards with which object-oriented software has been developed, indicate the high value of LCOM, allowing the identification of classes that should be refactored to improve the quality of the software.

17. Nested Block Depth (NBD)

1. Level. Method.

2. Definition. NBD measures the maximum depth of nested blocks in the program. Example, three commands nested in a method characterize a maximum block depth of three.

4. The implication for design. According to Sommerville [2012], deeply nested statements are challenging to understand and potentially prone to errors. Therefore, by keeping methods with a low NBD value, they are expected to become more readable and less prone to errors. We have found that the upper bound of the *Good/Common* range for this metric is one. Besides that, when NBD is more significant than three, the method is classified as *Bad/Uncommon*, suggesting that it should be refactored by splitting it into two or more methods.

18. Normalized Distance (RMD)

1. Level. Package.

2. Definition. RMD metric proposes balancing between Instability and Abstraction [Martin, 1994]. Instability (I) is a ratio between the number of efferent couplings and the sum of afferent and efferent couplings in the rated package. In this metric, $I = 0$ indicates the maximum stability, and $I = 1$ is the maximum instability. In other words, the lower the afferent coupling (service delivery) relative to the efferent coupling (service consumption), the closer to 1 will be the measured value, and the package will show greater instability. Abstraction (A) is a ratio proportion between the number of abstract classes and the total number of classes in a package. Packages having abstraction must have outer classes that extend it (afferent coupling). Considering that afferent couplings minimize the instability, it suggests a relationship between the minimization of instability through increasing abstraction. However, Martin [1994] argues for avoiding dependence on unstable modules. Therefore, unstable packages should not be abstract but concrete. Given the concepts of abstraction and instability, Martin [1994] proposes a notion of balancing the values of these metrics. The purpose of this is to discourage dependence on unstable packages. A package may be partially extensible because it is partially abstract. Since the partially stable package, its extensions are not subject to maximum instability. For the ideal relationship between instability and abstraction, packages with a minimum degree of instability should have a maximum degree of abstraction $((I, A) = (0, 1))$ and packages with the maximum degree of instability $((I, A) = (1.0))$. A line between these points may be drawn on a chart. This line, named by the author as Main Sequence, represents the balance between abstraction and instability that is considered acceptable. A package in the Main Sequence is not abstract for its stability nor so unstable for its abstraction; it presents an acceptable degree of balancing concrete and abstract classes concerning their afferent and efferent couplings. For these reasons, the Normalized Distance metric measures the perpendicular distance of (I, A) of the Main Sequence.

3. Formula. *(I) = Ce/(Ca + Ce);*
A = NumberAbstractClasses / TotalNumberClasses
and *RMD = [A + I − 1]*

4. Implication for design. According to the metric definition, a more balanced relationship suggests a reasonable balancing of concrete and abstract classes to their afferent and efferent couplings to obtain a more stable design that is less sensitive to software changes and represents a breakthrough in software quality. The thresholds suggested herein, to accomplish this, can discriminate a high value for this metric. These values may help evaluate the balance of the package. Identifying packages with high values of this metric may be important in the architectural evaluation of the project, seeking to evolve and better balance the division of responsibilities between software packages.

B Inspected Methods

This appendix presents the qualitative evaluation of *each method* considered in Case Study 1 (Section 8.1.1).

addGetterSetterChanges. This method has 34 lines of code, *McCabe* equals nine, five parameters, and a nested block depth equals four. Two of its parameters are of *boolean* type, *usingLocalGetter*, and *usingLocalSetter*. We use these parameters in another two class methods: *checkInHierarchy* and *checkMethodNames*. They are related, so we may replace them with one Parameter Object. If we do that, the method would be classified as *Regular/Casual* because it would have four parameters. The method name refers to the addition of changes in the plural, which suggests the method is responsible for making more than one change related to the addition of *getters* and *setters* for type refactoring *SelfEncapsulatedField*, which is a class responsibility. Besides, the method has two snippets of source code that perform different tasks, which, when exported to two smaller methods, will take it out of the *Bad/Uncommon* range for the four metrics used.

analyseAssignment. It has 89 code lines, *McCabe* equals 36, 5 parameters, and a nested block depth equals four. It has a *switch-case* with two cases where the executed code has more than ten lines and is deeply nested. We may export these parts using two new, more unique methods. It has a series of *ifs* in sequence. Exporting these code snippets to other methods by defining a single responsibility is possible. The method has many *ifs* inside of *loops* that make it difficult to understand the code. The method performs numerous tasks; it is very complex, inducing it to have a poor qualitative evaluation.

diffList. It has 163 lines of code, *McCabe* equals 54, six parameters, and a nested block depth equals eight. It has a *switch-case* with four *cases* in which the executed code has many rows and is deeply nested. Exporting these parts to four new, better-defined, and more straightforward methods is possible. Another critical point regarding this method's quality is that it has five points of return precisely because the method has more than one well-defined responsibility.

drawSubCategoryLabels. This method has 88 lines of code, *McCabe* equals 14, six parameters, and has a nested block depth equal to four. It performs tasks associated with graphic design. It has a series of *ifs*, which verifies whether the worked edge is of the type TOP, BOTTOM, LEFT, or RIGHT that could be replaced by a *switch-case*, making it better in terms of code quality. Analyzing the method, what attracts the most attention is that within these four ifs, it has duplicate code due to the use of different parameters in methods that are called by it. It is possible to export this duplicate code to a new method that receives these parameters according to the type of border. Repeat this situation with four more *ifs* at the end of the method. These two parts of four *ifs* deal with different design-related aspects of the method; they may be exported to new methods better defined and with a more specific responsibility.

findLocalMethods. Has 240 lines of code, *McCabe* equals 86, 18 parameters, and it has a nested block depth equal to seven. The method has related parameters, which we may export to a class; it is large and complex and uses *gotos*. These factors undermine its readability and comprehension.

prepMinion. This method has 287 lines of code, *McCabe* equals 50, five parameters, and a nested block depth equals seven. Going through the method, one realizes it is responsible for making many preparations (called "Minion") related to the database connection command. This method is too long and has many comments, where the largest has more than 30 lines. This type of situation corresponds to *bad smell Comments* **?**, in which it makes attempts to compensate for the difficulty of understanding the code. We could not establish the method's purpose because its name is unclear, and it has many responsibilities.

readAj5ClassAttributes. Has 102 code lines, *McCabe* equals 35, six parameters, and a nested block depth equals six. This method has four large blocks of source code, which could be exported individually to methods that serve a single purpose and help future understanding. Two other methods in the class are similar to this: **readAjd5MethodAttributes** and **readAj5FieldAttributes**. This fact shows that the **AtAjAttributes** class could be sub-split into three classes that deal with attributes of type *class*, *method*, and *field*. Although not directly related to the method level, we observed from the results obtained from this and other inspected methods that a class dealing with more than one responsibility tends to have more complex methods.

REPTree.Tree#buildTree. It has 137 code lines, *McCabe* equals 29, 10 parameters, and nested block depth equals four. We perceive qualitative analysis as a recursive method that generates a data structure in the tree format. It has several return points, which makes it difficult to understand. From the code's comments, this method performs several tasks separated by more than one space. The idea is to export each stretch to a method that performs a single task.

startFileInternal. It has 90 code lines, *McCabe* equals 21, nine parameters, and has a nested block depth equal to four. Despite what the method name suggests, it accomplishes much more than initializing a file internally, although the tasks seem related. The idea is to extract the responsibilities for private methods to facilitate understanding the code and reduce its complexity.

visitBranchInstruction. It has 31 code lines, *McCabe* equals five, five parameters, and a nested block depth equals four. It is precisely within the lower limit of the thresholds for the four metrics. So, a subtle difference exists between being evaluated as *Bad/Uncommon* by thresholds or evaluated as *Regular/Casual*. Manual inspection has disclosed a relatively acceptable method. It is possible to notice interest outside of its context related to the *log* register that appears inside an if, which checks whether the application is in *DEBUG* mode. The method's purpose is not to log a common interest spread by the application. A correct way to proceed with logging would be to call a class whose specific purpose is this. If we remove the *if* from the code by triggering a specific method of a class responsible for *log* registration, MLOC, NBD, and VG metrics, we would have ranked it in the *Regular/Casual*. We also observe that all the class methods use three of five parameters. This fact suggests a relationship between these parameters that may constitute a new system class.

C Inspected Classes

This appendix presents the qualitative evaluation of *each class* considered in Case Study 1 (Section 8.1.2).

CloneableEditor. It does not have a well-defined goal. There are parts of *log* code, file manipulation, and graphics component manipulation. It deals with the interaction with the user and a series of manipulations, having many responsibilities. To understand this class, it is necessary to refactor it to more specific classes.

CompilationUnitEditor. It has a depth of inheritance tree DIT equal to 8, and consequently, the project becomes more complicated due to the arduous task of understanding this hierarchy. Besides the complexity inherent in the class size, it inherits more methods and becomes even more complex, making it difficult to describe a well-defined role. This fact strongly suggests that the inheritance hierarchy may be wrong. There are so many methods in the class that it is hard to establish a specific purpose that agrees with the cohesion deficit shown by the reference value. The class seems to be difficult to reuse and maintain.

CtxHelpTreesubsection. This class has a depth of inheritance tree DIT equal to six, a complex inheritance hierarchy. It has a set of interrelated constants, and it would be better to replace it with an enumeration structure (*enum*). Although the class has mostly straightforward methods, they lack a relationship between them. Maybe grouping them into more cohesive classes with specific roles will better describe the project and facilitate understanding and maintenance. The class is a candidate for refactoring because it performs tasks that should be done by more than one class.

DependencyManagementsubsection. It has a high DIT value and, in addition to implementing many tasks, has many methods, which results in a complex class. It also has a series of constants better placed in an *enum*. We observe that the class treats a series of roles. Make the program more readable, and likely the most straightforward maintenance; placing them in more cohesive classes would be better.

DeploymentDisplay. It displays a Swing visual component. The code should be in a control layer, supposing an MVC architectural standard. An example of better componentization would be, for instance, the forward and back *JButton* being a specific navigation component. Another essential point is that this class has six methods that show duplicate code, which could be in a single method with the proper parameters.

GTKFileChooserUI. It is a huge class that could be splitted into more classes. It has many attributes, shown in more cohesive subgroups within the class code. There is no space between the more cohesive groups of attributes, and when the group changes, there is a series of break lines that suggest a new group of more related attributes. Four attributes are related to creating a new folder, which could be in a class for a new folder creation action, along with the associated behaviors. The programmer is aware of a more cohesive relationship between the declared attributes. He sees it through a grouping in the class, which is not ideal considering software quality. The class is large and complex, and it isn't easy to understand its specific purpose. There are even empty methods. It could be a better user interface class.

InfoProduct. This class is against the basic principles of software architecture because it contains the same class code referring to view, control, and access database, even with attributes that are string with SQL commands. This class includes screen configurations and does not compose it well within the view layer.

JarPackageWizardPage. It is a view component that represents the first page of the *JAR* export option in *Eclipse*. It has many attributes, mainly type *Button*. One perceives a deteriorated quality standard in the classes of *Eclipse* responsible for the screens, visible as poorly componentized. For instance, in this class, attributes related to export options and treatments of that data could be componentized in another class. Due to the lack of componentization, these classes become complex, with many attributes, constants, and specialized behaviors. Moreover, due to this complexity, the class is neither qualitatively nor quantitatively cohesive.

JoinNode. The class's comments suggest that it is not cohesive. The comment says that it represents a join result for *SELECT* and *INSERT-SELECT* for data manipulation type operations. It has two responsibilities, even at the project level. The code has a long and complex class with many attributes and constants. Constants should be in an *enum*. It seems to have a bad smell *Comments* because it presents disproportionate comments to the commented sections, showing a lack of legibility that tries to be compensated with an extensive comment. We evaluate this class with low cohesion, complex, and should be a candidate for refactoring.

JRCTXVisualView. This class is associated with a system screen. It has a class of *controller* associated with it and what seems to be some attributes that are sub-components of the screen in question. The class is big, and it is not easy to establish a specific purpose for the screen type. The comment does not contain meaningful information: *Top component displays something*. The class has many attributes and methods handled in isolation, indicating low cohesion. So, after analyzing the class, we recommend it as a candidate for refactoring.

JSVGCanvas. It is a seemingly well-programmed class with short lines that allow for proper viewing and concise comments. Their methods are short. But, it is a massive class with a general purpose of defining a *Swing* component due to the comment for the class. There is insufficient componentization for the class to present clarity and uniqueness within the project. One may componentize the actions and items on the screen in less complicated classes with a definite purpose.

JTitledPanel. It is a view class that exposes a title-related panel or the upper region of a screen. This class has three inner classes that undermine its evaluation. It visibly needs improvement, with the division of responsibilities into other classes with better cohesion. There are two JButtons for minimization and maximization and several methods of manipulating actions on them. We may export this responsibility to a new class, which possesses cohesion and uniqueness concerning this type of action. Despite this, the class shows lean and well-defined methods, and a better definition of individual class responsibilities is needed.

SAX2DTM2 (jre-1.6.0/xalan-2.7.1). This class implements a Document Table Model (DTM) for XML conversion to a table structure using SAX2 (Simple API for XML). This class seems to have *bad smell comments*, as the comments are not concise. There is more than one commented attribute with the comment of more than ten lines, as well as lines of code for overly commented methods. The class exposes many methods, which show many ties and control structures. It has some empty methods, without any implementation, publicly exposed. Despite a well-defined purpose, the practice does not show that the class is cohesive and should be a candidate for refactoring by exporting attributes and methods to other classes that perform specificities related to the conversion. We recommend implementing the code referring to log present in a specific class. Also, it is possible to see duplicate code in this class. It extends SAX2DTM by deploying conversion-related optimizations and overwriting operations that may be performed more efficiently. Also, it is possible to see duplicate code in this class. The class commentary references the limitation of the extensibility of the class due to the manipulation of direct aspects related to performance, which may impact its internal and external quality aspects, such as extensibility. The comments in the class expose its fragility, as they suggest that one should be very careful in changing it. All this indicates that it is a challenging class to modify, understand, and reuse, with low attributes on the internal quality that may directly impact its attributes of external quality.

SimpleCSMasterTreesubsection. When initializing the class inspection, we observed an extensive list of constants that impair the evaluation. They better group into an *enum*, whose name could also tell more about its usage. This class has attributes of type Action, indicating the separation of actions into objects that treat them more singularly. This fact did not occur in the other analyzed class of the same software: DependencyManagementsubsection. Although this has happened, the class handles several actions that are better grouped into different classes.

SVGFlowRootElementBridge. It has many constants, which should be in an *enum* type structure, better modularizing the code and making it more readable. It has many methods, many overwritten, some very long, and different purposes. It is better to group the methods into smaller, more cohesive classes.

D Metrics Description

Metric	Description
AID	Average inheritance depth of a class
AMC	Average Method Complexity
ATFD	Access To Foreign Data
AvgLineCode	average size of methods in a class in lines of code
AvgCyclomatic	the average value of cyclomatic complexity in the methods in a class
AvgCC	Average Cyclomatic Complexity [McCabe 1976]
ACAIC	[Briand et al.1997]
ACMIC	[Briand et al.1997]
AMMIC	[Briand et al.1997]
CA	Afferent Coupling [Martin 1994]
CAM	Cohesion Among Methods
CAMC	Cohesion among methods of classes
CBO	Coupling Between Object
CE	Efferent Coupling [Martin 1994]
CFC	Complexity Metric
CC	Cyclomatic Complexity
CC	Changing Classes [Marinescu 2002]
CYCLO	Cyclomatic Complexity
COF	Coupling Factor
CINT	Coupling Intensity
CM	Changing Methods
Coh	Variation of LCOM5 [Briand98]
CNR	Number of Constant Refinements
CTA	Coupling through data abstraction
CBI	Coupling based on inheritance
CLD	Class-to-leaf depth
DAC	Data abstraction coupling
DAM	Data Access Metric
DCC	Direct Class Coupling
DCd	Same as TCC
DCi	Same as LCC
DIT	Depth of Inheritance Tree
DP	Dynamic polymorphism in inheritance relations
DMMEC	[Briand et al.1997]
DPA	Dynamic polymorphism in ancestors
DPD	Dynamic polymorphism in descendants
Export	every exported event
EncF	Encapsulation Factor
FAN-IN	
FUN-OUT	number of other classes referenced by a class
ICH	Information-flow-based cohesion
ICP	Information-flow-based coupling
IHICP	inheritance-based coupling only
LCC	Loose Class Cohesion [Bieman95]
TCC	Tight Class Cohesion [Bieman95]
LCOM	Lack of Cohesion in methods
LCOM1	Lack of cohesion in methods [Chidamber91]
LCOM2	[Chidamber94]
LCOM3	number of disjoint set of methods [Hitz95]
LCOM4	[Hitz95]
LCOM5	Lack of Cohesion of Method [Henderson-Selles96]
NewLCOM5	A variation on LCOM5
LOC	lines of Code
MLOC	Lines of Code per Method
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
MaCabe	evaluate the complexity of a program
maxCC	Maximum Cyclomatic Complexity
MPC	Message passing coupling
NFC	Number of Function Call
NOF	Number of Attributes
NOM	Number of Methods
NORM	Number of Overridden Methods
NSC	Number of Daughters
NSF	Number of Static Attributes
NSM	Number of Static Methods
NST	Number of Statments
NBD	Nested Block Depth
NCR	Number of Constant Refinements
NIM	Number of Instance Method
NOC	Number of Children
NMR	Number of Method
NOA	Number of ancestors
NPM	Number of Public Methods [Bansiya & Dave 2002]
NOV	Number of Variables in a class
NIHICP	counts invocations to classes not related through inheritance
NPF	Number of Public Fields
NOAP	Number of Public Attributes
NOAM	Number of Added Methods
NOOM	Number of Overridden Methods
N00	Number of Operations
NOAV	Number of Accessed Variables
NLM	Number of Local Methods
NPRM	Number of Private Methods
NPROM	Number of Protected Methods
NHD	Normalized Hamming distance-based
NMA	Number of methods added
NMI	Number of methods inherited
NMO	Number of methods overridden
NOD	Number of descendants
NOP	Number of parents
NA	number of attributes in a class
NAIMP	number of attributes in in a class excluding inherited
NM	number of methods in a class
NMIMP	number of methods implemented in a class

Metric	Description
NumPara	sum of the number of parameters of the methods implemented
OCAEC	[Briand et al.1997]
OCAIC	[Briand et al.1997]
OCMEC	[Briand et al.1997]
OCMIC	[Briand et al.1997]
OMMEC	[Briand et al.1997]
PAR	Number of Parameters
PCC	Same as OCC, except G is a directed graph
Puf	Public Factor
RMD	Normalized Distance
RFC	Response For a Class
RFP	Response For Process
SNHD	Scaled NHD
SLOC	Source Line of Code
SIX	Specialization Index
SP	Static polymorphism in inheritance relations
SPA	Static polymorphism in anscestors
SPD	Static polymorphism in descendants
Stmts	number of declaration and executable statements in the methods of a class
TNCt	Total Number of Constants
TNR	Total Number of Refinements
TNMR	Total Number of Method
TNRC	Total Number of Refined
TNRM	Total Number of Refined Methods
TNG	Total Number of Gateway
TNE	Total Number of Events
VG	Cyclomatic Number McCabe Complexity
WMC	Weighted Methods by Class
Co	Connectivity
NewCo:	A variation on Co