A Catalogue of Thresholds for Object-Oriented Software Metrics

Tarcísio G. S. Filó and Mariza A. S. Bigonha

Department of Computer Science (DCC)
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Minas Gerais, Brazil
e-mail: {tfilo,mariza}@dcc.ufmg.br

Kecia A. M. Ferreira

Department of Computing (DECOM)
Federal Center for Technological Education (CEFET-MG)
Belo Horizonte, Minas Gerais, Brazil
e-mail: kecia@decom.cefetmg.br

***Abstract*—Thresholds for the majority of software metrics are still not known. This might be the reason why a measurement method that should be part of a software quality assessment process is not yet present in object-oriented software industry. In this work, we applied an empirical method to 111 system dataset, identifying thresholds for 17 object-oriented software metrics. Furthermore, we propose some improvements in this employed method. Differently from previous work, we have developed a catalogue of thresholds that gathers a greater amount of object-oriented software metrics, allowing the assessment of methods, classes and packages. Our approach suggests three ranges in the thresholds: Good/Common, Regular/Casual and Bad/Uncommon. Although they do not necessarily express the best practices in Software Engineering, they reflect a quality standard followed by most of the evaluated software. To evaluate our catalogue, we present a case study which shows its application in the evaluation of a proprietary software, in contrast with the developers consensus about its internal quality. Results show that our thresholds are capable of indicating the real panorama of the evaluated software.**

Keywords–Software Engineering; Object-oriented programming; Quality analysis and evaluation; Metrics/Measurement.

I. INTRODUCTION

Measurement is considered a fundamental part of any engineering discipline, and Software Engineering disciplines are not exceptions. In this context, software metrics refer to measurements that can be applied to check the indicators of processes, projects and software products. Evaluating software quality through measurements allows to define quantitatively the success or failure of a particular attribute, identifying the need of improvement. Managing software quality may allow to achieve a low number of defects and reliable standards of maintainability, reliability, and portability [1].

Despite the importance of metrics in object-oriented software quality management, they have not been effectively used in software industry [2][3]. One possible reason is the fact that for the majority of metrics, thresholds are not defined. Knowing these values is essential because they may allow the metrics to be used to the assessment of software quality. Moreover, without the knowledge of these thresholds, we cannot answer simple questions like "Which classes in the system have a large number of methods?" or "Which methods in the system have a large number of parameters?".

In the current scenario of Software Engineering, the internal quality of software is usually evaluated by means of qualitative inspections, which takes time and generates high costs. The use of metrics in conjunction with a catalogue of thresholds empirically derived may provide an efficient approach to assess the software quality in an automated way.

Our major contribution is a catalogue of thresholds for 17 object-oriented software metrics, which covers a larger amount of metrics, providing the assessment of methods, classes and packages. Even though previous researches have proposed different techniques to derive thresholds for software metrics, most of them cover only few metrics [3]–[6]. In such a scenario, we do not aim to propose a new method to derive thresholds. Instead, we employed the empirical method proposed by Ferreira et al. [3], which is based on the analysis of the statistical distribution of the measures found in practice. Moreover, we introduce some improvements in this method. When we compare the contributions of this paper with the results presented by Ferreira et al. [3] and other previous studies, we can spot three majors differences. (1) We provide thresholds for a large number of software metrics. (2) The proposed thresholds aim to provide a benchmark for the quantitative evaluation of the internal quality of software systems, considering not only classes, but also methods and packages. (3) Differently from previous work, we evaluate our catalogue of thresholds in a proprietary software, of considerable size, supported by the qualitative definitions about the aspects of its internal quality reported by the developers themselves, extending the thresholds evaluation to outside of the open-source universe.

The remaining of this paper is organized as follows: Section II presents the data collection, a set of systems, as well as the preparation of this data to be used in the proposed thresholds. Section III describes the employed method to extract thresholds, followed by illustrative examples presented in Section IV. Section V presents the results of this research, showing a catalogue of the identified thresholds. Section 6 relates a case study conduct in a proprietary software to verify the effectiveness of applying our catalogue in software quality management. In Section VII, we discuss how our work is related with existing efforts in the literature. Section VIII discusses threats to validity. Section IX presents possible future directions of this research and makes final remarks.

II. DATA COLLECTION

This section describes Qualitas.*class* Corpus [7], which is the set of systems used in this research, as well as the data preparation and the generation of statistical data on metrics necessary to the development of this work. Qualitas.*class* Corpus [7] provides compiled Eclipse Java projects for the 111 systems included in Qualitas Corpus, provided by Tempero et

al. [8]. Qualitas.*class* Corpus relied on Metrics 1.3.8 [9], which contains implementation details about the metrics, to compute their values, providing XML files with values of 17 metrics:

Basic Metrics: no. of classes (NOC), no. of methods (NOM), no. of fields (NOF), no. of overriden methods (NORM), no. of parameters (PAR), no. of static methods (NSM) and no. of static fields (NSF).

Complexity metrics: method lines of code (MLOC), specialization index (SIX), *McCabe* cyclomatic complexity (VG) and nested block depth (NBD).

CK metrics: weighted methods per class (WMC), depth of inheritance tree (DIT), no. of children (NSC) and lack of cohesion in methods (LCOM).

Coupling metrics: afferent/efferent coupling (CA/CE).

To read the available XML files at Qualitas.*class* Corpus, it was developed a tool that generates text files containing all the measurements for each metric. We relied on R [10], a tool for statistical computing, to generate the cumulative relative frequency graph, which gives the summary of the frequency below a given level, as well as on the histogram in logarithmic scale, which plots the histogram in double logarithm scale. We also used this tool to generate an statistical dataset on object-oriented software metrics [11], intending to help researchers in their work on software metrics.

III. Method to Identify Thresholds

An important problem in statistics is how to obtain information about the form of the population from which a sample is drawn. For this purpose, it is used EasyFit [12] to perform the selection of the appropriate distribution that has a best fit for a dataset. Besides that, EasyFit plots the *pdf* (probability density function) graph which describes the probability of a variable assuming a value x: $f(x) = p(X = x)$. The purpose of this step is to set the appropriate distribution for each software metric studied. Exploring the distributions of software metric values is crucial to improve the understanding of the internal structures of software [13]. From the distribution, it is possible to understand its characteristics, for example, if its average value is representative for the analysis or if the distribution is heavy-tailed or skewed-right [3][14].

Given the graphical views and the knowledge of the characteristics of the probability distributions which are best fitted to the measures, it is possible to derive thresholds for the metrics. In the approach of Ferreira et al. [3], when the metric has a distribution with a representative average value, like the *Poisson* distribution, this value is taken as typical for this metric, otherwise, the authors worked with three ranges for the metric values: *Good*, *Regular* and *Bad*. The *good* range corresponds to values with high frequency. The authors argue this is the most common values of the metric in practice, and nevertheless these values do not necessarily express the best practices in Software Engineering, they expose the pattern of most software systems. The *bad* range corresponds to values with quite low frequency, and the *regular* range corresponds to values that are not too frequent neither have very low frequency. The visual analysis of the graphical views allows establishing the thresholds. It is important to notice that Ferreira et al. applied this method to all set of software systems, and also group them by application domain, size and type, but they did not find relevant differences in the suggest thresholds among these approaches. So, in accordance with these results, we applied the method to the entire set of systems, expecting that the suggested thresholds are useful for all systems, regardless of the of application domain, size and type.

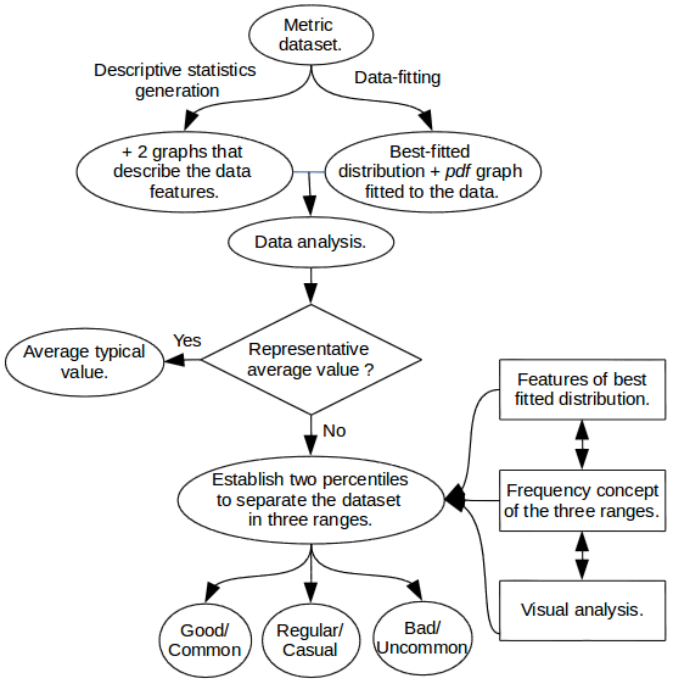


Figure 1. Flow diagram to the thresholds identification.

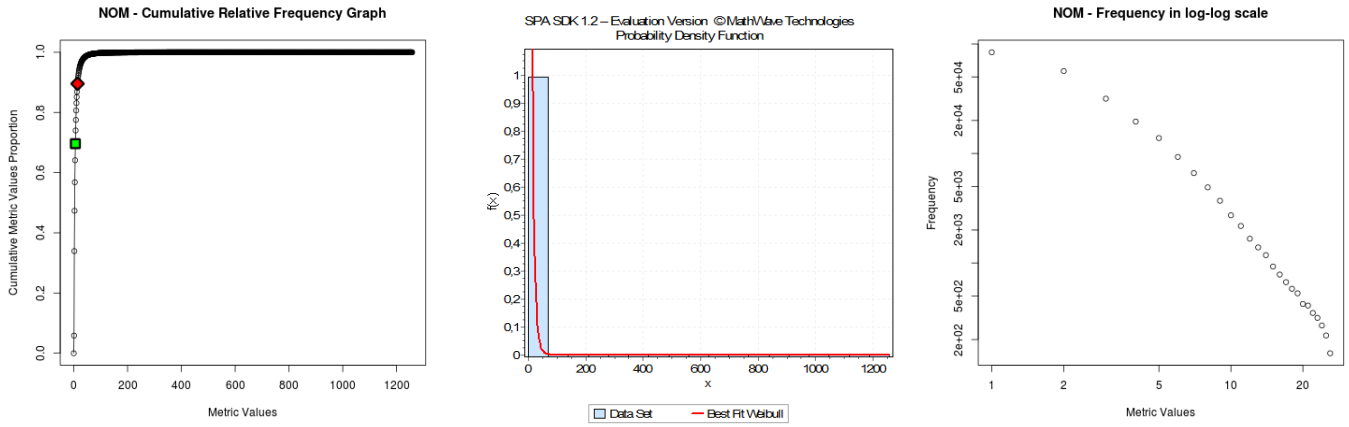
This paper also presents the proposed improvements to the original method of Ferreira et al. [3]. First, we modify the ranges names to: *Good/Common*, *Regular/Casual* and *Bad/Uncommon*, which we believe will express better the importance of frequency concept in the suggested thresholds. Secondly, we established, rather than the values directly, two percentiles, based on a visual analysis of the graphical views and on the frequency concept in the thresholds. These percentiles are capable to separate the dataset in the three ranges of values mentioned. Although the visual analysis is not dispensed, the use of predefined percentiles brings a relevant improvement to the method, it allows to obtain the values directly from the dataset, making the application of the method more reproducible. Figure 1 summarizes the method to identify thresholds in a flow diagram.

IV. Illustrative Examples

This section describes the data analysis of *Number of Methods (NOM)* and *Depth of Inheritance Tree (DIT)*.

A. Number of Methods

The Cumulative Relative Frequency Graph showed in Figure 2a suggests a heavy-tail distribution, because the approximation of 100% of 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 means that the systems under analysis possess several classes with few methods and a small number of classes with many methods. Figure 2b shows that the dataset of NOM is best fitted to *Weibull* distribution, with parameters



(a) (b) (c)

Figure 2. NOM: (a) Cumulative Relative Frequency Graph (b) *pdf* fitted to the Weibull (c) Histogram *log-log* scale.

$\alpha = 0,852$ and $\beta = 5,879$. As the shape parameter α is less than one, Weibull is a heavy-tailed distribution. If this is the case, the sample mean and variance cannot be used as estimators of the population because the central limit theorem does not apply, which would mean that basing any conclusions on sample means without fully understanding the distribution would be questionable at best [13]. So, the mean value is not representative. Figure 2c exhibits the dataset in *log-log* scale. In this graph, it is noticed a straight leaning to the left, a power law feature [3][13]. This pattern enhances the features already mentioned, the majority of classes has few methods and the mean is not representative. As this metric does not have a value which may be taken as typical, we identified the values representing the 70° and 90° percentiles of the dataset, which correspond to the values 6 and 14. The 70° and 90° percentiles were chosen by a visual analysis of the Cumulative Relative Frequency Graph showed in Figure 2a. The points marked with green color/square shape and red color/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. Thus: (1) based on visual analysis, (2) the established concept for the ranges, and (3) inspired by the work of Alves et al. [5] — who used percentiles to statistically part quality metric profiles in thresholds identification — we tried to apply the 70° and 90° percentiles in most of the metrics in order to identify the measures able to separate the three suggested ranges. We found up there are variations that go in accordance with the distribution curve features, when they are taller or flattened, or depending on a higher or lower metric value (x axis) to reach a greater 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 are able to separate the three suggested ranges. The values 6 and 14 allow us to separate number of methods metric in three ranges: *Good/Common* ($NOM \leq 6$), *Regular/Casual* ($6 < NOM \leq 14$) and *Bad/Uncommon* ($NOM > 14$).

B. Depth of Inheritance Tree

The data of DIT do not suggest a heavy-tailed distribution, but a right-skewed one, as showed by the Cumulative Relative Frequency Graph in Figure 3a. According to Figure 3b, the dataset is best fitted to the *Gumbel Max* distribution, with parameters $\alpha = 2.170$ and $\beta = 1,469$. By the adjustment line of the data to the distribution, it is possible to identify the right-skewed feature. In this kind of distribution, the mean value is not representative. Figure 3c shows the dataset in a *log-log* scale. In this graph, it is not noticed a straight leaning to the left, which does not suggest a power law. We have identified the values that represent the 70° and 90° percentiles of the dataset, which correspond to the values 2 and 4. The 70° and 90° percentiles were identified by a visual analysis of the Cumulative Relative Frequency Graph showed in Figure 3a. So, the identified threshold for this metric is: *Good/Common* ($DIT \leq 2$), *Regular/Casual* ($2 < DIT \leq 4$) and *Bad/Uncommon* ($DIT > 4$).

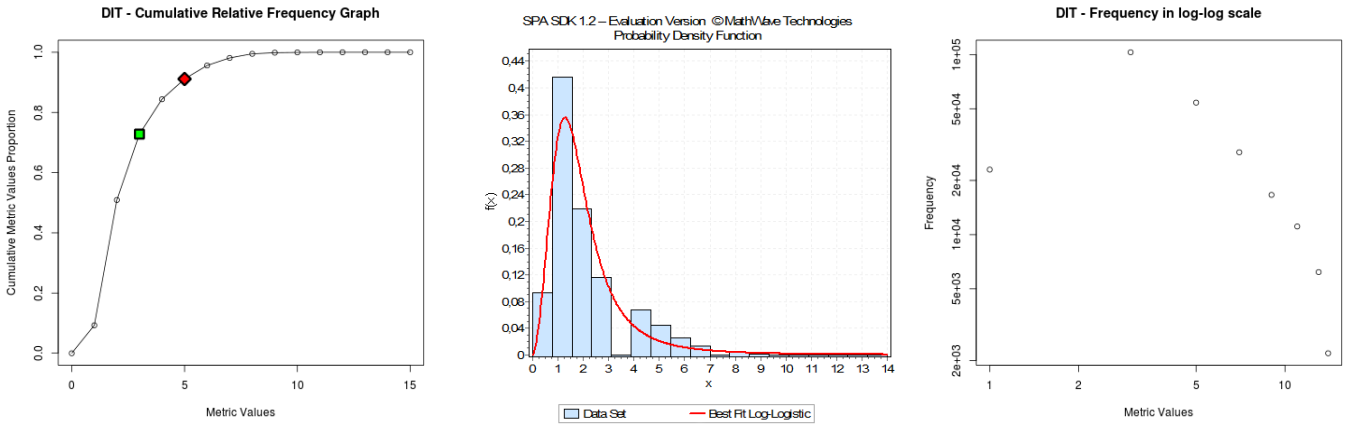
V. Results

Table I presents the identified thresholds for each metric analysed. Table II shows the best fitted distributions for each metric, with their parameters. Our catalogue of thresholds reflects a pattern followed by most of the software systems in Qualitas.*class* Corpus, which may be useful in some scenarios of Software Engineering.

TABLE I. IDENTIFIED THRESHOLDS.

Metric	Good/Common	Regular/Casual	Bad/Uncommon
CA	$m \leq 7$	$7 < m \leq 39$	$m > 39$
CE	$m \leq 6$	$6 < m \leq 16$	$m > 16$
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	$m \leq 2$	$2 < m \leq 4$	$m > 4$
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	$m \leq 11$	$11 < m \leq 34$	$m > 34$

The identification of anomalous measurements is seen as a part of the measurement process that may be part of a software



(a) (b) (c)

Figure 3. DIT: (a) Cumulative Relative Frequency Graph (b) *pdf* fitted to the Gumbel Max (c) Histogram *log-log* scale.

TABLE II. BEST FITTED DISTRIBUTIONS FOR THE METRICS.

Metric	Distributition	Parameters
CA	*Gen. Extreme Value*	$k = 0.797$, $\sigma = 4.303$, $\mu = 1.775$
CE	*Gen. Extreme Value*	$k = 0.527$, $\sigma = 2.834$, $\mu = 2.397$
DIT	*Log-Logistic*	$\alpha = 2.170$, $\beta = 1,469$
LCOM	*Beta*	$\alpha_1 = 0.043$, $\alpha_2 = 6.777$, $b = 8.290$
MLOC	*Pareto 2*	$\alpha = 1.226$, $\beta = 3.051$
NBD	*Gumbel Max*	$\sigma = 0.858$, $\mu = 0.931$
NOC	*Log-Logistic*	$\alpha = 1.452$, $\beta = 5.520$
NOF	*Beta*	$\alpha_1 = 0.059$, $\alpha_2 = 64.580$, $b = 2026.446$
NOM	*Weibull*	$\alpha = 0.852$, $\beta = 5.879$
NORM	*Power Function*	$\alpha = 0.006$, $a = 0$, $b = 235.200$
NSC	*Beta*	$\alpha_1 = 0.001$, $\alpha_2 = 9.467$, $b = 25465.529$
NSF	*Beta*	$\alpha_1 = 0.018$, $\alpha_2 = 18.284$, $b = 4130.615$
NSM	*Beta*	$\alpha_1 = 0.017$, $\alpha_2 = 27.961$, $b = 1646.287$
PAR	*Gumbel Max*	$\sigma = 0.973$, $\mu = 0.399$
SIX	*Power Function*	$\alpha = 0.031$, $a = 0$, $b = 24.578$
VG	*Chi-Squared*	$\nu = 1.000$, $\gamma = 1.000$
WMC	*Log-Logistic*	$\alpha = 1.142$, $\beta = 4.687$, $\gamma = 0.000$

quality assessment. After the measurements have been made, developers should compare them with previous ones, looking for unusually high values [1]. The identified thresholds may be useful in this identification, as they provide a way to do this comparison with the quality that is common in software development. But, an anomalous measure does not necessarily mean a problem, it suggests that there might be problems. Once the artifacts are quantitatively identified, one must inspect them to decide if the anomalous metric measures mean that the software quality is compromised [1]. Other scenario is the application of the thresholds in a filtering mechanism to reduce the dataset in a *bad smell* detection strategy [15].

VI. Evaluation

For the evaluation of our catalogue, we conducted a case study which evaluated a proprietary software from a public organization with a deteriorated internal quality, in order to verify the ability of the proposed thresholds in indicating this panorama. For privacy reasons, we call it XYZ. This study was divided into 3 parts, aiming to evaluate, respectively, the metrics of methods and classes, and the correlation of bad smells occurrence with our thresholds evaluation. At the end of this section, we present a qualitative analysis of the utility of the identified thresholds for the metrics which were not applied in the evaluation of XYZ.

XYZ is considered a successful software by its users and stakeholders. However, there is consensus on the team that led its development and now leads the maintenance and evolution that, actually, it has a bad internal quality. XYZ has 54,297 TLOC, 2,532 methods, 603 classes and 139 packages.

From its deployment in 2009, there was no execution of preventive maintenance, neither refactorings to improve its internal quality. During the last years, XYZ has been suffering constant maintenance, such as: fixing runtime errors or system requirements, adding new features or modifying the existing ones, improving the processing speed of its functionalities and adjusting the code to changes in the environment.

As XYZ is constantly changing, it is natural that their internal structures become more complex [16]. Aiming to compare the information about the aspects involving the maintenance and evolution of XYZ with the view of the programmers, we collected opinions of four members of the development team about its internal quality and the reasons why they think the software is in this situation. There were no forms or specific directions, we chose to let opinions to flow naturally in order to characterize the software quality from their qualitative view. Analysing the reports, we noticed a consensus that XYZ has a deteriorated internal quality. The factors cited as the root of this problem are: the lack of adoption of methodologies to systematize maintenance, lack of a software architect and ineffective requirements.

A. Part 1 - Evaluation of Methods

The purpose of Part 1 is to check if XYZ has relatively more methods classified as *Bad/Uncommon* and less methods classified as *Good/Common* than most existing software of the sample, Qualitas.*class* Corpus, what would be in conformance to the qualitative consensus of its low quality.

To do that, we measured the percentage of methods classified as *Good/Common*, *Regular/Casual* and *Bad/Uncommon* by the metrics of methods of 111 system of Qualitas.*class* Corpus, i.e., one of the software in the dataset, hsqldb, has 73.55% of its methods classified as *Good/Common*, 17.69% as *Regular/Casual* and 8.76% as *Bad/Uncommon* by MLOC metric. Subsequently, the systems were ordered as follows: in ascending order by the percentage of methods classified as

Bad/Uncommon and in descending order by the percentage of methods classified as *Good/Common*. Then, we got the positions of XYZ in the obtained rankings. A low position at the first ranking means that, relatively, XYZ shows a higher proportion of methods classified as *Bad/Uncommon* than most of the analysed software. A low position at the second ranking suggests that few methods of XYZ are well evaluated by the suggested thresholds than other software in the sample. As the software has poor quality, this would suggest a correct evaluation, i.e., the software is notoriously bad and its methods were evaluated as *Bad/Uncommon*. The evaluated metrics were:

Method Lines of Code (MLOC): in the first proposed ranking, XYZ was at $100\,^{\circ}$ position. So, it was no worse than 12 of the 111 systems dataset. By the second one, XYZ was at $104\,^{\circ}$ position, not being worse than 8 other systems.

Nested Block Depth (NBD): in the first ranking, XYZ was at $92\,^{\circ}$ position. In the second one at $101\,^{\circ}$ position.

Number of Parameters (PAR): in the first one, XYZ was at $107\,^{\circ}$ position. In the second one, at $105\,^{\circ}$ position.

***McCabe* cyclomatic complexity (VG):** in both rankings, XYZ was at $103\,^{\circ}$, not worse than just 9 other systems.

1) Conclusion: The data show that XYZ has, relatively, fewer methods evaluated as *Good/Common* than the vast majority of the systems present in the Qualitas.*class* Corpus. This is in agreement with the established qualitative scenario about the low quality of this system, i.e., the proposed thresholds for method metrics were able to reflect, quantitatively, the scenario of low quality of this system. In contrast, XYZ has, relatively, a high number of methods evaluated as *Bad/Uncommon* than other software in the dataset. This result suggests that the proposed thresholds to metrics that evaluate methods will not show quality where there are problems.

B. Part 2 - Evaluation of Classes

In this part of the case study, we evaluated a sample of low quality classes defined by the development team of XYZ with respect to the obtained classifications with the identified thresholds for metrics of classes. Table III presents the sample of classes, with fictitious names, and the obtained classifications, where $+1$ means *Good/Common*, -1 means *Bad/Uncommon* and 0 means *Regular/Casual*. Next we analyze the results of each metric.

TABLE III. SAMPLE OF LOW QUALITY CLASSES OF XYZ.

	NOF	DIT	WMC	NSC	NORM	LCOM	NOM	SIX
LSI	+1	+1	-1	+1	+1	-1	-1	+1
NSI	+1	+1	-1	+1	+1	-1	-1	+1
RSI	+1	+1	-1	+1	+1	-1	-1	+1
NB	-1	0	-1	+1	-1	-1	-1	-1
NTB	-1	+1	-1	+1	+1	-1	-1	+1
RB	-1	+1	-1	+1	+1	-1	-1	+1
ND	+1	0	-1	+1	+1	+1	-1	+1
LD	+1	0	-1	+1	+1	+1	-1	+1

Number of Fields (NOF): the NOF column of Table III shows that 5 classes were well evaluated and 3 classes were poorly evaluated by the threshold of NOF. As these classes are defined qualitatively as problematic, they were inspected in order to identify their features across to the received quantitative evaluation. LSI, NSI and RSI are *service* classes (fictitious class names). According to Fowler [17], an anemic domain model occurs when the business logic is not put in the domain objects. Instead, there are a number of *service* objects that capture this logic. When pulling behaviors into *services*, they become *Transaction Scripts*, which organize the business logic by procedures that treat requests from the view layer. These *services* are at the top of the domain model and use the model as a data repository. Then, the domain objects become "bags of getters and setters", not encapsulating the logic for the data. The *services* are a grouping of procedural functions related to domain data. Thus, anemic objects do not have behaviors and *services* are grouping of procedural functions. Therefore, these classes were classified as *Good/Common* by the identified threshold for NOF, not because they have a reasonable amount of fields, but for not having fields at all. Therefore, by using only NOF, these classes would be well evaluated, despite being problematic within the project, providing potential errors in the quantitative evaluation. NDAO and LDAO are objects of type DAO (*Data Access Object*), which encapsulate all data access logic within an application. These are classes that, by their goal within the system architecture, have few attributes, without characterizing an architectural violation or a bad programming practice, as occurs with the anemic domain model. These classes are problematic because of their size and complexity, and not by their number of fields. Thus, the classes were correctly well evaluated in relation to the NOF threshold. NB, NTB and RB are objects of type *managed bean*, which were poorly evaluated by the identified threshold for NOF. This type of class is typical in *JavaServer Faces* applications, where each *managed bean* should be associated to one or more components of a web-page [18]. These classes are large and complex because they are not being componentized in *managed beans* more cohesive, which meet a specific purpose within the web-page. So, the qualitative assessment of these classes match the result of applying the identified threshold for NOF, both suggesting that the classes show poor quality.

Depth of Inheritance in Tree (DIT): the DIT column of Table III shows that three classes were evaluated as *Regular/Casual* and the rest of them were evaluated as *Good/Common*. Inheritance is a resource rarely used in this system, mainly because of the anemic domain model. Thus, the vast majority of classes do not use this resource and, consequently, do not have a deep inheritance tree. Therefore, they were evaluated as *Good/Common* by the DIT threshold. In fact, there are no problems in these classes related to the depth of inheritance tree. So, these cases are not considered incorrect evaluations, on the contrary, the identified threshold of DIT correctly evaluated these classes as *Good/Common*. It is necessary to understand correctly what is being evaluated with the metric to perform a correct interpretation of the obtained results. We must remember that a positive rating by the DIT threshold in most classes of the system does not mean that the design is making an appropriate use of inheritance, because inheritance can not even being used. The classes that were evaluated as *Regular/Casual*, ND and LD have an inheritance hierarchy imposed by the used persistence framework. In a qualitative evaluation, it was concluded that the DIT threshold evaluated these classes correctly, as they are not impossible to be understood neither are of easy understanding.

Weighted Methods per Class (WMC): by WMC column of Table III, we observe that all classes were evaluated as

Bad/Uncommon by the identified threshold. WMC is a measure of the class complexity and taking into account the characteristics of XYZ and its evolutionary process, our threshold is capable of showing the natural increase of complexity related by Lehman [16] in a quantitative way. These classes are really complex and they assumed many responsibilities throughout the software evolution, becoming hard to understand and maintain.

Number of Children (NSC): the NSC column of Table III shows that all classes were evaluated as *Good/Common* by the NSC threshold. Indeed, these are classes that do not have children, not showing problems related to the evaluated aspects of this metric.

Number of Overriden Methods (NORM): the NORM column of Table III shows that all classes were well evaluated by the identified threshold, except for NB class. Indeed, by the low utilization of inheritance resource shown by the results of DIT and the qualitative analysis of the software, this result was expected, after all, the classes have no problems related to excessive overwriting of methods and, therefore, were correctly evaluated well. For the NB class, which is poorly evaluated, we can see that this class was one of the three classes that were classified as *Regular/Casual* by the DIT threshold, indicating a depth of inheritance tree outside the ideal and the next to be considered inappropriate. NB overwrites 7 methods, an amount considered high by the threshold. The major problem of this quantity of overwritten methods is that the class becomes difficult to understand. This problem is aggravated when combined with an improper inheritance hierarchy, because the class may override many methods due the fact that the parent class is not appropriate. So, the negative evaluation is consistent with the qualitative evaluation, as well as the positive evaluations were considered correct.

Lack of Cohesion in Methods (LCOM): the LCOM column of Table III shows that 6 of the 8 classes have low cohesion. Classes of *service* type do not have cohesion between their methods since they have no fields. Therefore, the inspected *services* classes have a consistent poorly evaluation by the LCOM threshold. As the *managed beans* are poorly componentized, they have low cohesion. If they were better componentized, cohesion would increase naturally, after all, the methods would have higher similarity. On the other hand, DAO objects have cohesion by the fact that their methods use fields inherited from the parent classes related to the *framework*, performing correlated operations in the database. Therefore, the positive evaluations were correct.

Number of Methods (NOM): the NOM column shows that all classes were poorly evaluated by NOM threshold, which is in accordance with the assessments made by WMC threshold. According to Lehman [16], the functionalities offered by a system must be continuously incremented in order to maintain user satisfaction. If this growth is done in an uncontrolled way, classes will grow more and more by adding methods and fields that meet the growing expectations of the user. Failure to do this growing in a designed way will deteriorate the software quality, since the classes become large and complex, as in the case of the classes analysed.

Specialization Index (SIX): the SIX column of Table III shows that 7 classes were well evaluated and the remaining one was poorly evaluated by the SIX threshold. This metric aims to assess how much a particular class overrides the behavior of its superclasses. As expected, due the relation of SIX with NORM and DIT, NB was poorly evaluated by SIX, as was poorly evaluated by NORM and not well evaluated by DIT. The other classes do not exceed normal levels of specialization index suggested by SIX threshold and, in fact, they do no overwrite the behavior of their superclasses in an excessive way.

1) Conclusion: For this part of the study case, it was established with the programmers of XYZ a set of 8 poor-quality classes, with the aim of study the evaluations obtained by applying the proposed thresholds. All classes had at least 3 classifications out of range *Good/Common*. This suggests that if our catalogue of thresholds were applied in the management of internal quality of software systems, all the classes of the sample would be defined as objects of inspection in a measurement process, for presenting unusually high values. However, we also concluded that a single metric of our catalogue should not be used to define the quality of a class. For example, if only NOF was used to evaluate the 8 classes, only two would be considered as poor-quality. This conclusion is consistent with the work of Rosenberg et al. [4], which suggests that a single metric should not be used to evaluate a class. So, the results suggest that our catalogue is an efficient way to evaluate the classes effectively.

C. Part 3 - Bad Smells Correlation

We applied JDeodorant [19], a plugin for Eclipse that identifies bad smells, in order to identify *long methods* and *god classes* in XYZ. After that, we evaluated all the methods and classes of this system with some metrics of our catalogue related to these bad smells concepts. With these data available, we crossed the information of two qualitative variables: the presence or absence of the bad smell and the classification or no classification as *Good/Common* by the threshold. So, we obtained the number of methods or classes that have or not the bad smell against the amount of classified and unclassified methods or classes as *Good/Common* by the thresholds. This type of information is called contingency table of two qualitative variables. Furthermore, we raised the following null hypothesis about the bad smell occurrence and the evaluation obtained by the method or class with m threshold, H_m^{null}: the bad smell occurrence is independent of the method or class not be classified by the metric m in *Good/Common* range.

To evaluate this hypothesis, we used a statistical test that evaluates the dependency between the qualitative variables, called Chi-Square Test for Independence, which determines whether there is a significant association between the two established variables. The input of this test is the 2×2 contingency table, and the output is the *p*-value, which is the probability of obtaining an statistic test equal or more extreme than the one observed in the sample about. If the *p*-value is greater than 5%, we do not reject the null hypothesis. Otherwise, we can reject it, which would suggests that there is a relationship between the presence of the bad smells and the method or class not be classified as *Good/Common* by the threshold. The test was applied using the R tool [20] for the WMC, NOM, NOF and LCOM thresholds with the identified *god classes* and the evaluation obtained by the MLOC, NBD and VG with the identified *long methods*. For

all of these tests, we obtained p-values less than 1%, rejecting the null hypothesis at 99% of confidence level. Therefore, it was possible to check that there is a statistical dependency between: (1) the class be evaluated as *Good/Common* by the thresholds of WMC, NOM, LCOM and NOF and do not have the bad smell *god class* and (2) the method be evaluated as *Good/Common* by the thresholds of MLOC, NBD and VG and do not have the bad smell *long method*. These situations testify in favour of the correctness of the evaluations performed by our thresholds.

D. Other Metrics

Number of Classes (NOC): a package is a collection of related types providing access protection and name space management [21]. As much as classes are added to a given package, group of classes tend to be less interrelated, suggesting a possible re-division into smaller packages to create new groups which reflect a better defined domain. The NOC threshold may be used to identify packages with a large number of classes, in order to evaluate possible adjustments.

Afferent Coupling (CA): CA measures the number of external classes to a specific package that depend on their inner classes [22]. The higher the CA value, the greater the responsibility of that package and the higher its relevance within the software. A package with many external dependencies becomes a risk artifact, knowing that a change on it may impact directly and indirectly in many classes. In this sense, the identified thresholds are useful in order to say what is a high CA value, based on the standards of software quality that has been developed. Therefore, knowing what is a high value of CA may aid to identify the need for re-distribution of responsibilities of a too influential package in a set of packages more cohesive.

Efferent Coupling (CE): CE is the number of internal classes in a package that depend on external classes of this package [22]. A high value of CE means that the package strongly depends on other classes of other packages, making it a more unstable artifact, given the high degree of dependent classes. Keeping a low degree of CE means getting a package with greater independence. The identified thresholds show that most packages in OO software have been developed with up to 6 dependent classes, occasionally they have 7 to 16 dependent classes and rarely more than 16.

Number of Static Methods (NSM): a static method belongs to the class, rather than to an instance of the class [21]. Despite this mechanism breaks the object-oriented programming concept, it has practical utility in the context of object-oriented software development in the Java Platform [23]. However, they make the software less flexible because they cannot be overridden. Therefore, the NSM threshold allows to identify high values of static methods.

Number of Static Fields (NSF): a static field creates an attribute that belongs to the class rather than being associated with an instance of that class [21]. All class instances share the static field, which is in a fixed location in memory. Any change in an static field value will reflect in all instances of the class. Static fields are extremely useful in the object-oriented software developed in Java platform [23]. An example is the implementation of the design pattern *Singleton*, which guarantees the existence of only one instance of a particular class, providing global access to that object. However, classes that excessively use this feature and also static methods have acquired a bad reputation because it prevents developers to think in terms of objects [23]. The identified thresholds indicate what is a high value for NSF, allowing the system developer to identify classes in the design that are using this feature excessively.

VII. Related Work

In this section, we discuss related work that has been done in order to identify thresholds for object-oriented software related metrics. Rosenberg et al. [4] identified thresholds of metrics with the goal of applying these metrics to assess the reliability of software systems at NASA. The research was conducted in more than 20,000 classes distributed over 15 projects. The goal was to identify thresholds capable of discriminating weak code from solid code by statistical studies. This study presents thresholds for six software metrics at class level. As the analysis was done in the context of software development at NASA, the results not necessarily can be applied to other application domains. Moreover, as the dataset is not open and the method is not described in details, the results of the research are not reproducible.

Shatnawi et al. [24] presented a study of the relationship between object-oriented metrics and error-severity categories, identifying thresholds values that separate no-error classes from classes that had high-impact errors. This study presents thresholds for five metrics at class level. Moreover, the method was applied in a limited size and domain sample, being a threat to use these thresholds for software in general.

Alves et al. [5] designed a method that determines metric thresholds empirically from a statistical analysis of a benchmark of software systems, which are derived by choosing the 70%, 80% and 90% percentiles from these data. The authors focus on the method description, presenting thresholds for 3 metrics at method level and 2 metrics at class level. Besides that, fixed percentiles not necessarily work for all metrics. Due to this limitation, in the present work we carried out the data analysis by understanding the distribution curve of the values to establish these percentiles.

Oliveira et al. [6] proposed the concept of relative thresholds for evaluating metrics data that follow a heavy-tailed distribution. The thresholds are called relative because they assume that metric thresholds should be followed by most sources code entities, but that is also natural to have a number of entities in the "long-tail" that do not follow the defined limits. So, absolute thresholds should be complemented by the percentage of entities that the upper limit should be applied to. This work has focused on the method description, deriving thresholds for seven metrics at class level.

Our work presents 17 object-oriented software metrics derived by the same method, that is a large amount of thresholds compared with previous studies. Besides that, our catalogue does not cover only metrics at class level, but also metrics at method and package level. The used approach is easily reproducible and does not bring much statistical complexity. Moreover, the proposed thresholds are based on analysis of a large amount of software, of various sizes and domains, making the results more reliable in terms of representativeness of software generally.

VIII. Threats to Validity

Software metric tools can measure different values for the same metric. The identified thresholds may classify artifacts as worse than they in fact are. So, like Qualitas.*class* Corpus [7] relied on Metrics Plugin for Eclipse to collect the measures, we cannot assure that the identified thresholds will be applicable when using tools that collect metrics with an implementation different from Metrics. The sample of applications used in this research may also be a threat to the validity of this study, because of its size and representativeness for software in general. However, Qualitas.*class* Corpus has at least equal software samples than other analysed studies and is composed of various types and domains of software systems. Another threat to validity is that this approach assumes that the metrics are unidirectional in the sense of having a clear good and bad orientation. So, if all classes show, for example, $DIT = 0$, they would be classified as *Good/Common* by the suggested thresholds, although the design is suboptimal, it does not use inheritance. However, none of our suggested thresholds contemplate the evaluation of the global quality of the software systems and $DIT = 0$ is the most frequent value found in practice. This triggers an alarm about the misuse of our catalogue and misinterpretation of its results, being fundamental to evaluate the scenario in which it will be applied.

IX. Conclusion and Future Work

The knowledge of the thresholds is of fundamental importance in the promotion of the effective use of software metrics regarding the management of internal quality of software systems. In this research, we employed the method proposed by Ferreira et al. [3] to propose a catalogue of 17 thresholds for object-oriented software metrics, covering the quantitative evaluation of methods, classes and packages. The method is based on the analysis of the statistical distributions of measures found in practice. It was observed that the metrics fit a heavy-tailed or a skewed-right distribution. So, three ranges of values were taken as the metric thresholds. The range names were modified to *Good/Common*, *Regular/Casual* and *Bad/Uncommon*, which express better the importance of frequency concept in the thresholds. Although they do not necessarily express the best design principles established for Software Engineering, they reflect a quality standard followed by most of the software evaluated.

The evaluation of the proposed thresholds in a proprietary software showed the effectiveness of our catalogue of the thresholds in indicating the real panorama of the internal quality of software systems, that is, the evaluation do not show quality where there is not. For metrics that do not participate in the evaluation conducted in a proprietary software, we presented a qualitative analysis which describes their appliance in the identification of possible problems in object-oriented software systems. So, we presented at least one analysis for each one of our suggested thresholds. With our catalogue of thresholds, we presented a contribution in the promotion of software metrics as an effective instrument to manage the internal quality of software systems.

As future work, we intend to continue evaluating the identified thresholds through more case studies, aiming to continue the investigation if the range in which the metric falls reflects the real situation of the assessed artifact. Furthermore, we intend to conduct the development of a tool that performs a strategy composition of metrics to identify software entities to be refactored, by applying metric thresholds.

References

[1] I. Sommerville, Software Engineering, 9th ed. Harlow, England: Addison-Wesley, 2010.

[2] M. Riaz, E. Mendes, and E. D. Tempero, "A systematic review of software maintainability prediction and metrics," in ESEM, 2009, pp. 367–377.

[3] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," Journal of Systems and Software, vol. 85, no. 2, Feb. 2012, pp. 244–257.

[4] L. Rosenberg, S. Ruth, and A. Gallo, "Risk-based Object Oriented Testing," in Proceedings of the 24 th annual S.E. Workshop, NASA, S.E.Lab, 1999, pp. 1–6.

[5] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data." in ICSM. IEEE Computer Society, 2010, pp. 1–10.

[6] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in CSMR-WCRE, Software Evolution Week - IEEE Conference on, Feb 2014, pp. 254–263.

[7] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas.class Corpus: A compiled version of the Qualitas Corpus," Sof. Eng. Notes, vol. 38, no. 5, 2013, pp. 1–4.

[8] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in APSEC2010, Dec. 2010, pp. 336–345.

[9] "Eclipse metrics plugin 1.3.8," 2014, URL: http://metrics2.sourceforge.net [accessed: 2014-12-30].

[10] R, "R project for statistical computing," 2014, URL: http://www.r-project.org/ [accessed: 2014-12-30].

[11] T. G. Filó, M. A. Bigonha, and K. A. Ferreira, "Statistical dataset on software metrics in object- oriented systems," Sof. Eng. Notes, vol. 39, no. 5, 2014, pp. 1–6.

[12] "Easyfit," 2014, URL: http://www.mathwave.com/products/easyfit.html [accessed: 2014-12-30].

[13] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the shape of java software," in OOPSLA, New York, NY, USA, 2006, pp. 397–412.

[14] L. Doane, David & Seward, "Measuring Skewness: A Forgotten Statistic?" J. of Statistics Education, 2011, pp. 1–18.

[15] M. Lanza, S. Ducasse, and R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, 2007.

[16] M. M. Lehman, "Programs, cities, students, limits to growth?" Programming Methodology, 1978, pp. 42–62, inaugural Lecture.

[17] M. Fowler, "Anemic domain model," 2014, URL: http://www.martinfowler.com/bliki/AnemicDomainModel.html [accessed: 2014-12-30].

[18] Oracle, "The java ee 6 tutorial," http://docs.oracle.com/javaee/6/tutorial/doc/bnaqm.html, 2014.

[19] "Jdeodorant," 2014, URL: http://www.jdeodorant.com/ [accessed: 2014-12-30].

[20] "Chi-squared Test of Independence," 2014, URL: http://www.r-tutor.com/elementary-statistics/goodness-fit/chi-squared-test-independence [accessed: 2014-12-30].

[21] Oracle, "Java se technical documentation," 2014, URL: http://docs.oracle.com/javase/ [accessed: 2014-12-30].

[22] R. Martin, "OO design quality metrics - an analysis of dependencies," in Workshop Pragmatic and Theoretical Directions in O.O. Software Metrics. OOPSLA, 1994, pp. 1–6.

[23] J. Bloch, Effective Java, 2nd Edition, The Java Series. NJ, USA: Prentice Hall PTR, 2008.

[24] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding software metrics threshold values using ROC curves," Journal of Software Maintenance and Evolution: Research and Practice, vol. 22, no. 1, 2010, pp. 1–16.