Automatic metric thresholds derivation for code smell detection

Francesca Arcelli Fontana*, Vincenzo Ferme†, Marco Zanoni*, Aiko Yamashita‡
*Department of Informatics, Systems and Communication
University of Milano-Bicocca, Milano, Italy
Email: arcelli@disco.unimib.it, marco.zanoni@disco.unimib.it
†Faculty of Informatics
University of Lugano (USI), Switzerland
Email: vincenzo.ferme@usi.ch
‡Mesan AS
Henrik Ibsens gate 20, Oslo, Norway
Email: aikoy@mesan.no

***Abstract*—Code smells are archetypes of design shortcomings in the code that can potentially cause problems during maintenance. One known approach for detecting code smells is via *detection rules*: a combination of different object-oriented metrics with pre-defined threshold values. The usage of inadequate thresholds when using this approach could lead to either having too few observations (too many false negatives) or too many observations (too many false positives). Furthermore, without a clear methodology for deriving thresholds, one is left with those suggested in literature (or by the tool vendors), which may not necessarily be suitable to the context of analysis. In this paper, we propose a data-driven (i.e., benchmark-based) method to derive threshold values for code metrics, which can be used for implementing *detection rules* for code smells. Our method is transparent, repeatable and enables the extraction of thresholds that respect the statistical properties of the metric in question (such as scale and distribution). Thus, our approach enables the calibration of code smell detection rules by selecting relevant systems as benchmark data. To illustrate our approach, we generated a benchmark dataset based on 74 systems of the *Qualitas Corpus*, and extracted the thresholds for five smell detection rules.**

I. Introduction

Code smells [1] are symptoms of design shortcomings in the code that can potentially cause problems during maintenance and evolution. Many of the available smell detection tools implement a metrics-based approach, with 'fixed' or configurable threshold values. In some cases, tool providers using 'fixed' threshold values do not provide a clear rationale on how these thresholds have been devised. In other cases such as in [2], the authors provide a rationale on how the thresholds values for some metrics have been identified (i.e., heuristics derived from the interpretation of object-oriented design principles, together with statistical analysis).

The usage of thresholds in code smell detection entails several challenges. First, one can argue the adequacy of thresholds provided by tool vendors, for which the derivation rationale is often undisclosed. This leads to unclear assumptions on the validity and representativeness of these thresholds. Even if the rationale is given for the thresholds, if the process followed to arrive to them is not repeatable, it is not possible in practice to compare them with other values or techniques. Furthermore, the usage of inadequate thresholds could lead to either having too many false negatives or too many false positives. Developers more than often end up with long lists of 'candidates' for examination, and may have hard time prioritizing the most critical instances.

In sum, without a clear methodology for deriving thresholds, one is left with those suggested in the literature (or by tool vendors), which may not necessarily be suitable to the context of analysis. A transparent, repeatable mechanism is needed in order to derive the thresholds most suitable to the context. In that way, we can move towards more useful and adequate code smell-based evaluations of source code quality. In this paper, we propose a data-driven method (i.e., using a benchmark dataset of software systems) to extract metric thresholds for defining code smell detection rules. We call the extracted thresholds as Benchmark-Based Thresholds (BBT).

Our method is characterized by three features: *1)* it relies on the statistical distribution of metrics for selecting thresholds; *2)* it applies a non-parametric process to discard the part of the distribution that is not useful for deriving thresholds; *3)* it customizes the definition of the thresholds for each code smell, by filtering the dataset according to the code smell.

To illustrate our method, we extracted 40 different metrics from 74 systems of the *Qualitas Corpus* (v. 20120401r) curated by Tempero et al. [3] to build a benchmark dataset, and we illustrate how the extracted threshold values can be used in five code smells detection rules defined by Lanza and Marinescu [2].

The paper is organized as follows. In Section II, we describe our approach inspired by a previously suggested data-driven technique. In Section III, we illustrate how some of the thresholds derived through our method can be used in an existing code smell detection approach. In Section IV, we discuss the current limitations of the work. In Section V, we provide related work on metric thresholds derivation. Finally,

2327-0969/15 $31.00 © 2015 IEEE
DOI 10.1109/WETSoM.2015.14

Section VI provides concluding remarks and discusses avenues for future research.

II. Thresholds Derivation Process

Our data-driven approach is inspired by the thresholds derivation work reported by Alves et al. [4]. The primary goal of the approach by Alves et al. is to use the extracted thresholds for building maintainability assessment models [5]. Conversely, the main goal of our approach is to derive thresholds for code smell detection rules, which can furthermore be used for software quality and maintainability assessments.

Our method is meant to be applied on metrics that do not represent ratio values of other metrics, which are typically ranging in the interval [0,1]. An example of a ratio value metric is the *Tight Class Cohesion* (TCC), and such metrics are not covered in our approach. We will hereon refer to the thresholds computed with our method as BBT.

In the remainder of this section, we will explain the process for extracting thresholds, by first describing the dataset from which the metrics were extracted. Then, we explain how we used the distributions of the metrics for mapping symbolic threshold values to actual values.

We focus on the metrics used in the book by Lanza and Marinescu [2] for the definition and detection of the following code smells: God Class, Data Class, Brain Method, Dispersed Coupling, and Shotgun Surgery. The metrics used for the code smell detection rules are reported in Table I.

A. Metric Thresholds Definition

Detection rules for code smells are often defined in the terms of metric groups or classifications. An example can be: "identify classes that have LOW cohesion" or "identify methods that have a HIGH complexity". We want to derive thresholds in a way that they can be semantically mapped to these informal requirements, to find out what LOW cohesion or HIGH complexity means in terms of the metrics we use to measure the cohesion and complexity of the software.

With this need in mind we compute, for each metric, a set of three thresholds that identifies three points on the metrics distribution. We define a threshold using a name and a corresponding percentile. We use the name to refer to a threshold with a semantic meaning. The names we use for the three thresholds are: LOW, MEDIUM, and HIGH.

The percentile corresponding to a threshold name is the percentile on the metrics distribution where to find the value for the given threshold. In other words, every name is an alias to a percentile, which is a pointer to the place on the metric distribution where to find the value of the metric to use as the actual threshold.

B. Benchmark-Based Thresholds

The metrics we consider in our method have values belonging to the extended real line $\mathbb{R}$ or to the natural number line $\mathbb{N}$, mostly starting from zero.

The starting point of our method is the one proposed by Alves et al. [4], which follows three core principles:

1) *"The method should not be driven by expert opinion but by measurement data from representative set of systems (data-driven);"*
2) *"The method should respect the statistical properties of the metric, such as metric scale and distribution and should be resilient against outliers in metric values and system size (robust);"*
3) *"The method should be repeatable, transparent, and straightforward to carry out (pragmatic)."*

Their method is aimed at finding thresholds for system-level metrics, to assess the risk levels of a given system. Our method makes similar assumptions on the nature of the raw data, but is aimed at finding metric thresholds applicable to single elements in the code (e.g., a class, a method), to determine if they belong to a certain category or group.

The proposed method allows to compute in an automated way, thresholds that are needed to detect each code smell. The method displays a high level of automation, allowing the extension of the benchmark data with additional systems over time, to support a meaningful and up-to-date set of thresholds. The main steps of the method are:

1) Metrics computation: metric values of all classes and methods of the analyzed systems are computed;
2) Aggregated metrics distribution analysis: the quantile function of each metric distribution is computed;
3) Metrics thresholds derivation: the point on the quantile function to assign to the labels (LOW, MEDIUM, HIGH) is selected.

1) Metrics computation: The first step consists of identifying a set of systems to be used as a benchmark, and computing the metrics values. Table I presents the metrics we use to illustrate our work, and their definitions by Lanza and Marinescu [2]. In addition, we report in Table VII all the other metrics we analyzed with our method; metrics defined in the literature have a reference, and the ones we defined are in bold font.

The systems we selected as a benchmark are the ones from the *Qualitas Corpus* (v. 20120401r), a curated collection of 111 open source systems created by Tempero et al. [3]. We selected 74 of the available systems for our benchmark. The size of the selected systems is reported in Table II. We selected the systems that were possible to compile, and added missing libraries when required. Compilation is necessary for the computation of dependency metrics in our environment (we exploit the Eclipse JDT library), for the resolution of dependent classes also in external libraries. We use the metadata from the *Qualitas Corpus* to consider only the "core" classes for our analysis; we exclude, i.e., tests and demos from the dataset. During the manual inspection of the systems (e.g., to solve compilation problems), we also removed all the test packages we were able to identify.

For each code smell, we compute a benchmark database that contains *only* the entities that can be affected by that specific smell (this is done by using a *selection criterion* for each code smell). The set of values for all the metrics that are part

Table I
METRICS DEFINITIONS

Metric Label	Metric Name	Definition
ATFD	Access To Foreign Data	The number of attributes from unrelated classes belonging to the system, accessed directly or by invoking accessor methods.
WMC	Weighted Methods Count	The sum of complexity of the methods that are defined in the class. We compute the complexity with CYCLO.
NOAP	Number Of Public Attributes	The number of public attributes of a class
NOAM	Number Of Accessor Methods	The number of accessor (getter and setter) methods of a class.
LOC	Lines of Code	The number of lines of code of an operation or of a class, including blank lines and comments.
CYCLO	Cyclomatic Complexity	The maximum number of linearly independent paths in a method. A path is linear if there is no branch in the execution flow of the corresponding code.
MAXNESTING	Maximum Nesting Level	The maximum nesting level of control structures within an operation.
NOAV	Number Of Accessed Variables	The total number of variables accessed directly or through accessor methods from the measured operation. Variables include parameters, local variables, but also instance variables and global variables declared in classes belonging to the system.
CINT	Coupling Intensity	The number of distinct operations called by the measured operation.
CM	Changing Methods	The number of distinct methods calling the measured method.
CC	Changing Classes	The number of classes in which the methods that call the measured method are defined in.

of a given code smell detection rule is hereon called *code smell metrics set*. The *selection criteria* for the code smells considered in this paper are:

- God Class: any class, including anonymous classes; interfaces and enums are discarded;
- Data Class: any concrete (non-abstract) class, and anonymous classes; interfaces and enums are discarded;
- Brain Method and Dispersed Coupling: non-abstract methods contained in a class, anonymous class or enum (no interface); constructors are included, but not default constructors;
- Shotgun Surgery: non-abstract methods contained in a class or enum (no anonymous classes or interfaces); constructors are excluded.

2) Metrics Aggregated Distribution Extraction: At the end of step 1, we obtain for each *code smell metrics set*, an aggregated set of values from entities belonging to different systems that are part of the benchmark. For each metric, we aggregate all the values obtaining an *aggregated distribution*, and we plot the Inverse Cumulative Density Function i.e., the Quantile Function (QF) for the aggregated metric distribution of each system.

There are many methods to estimate the quantiles of a population and define the QF. We refer to the *type 1* method defined by Hyndman and Fan [6]. This kind of analysis is not influenced by outliers, as it does not rely on the actual values of the metrics.

Figure 1 and Figure 2 show the QF plots regarding metrics measured on types (classes and interfaces) and methods,

Table II
STATISTICS ABOUT THE ANALYZED SYSTEMS

Number of systems	74
Lines of Code	6,785,568
Number of Packages	3,420
Number of Classes	51,826
Number of Methods	404,316

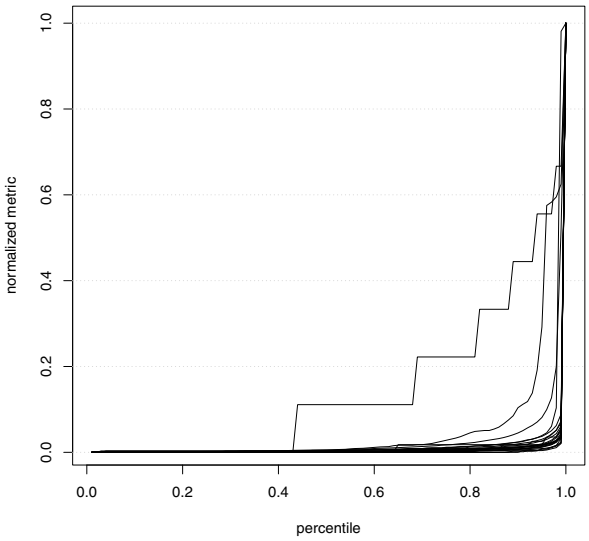

Figure 1. Type metrics QF overview

respectively. To represent the different curves together in the plots, the values of metrics have been normalized in the range [0,1]. The figures show that most metrics in the benchmarks have a common heavily skewed distribution: most values are concentrated in a small portion of the possible values, the one closest to the lowest quantiles.

The QF makes this fact clear because it grows very slowly for most of the quantiles, and after a given point, it grows very fast. As an example, we plot in Figure 3 the QF for metric LOC on types, cropped at 90% (only the last 10% of the distribution).

3) Metrics Cropped Distribution Extraction: If we use the distribution *as-is* to derive thresholds by applying common statistical approaches (e.g., the boxplot approach by Fenton et al. [7]), we run the risk of obtaining values that do not characterize problematic entities. For example, the first quartile

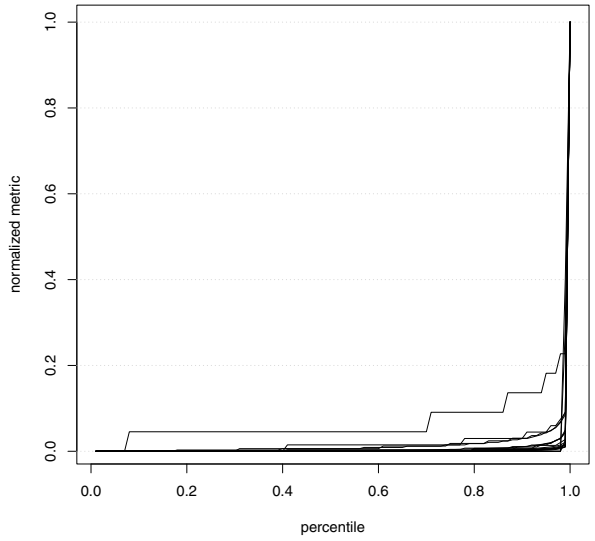

Figure 2. Method metrics QF overview

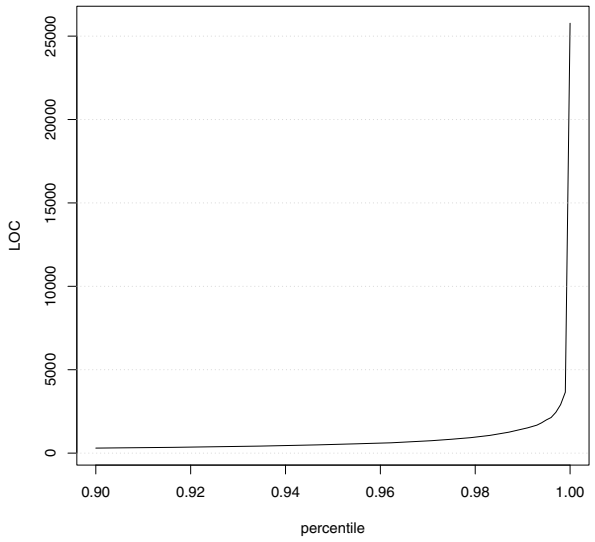

Figure 3. Aggregated LOC QF Cropped at 90%

is often equal to the lowest value the metric can have (e.g., zero for LOC); even the median can be one of the very few first values in the distribution. For this reason, we define an automatic method to select the point of the metric distribution where the variability among values starts to be higher than the average, and we use it as the starting point for deriving thresholds.

The goal of the procedure is to find a *cut-point* on the x-axis of the QF that splits the QF in two parts: the left part contains the lower and most repetitive values, the right part contains the higher and more variable values, which we will use for the threshold computation; we call the right part the *cropped distribution*. We start from the verified assumption that the distribution is highly right-skewed.

The procedure samples 100 equally distributed values of the QF (i.e., percentiles), in the range (0.01, 1). The outcome is an array a of 100 elements; each element is the value of the QF in the respective position.

Given a function f, representing the frequency of a value in a, a f_{mid} value is computed as the median of the frequency of values in a. With f_{mid}, we can compute v_{mid} as $min\{v \in a | f(v) \leq f_{mid} \wedge (\forall w \in a | w \geq v \rightarrow f(w) \leq f_{mid})\}$. v_{mid} represents the minimum value of the metric we will consider in the output distribution. Therefore, to select the quantile where to crop the distribution, we look for the maximum quantile having a value strictly less than v_{mid}.

As an example, Table III shows the different values of NOM (Number Of Methods) in descending order, associated to their frequency in the distribution. Figure 4 shows an histogram representing the same data. The median of the frequencies is 1. Therefore, we select in Table III the rightmost (smaller) value having all frequencies on his left less than or equal to 1. The resulting value is 13 (marked in bold in Table III). Value 13 is then searched over the QF for metric NOM; the first element having value 13 is at the 87th percentile. As a consequence, the last 14% of the distribution is used for computing the threshold values, while the first 86% is discarded.

Alves et al. [4] too, as part of their method, select a part of the distribution where to pick dangerous reference values for a metric. They do this work by hand, and highlight the problem of identifying those points according to the metric distribution. They use 70th, 80th, and 90th percentile as common choice and then they adapt the selection based on the metric distribution. Our method executes this task in an automated way. The cut points we found for the considered metrics are reported in Table IV.

4) Thresholds Derivation: After cropping the distribution in the previous step, we compute the new QF and then we derive the three thresholds. Here are the labels associated to the thresholds, and the respective percentile on the new QF:

- LOW: 25th percentile;
- MEDIUM: 50th percentile;
- HIGH: 75th percentile.

In Table V we show all the final extracted threshold values of the metrics used for the detection of the code smells we have considered in this paper.

It can happen that the values of two or more derived thresholds for a metric are the same, depending on the values distribution. For example, in Table V we can see that MAXNESTING has the same value (4) for both the LOW and MEDIUM thresholds. This happens because the values of the MAXNESTING metric are distributed in a little range. In this situation, associating different labels to the same actual value can be confusing, especially when the values are exploited for code smell detection: different rules could represent the same query. It is possible to recalibrate thresholds in this case. A simple recalibration procedure is be to keep the MEDIUM threshold fixed, and move the other two thresholds

Table III
NOM CROP DATA

NOM	674	60	42	34	29	25	23	21	19	17	16	15	14	**13**	12	11	10	9	8	7	6	5	4	3	2	1	0
Freq	1	1	1	1	1	1	1	1	1	1	1	1	1	**1**	2	2	2	3	3	4	4	6	8	10	14	26	2

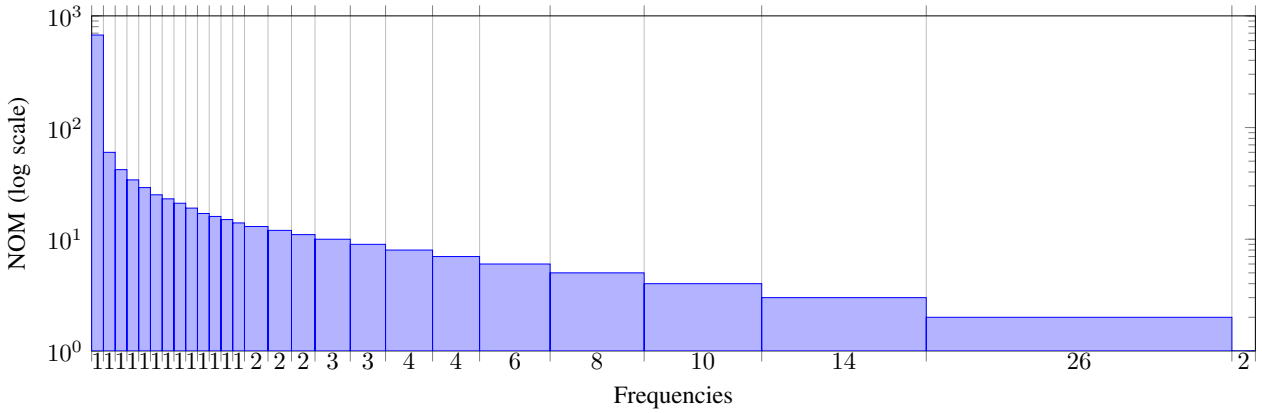

Figure 4. NOM crop graph

Table IV
CUT POINTS FOR DISTRIBUTION CROPPING

Code Smell	Metric	Value	Percentile
God Class	ATFD	8	91
	WMC	19	79
Data Class	NOAP + NOAM	6	92
	WMC	16	76
Brain Method	LOC	16	82
	CYCLO	7	94
	MAXNESTING	4	95
	NOAV	10	90
Dispersed Coupling	MAXNESTING	4	95
	CINT	6	94
Shotgun Surgery	CM	4	94
	CC	4	96

Table V
FINAL EXTRACTED THRESHOLDS

Code Smell	Metric	LOW	MEDIUM	HIGH
God Class	ATFD	10	13	22
	WMC	25	36	64
Data Class	NOAP + NOAM	7	10	17
	WMC	21	32	57
Brain Method	LOC	20	28	45
	CYCLO	8	10	15
	MAXNESTING	4	4	5
	NOAV	11	14	20
Dispersed Coupling	MAXNESTING	4	4	5
	CINT	7	9	12
Shotgun Surgery	CM	5	7	12
	CC	4	6	10

on the distribution in opposite directions, until the values are disambiguated. Following this procedure, we could simply change the LOW threshold to 3. However, this could lead to a misalignment of information between threshold values and percentiles. The end user inspecting the results would at least need to be notified that one of the applied thresholds is not tied to its default percentile. For this reason, we choose to avoid applying recalibration. We can think of recalibration as an optional procedure, to be explicitly activated by the user.

III. EXTRACTING THRESHOLDS FOR EXISTING RULES: AN EXAMPLE

We analyse the extracted thresholds under the light of the work by Lanza and Marinescu [2], which reports a set of code smell detection rules. The thresholds reported in the book were initially derived from the statistical analysis of 45 Java software systems, and then manually adapted or combined for each particular code smell, as needed.

We focus on the rules regarding the detection of five code smells: God Class, Data Class, Brain Method, Dispersed Coupling, Shotgun Surgery. By using our method, we compute the thresholds of all metrics involved with these rules and compare them with the ones proposed in the book.

Table VI displays the detection rules for the smells considered and their respective metrics (definitions are available in Table I). The metrics are reported alongside their comparison with a symbolic threshold, as defined by each detection rule. Detection rules in [2] are expressed as predicates in logical composition, where metric values are compared with symbolic thresholds. Figure 5 provides as an example, the detection rule for the Brain Method code smell.

Table VI
CODE SMELL AND METRICS WITH THRESHOLDS

Code Smell	Metric	Comparator	Threshold	Threshold OO	Threshold A	Threshold B	Threshold C
God Class	ATFD	$>$	FEW	2-5	0	10	10
	WMC	$\geq$	VERYHIGH	47	15	59	64
Data Class	NOAP	$+$	NOAM $>$ FEW	2-5	0	8	7
	NOAP	$+$	NOAM $>$ MANY	5	1	20	17
	WMC	$<$	HIGH	31	15	59	57
	WMC	$<$	VERYHIGH	47	15	59	57
Brain Method	LOC	$>$	HIGH (Class) / 2	65	61	94	91
	CYCLO	$\geq$	HIGH	3	2	19	15
	MAXNESTING	$\geq$	SEVERAL	2-5	2	5	5
	NOAV	$>$	MANY	5	5	17	20
Dispersed Coupling	MAXNESTING	$>$	SHALLOW	1	1	4	4
	CINT	$>$	Short Memory Cap	7-8	0	9	9
Shotgun Surgery	CM	$>$	Short Memory Cap	7-8	0	8	7
	CC	$>$	MANY	5	1	10	10

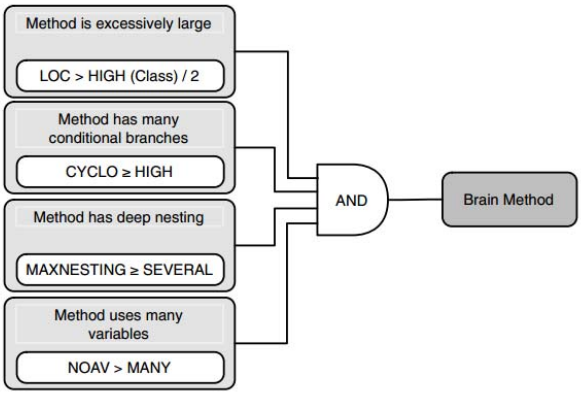

Figure 5. Brain Method Detection Rule

We derived from [2] the values assigned to the symbolic thresholds (for Java), and we report them in the "Threshold OO" column in Table VI. As the book does not explicitly report the exact value for all the thresholds, we did our best to estimate those not available. The last three columns of Table VI are the thresholds we computed considering:

- the *aggregated distribution* on the whole dataset, without filtering the dataset according to the code smell ("Threshold A");
- only the *cropped distribution* on the whole dataset, without filtering the dataset according to the code smell ("Threshold B");
- the *cropped distribution* on the code smell metrics set, i.e., the dataset filtered according to the code smell ("Threshold C").

The symbolic names used in Lanza and Marinescu's book for thresholds are different than our threshold names. To allow the comparison of the extracted values, we associated every symbolic name used in the book to one of the three names defined in our method:

- SHALLOW/FEW $\rightarrow$ LOW
- Short Memory Cap $\rightarrow$ MEDIUM
- MANY/HIGH/SEVERAL/VERYHIGH $\rightarrow$ HIGH

In Table VI, one of the predicates of the Brain Method detection rule compares the LOC metric to the $HIGH(class)/2$ threshold. This means that the LOC of the evaluated method are compared to half of the LOC of the method's container class. The usage in the Brain Method rule of a metric regarding the class that contains the evaluated method does not fit our procedure. Our method filters the dataset, keeping only the elements (methods in this case) that can be affected by the particular smell, and computes thresholds on the metric of only those elements. To allow the comparison of the thresholds we extract using our method with the ones used by Lanza and Marinescu [2], in Table VI we report in as Threshold A the value of $HIGH(class)/2$ as used by the reference, and in Thresholds B and C the value of our threshold $HIGH/2$, but computed considering the dataset we use for God Class, not the one used for Brain Method. This exception makes the comparison of our thresholds and the reference one more consistent. Following our approach, instead, we would prefer to use in a detection rule the regular HIGH threshold computed on the Brain Method dataset, achieving different thresholds, i.e., 11, 43, 45 respectively for Thresholds A, B, C.

From Table VI we can see that the thresholds produced automatically by our approach tend to be higher than the ones from Lanza and Marinescu. Threshold C values are from 2 to 4 times Threshold OO, with the exception of the VERYHIGH threshold of WMC, where the difference is less strong (47 vs 57–64). Another exception is CM and CINT metrics w.r.t. the Short Memory Cap threshold, which was mapped to MEDIUM in our approach. In this case, values differ of one unit at most.

We can attribute the reason of the large difference in values between the thresholds from [2] and the thresholds from our approach to the fact that we are selecting very narrow

portions of the whole distribution in our cropping procedure. This moves the focus on the highest values in the metric distribution.

It is also worth noting the difference among Thresholds A, B and C. The highest difference is obviously between A and B, since in B we are selecting a small part of the distribution for threshold calculation. The difference between B and C is less pronounced, yet it displays interesting properties. First of all, different selection criteria can change the sign of the difference between B and C. For example, WMC is increased in God Class and decreased in Data Class. Second, the values did not change for metrics with values in a small range, e.g., MAXNESTING. This is natural, since the chance of having a different value in the respective distribution is much smaller, also considering that we already cropped the datasets. Overall, in 12 cases out of 14, Threshold B $\geq$ Threshold C, meaning that the most evident effect of filtering the dataset is to lower thresholds.

IV. Challenges and Limitations

Despite the greater level of automation that our proposed method provides, we do not claim that there is no need for human interpretation or adjustment. For instance, our method cannot identify if an overall tendency on a given metric is symptomatic of a greater design issue. For instance, one could verify that DIT should not exceed a certain threshold, but if all DIT values are too low, then the system does not benefit from the power of object-orientation and is probably facing a larger design issue. Instead, we propose this approach as a complement to other analysis techniques, intended to simplify the task of metric/code smell interpretation and quality evaluation.

We are also aware that derivation of metrics thresholds from a benchmark dataset introduces dependency on the dataset and its representativeness. Also, threshold values may be dependent on the way metrics were computed. Further investigation needs to be done to determine the degree of sensitivity of this approach with respect to different benchmark datasets, precomputed metrics, and different metric extraction tools. However, our intended contribution is the method/strategy to extract threshold values, and not the threshold values themselves.[1]. The idea is that this approach can be used and repeated across different contexts so that code analysis (in particular code smell analysis) can be calibrated according to the context of analysis.

Furthermore, a full experiment evaluating the effect of the application of our thresholds for the detection of code smells has not been performed yet. The construction of such an experiment is also complicated, because we have no reference benchmark to evaluate our approach. From our preliminary results, we can say that the reported thresholds are more selective than the reference ones from [2], i.e., less false positives, whiles we cannot currently quantify the effect on false negatives. We are investigating the results of our approach from different perspectives, e.g., the effect of selecting thresholds using datasets from specific domains, using different criteria for selecting the QF cut point.

[1] These values are used primarily to illustrate our approach

V. Related Work

In the literature, besides the already cited work by Alves et al. [4], different approaches and techniques to derive metric thresholds can be found, e.g., via observation, correlation with defects, metrics analysis, statistical analysis, and machine learning.

In earlier work, Coleman, McCabe and Nejmeh, defined metric thresholds according to their experiences [8], [9], [10]. Other efforts for defining metric threshold values can be seen in the work by Henderson-Sellers [11], who suggested three ranges for different metrics to classify classes into three categories: *safe*, *flag*, and *alarm*. Rosenberg [12], [13] suggested a set of threshold values for the Chidamber-Kemerer (CK) metrics. These values can be used to select classes for inspection or redesign.

Shatnawi et al. [14] and Benlarbi et al. [15] attempt to identify thresholds by investigating the correlation of the metrics with the existence of bugs or failures. Shatnawi et al. derive thresholds to predict the existence of bugs using three releases of the Eclipse project, while Benlarbi et al. investigate the relation of metric thresholds and software failures for a subset of the CK metrics using linear regression. However, both approaches are valid only for a specific error prediction model and thus, lack of general applicability.

Other authors tackle the thresholds computation using metric analysis. An example is by Erni and Lewerentz [16], who proposed a threshold derivation approach based on the mean and the standard deviation of a given metric. Often these analyses assume that the metrics are normally distributed (such as the case of [16]). We postulate alongside the authors of [17], [18], [19], that this assumption does not hold in most cases, as often metrics follow a power-law distribution.

More recent work have also used benchmarks for threshold derivation. Oliveria et al. [20] propose the concept of *relative thresholds* for evaluating metrics data following heavy-tailed distributions. The proposed thresholds are relative because they assume that metric thresholds should be followed by most source code entities, but that it is also natural to have a number of entities in the "long-tail" that do not follow the defined limits. They introduced the notion of a *relative threshold*, i.e., a pair including an upper limit and a percentage of classes whose metric values should not exceed this limit and describe an empirical method for extracting relative thresholds from real systems. They argue that the proposed thresholds express a balance between real and idealized design practices. Oliveira et al. [21] extend their previous work and propose RTTooL, an open source tool for extracting relative thresholds from the measurement data of a benchmark of software systems.

Ferreira et al. [22] defined thresholds for six object-oriented metrics using a benchmark of 40 Java systems. They assert that the extracted metrics values, except for DIT follow heavy-tailed distribution. Based on their experience, they defined

three threshold categories, but without establishing the percentage of expected classes for each category.

Another approach, proposed by Foucault et al. [23] assumes that thresholds depend greatly on the context (as programming language or application domain), thus the threshold computation is done by taking into account the context, for making a trade-off between the representativeness of the threshold and the computational cost. Their approach is based on an unbiased selection of software entities and makes no assumptions on the statistical properties of the metrics.

In order to facilitate metrics interpretation, Vasa et al. [24] propose analyzing software metrics over a range of software projects by using the Gini coefficient. They observed that many metrics not only display remarkably high Gini values, but these values are remarkably consistent as a project evolves over time.

Serebrenik et al. [25] propose a new approach for aggregating software metrics at the micro-level (e.g., methods, classes and packages) to the macro-level of the entire software system. They used the Theil index, an econometric measure of inequality, to study the impact of different categorizations of the artifacts, e.g., based on the development technology or developers' teams, and on the inequality of the metrics measured. Theil index is not metrics-specific and can be used to aggregate values produced by a wide range of metrics satisfying certain conditions.

Mordal et al. [26], considering the growing need for quality assessment of software systems in the industry, proposed and analyzed the requirements for aggregating metrics for quality assessment purposes. They present a solution through the Squale model for metric aggregation and validate the adequacy of Squale through experiments on Eclipse. Moreover, they compare the Squale model to both traditional aggregation techniques (e.g., the arithmetic mean), and to econometric inequality indices (e.g., the Gini or the Theil indices).

Finally, Herbold et al. [27] used a machine learning algorithm for calculating thresholds, where classes and methods are classified as satisfying or not the computed thresholds. They used 4 metrics to analyze methods and functions and 7 metrics for the analysis of classes.

None of the above mentioned approaches has as central focus the derivation of thresholds for code smell detection. In the literature, we did not found any comparable approaches with our work, except for the approach of Lanza and Marinescu, which combines statistical analysis and expert judgment (as briefly described in Section III). In our approach, we provide a filter mechanism to include in the benchmark only metrics regarding particular methods or classes, according to the code smell to be detected; and we automatically select values within the metrics distribution, without manually adjusting the percentiles representing the threshold values.

Our approach, as other previous work *1)* fully relies on data extracted from software systems, *2)* takes into consideration the metric distribution properties, without making unproven assumptions about the nature of data, and *3)* is completely automatic and repeatable. Finally, we derive thresholds at a lower level of granularity (i.e., at method or class levels), in contraposition to other approaches such as Alves et al. [4], which use a system-level type of granularity.

VI. Conclusion and Future Work

In this work, we proposed a data-driven approach to derive metric thresholds. We claim our method is repeatable, transparent and enables contextual customization. We have illustrated our approach by deriving metric thresholds from 74 Open Source systems from the Qualitas Corpus.

Our method allows to quantify systematically the elements of an informal definition (e.g., a threshold for metrics that measure class complexity, to identify "high complexity" classes) by observing the statistical properties of the metrics of interest, and by taking into account the metrics distribution (via non-parametric statistical analysis techniques). The proposed method can be used for more general software quality assessment tasks, as it allows defining ranges for metric values, which in turn can be exploited for detecting different kinds of design violations.

We are currently comparing the thresholds derived by this approach, with those previously obtained through a machine learning approach [28]. From our preliminary results, we observe that thresholds compared with the $\geq$ operator have comparable values for God Class and Brain Method smells. For Data Class, the rules are different across approaches, and so are the threshold values. The comparison of code smell detection tools is a complex task [29] and we aim to work towards this direction by exploiting our BBT method.

Future work includes the comparison of the results of our BBT derivation approach with other thresholds available in the literature or extracted by other tools, e.g., the recent RTTool [21]. In addition, we aim to evaluate our approach by considering aspects related to the context [30] of the projects involved in the benchmark, such as the application domain, size, age of the systems, and number of changes.

Furthermore, we are interested in integrating historical data into our threshold derivation method. In particular, the evolution of the metric values in different versions of a system can be exploited for fine-tuning thresholds, or for predicting possible future threshold violations.

Additional future work includes exploring the usefulness of the thresholds by investigating if the derived code smell rules help us to distinguish better classes and methods more prone to defects, by analysing medium-large industrial Java systems.

As mentioned previously, the derivation of metrics thresholds from benchmark datasets introduces dependency on the benchmarks and its representativeness. In this work, we have considered the Qualitas Corpus, but we aim in the future to consider industrial projects as part of the benchmark, and other corpora or precomputed metrics datasets, e.g., COMETS[2] [31], Helix[3] and Promise[4], as well as consider other metrics within the datasets. More formal methods can also be applied to maximize the representativeness of the datasets [32].

[2] http://java.llp.dcc.ufmg.br/comets/index.html
[3] http://www.ict.swin.edu.au/research/projects/helix/
[4] https://code.google.com/p/promisedata

REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2005.

[3] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Proceedings of the 17th Asia Pacific Software Engineering Conference*. Sydney, NSW, Australia: IEEE Computer Society, December 2010, pp. 336–345.

[4] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proc. IEEE Int'l Conf. Softw. Maintenance (ICSM2010)*. Timisoara, Romania: IEEE, Sep. 2010, pp. 1–10.

[5] R. Baggen, J. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[6] R. J. Hyndman and Y. Fan, "Sample quantiles in statistical packages," *The American Statistician*, vol. 50, no. 4, pp. pp. 361–365, Nov. 1996. [Online]. Available: http://www.jstor.org/stable/2684934

[7] N. E. Fenton and S. L. Pfleeger, *Software Metrics - A Rigorous And Practical Approach*. PWS Publishing, 1998.

[8] D. Coleman, B. Lowther, and P. Oman, "The application of software maintainability models in industrial software systems," *Journal of Systems and Software*, vol. 29, no. 1, pp. 3–16, 1995.

[9] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[10] B. Nejmeh, "NPATH: a measure of execution path complexity and its applications," *Communications of the ACM*, vol. 31, no. 2, 1988.

[11] B. Henderson-Seller, *Object-oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[12] L. Rosemberg, "Metrics for object oriented environment," in *Proc. EFAITP/AIE 3rd Annual Software Metrics Conference*, 1997.

[13] L. H. Rosemberg, "Applying and interpreting object oriented metrics," Software Assurance Technology Center at NASA Goddard Space Flight Center, Tech. Rep., 2001.

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

[15] S. Benlarbi, K. El Emam, N. Goel, and S. Rai, "Thresholds for object-oriented measures," in *Proc. 11th Int'l Symp. Software Reliability Engineering (ISSRE 2000)*. San Jose, CA, USA: IEEE, Oct. 2000, pp. 24–38.

[16] K. Erni and C. Lewerentz, "Applying design-metrics to object-oriented frameworks," in *Proc. 3rd Int'l Software Metrics Symposium*. Berlin, Germany: IEEE, Mar. 1996, pp. 64–74.

[17] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-laws in a large object-oriented software system," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 687–708, Oct. 2007.

[18] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, pp. 1–26, Sep. 2008.

[19] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," in *Proc. 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*. IEEE, 2005, pp. 45–54.

[20] P. Oliveira, M. Valente, and F. Paim Lima, "Extracting relative thresholds for source code metrics," in *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014)*. Antwerp, Belgium: IEEE Computer Society, Feb. 2014, pp. 254–263.

[21] P. Oliveira, F. Lima, M. Valente, and A. Serebrenik, "RTTool: A tool for extracting relative thresholds for source code metrics," in *IEEE International Conference on Software Maintenance and Evolution (ICSME 2014)*. Victoria, BC, Canada: IEEE Computer Society, Sep. 2014, pp. 629–632.

[22] 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, pp. 244–257, 2012.

[23] M. Foucault, M. Palyart, J.-R. Falleri, and X. Blanc, "Computing contextual metric thresholds," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*. Gyeongju, Republic of Korea: ACM, 2014, pp. 1120–1125.

[24] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative analysis of evolving software systems using the gini coefficient," in *IEEE International Conference on Software Maintenance (ICSM 2009)*. Edmonton, AB, Canada: IEEE Computer Society, Sep. 2009, pp. 179–188.

[25] A. Serebrenik and M. van den Brand, "Theil index for aggregation of software metrics values," in *IEEE International Conference on Software Maintenance (ICSM 2010)*. Timisoara, Romania: IEEE Computer Society, Sep. 2010, pp. 1–9.

[26] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Software quality metrics aggregation in industry," *Journal of Software: Evolution and Process*, vol. 25, no. 10, pp. 1117–1135, 2013.

[27] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *Empirical Software Engineering*, vol. 16, no. 6, pp. 812–841, 2011.

[28] F. Arcelli Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code smell detection: towards a machine learning-based approach," in *Proc. 29th IEEE International Conference on Software Maintenance (ICSM 2013), ERA Track*. Eindhoven, The Netherlands: IEEE, Sep 2013, pp. 396–399.

[29] F. Arcelli Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, pp. 5:1–38, Aug. 2012.

[30] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan, "How does context affect the distribution of software maintainability metrics?" in *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013)*, Eindhoven, The Netherlands, Sep. 2013, pp. 350–359.

[31] C. Couto, C. Maffort, R. Garcia, and M. T. Valente, "COMETS: A dataset for empirical research on software evolution using source code metrics and time series analysis," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 1, pp. 1–3, Jan. 2013.

[32] M. Nagappan, T. Zimmermann, and C. Bird, "Representativeness in software engineering research," Microsoft Research, Tech. Rep. MSR-TR-2012-93, September 2012.

[33] M. Lorenzen and J. Kidd, *Object-oriented software metrics: a practical guide*. PTR Prentice Hall, 146.

[34] R. Marinescu, "Measurement and Quality in Object Oriented Design," Doctoral Thesis, Politehnica University of Timisoara, 2002.

[35] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

APPENDIX A
CODE SMELLS DEFINITIONS

We provide the definition of the smells we have considered in this paper according to the definitions proposed in the book [2], plus the definition for the Message Chain smell [1]. We considered also this smell in our experiments; we defined also metrics for its detection (see Table VII).

God Class: The God Class design flaw refers to classes that tend to centralize the intelligence of the system. A God Class performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes. This has a negative impact on the reusability and the understandability of that part of the system.

Data Class: Data Classes are "dumb" data holders without complex functionality but other classes strongly rely on them. The lack of functionally relevant methods may indicate that related data and behavior are not kept in one place; this is a sign of a non-object-oriented design. Data Classes are the manifestation of a lacking encapsulation of data, and of a poor data-functionality proximity.

Brain Method: Often a method starts out as a "normal" method but then more and more functionality is added to it until it gets out of control, becoming hard to maintain or

Table VII
ADDITIONAL EVALUATED METRICS

Quality Dimension	Metric Label	Metric Name	Granularity
Size	**LOCNAMM**	Lines of Codes Without Accessor or Mutator Methods	Class
	NOPK	Number of Packages	Project
	NOCS	Number of Classes	Project, Package
	NOM	Number of Methods [33]	Project, Package, Class
	NOMNAMM	Number of Not Accessor or Mutator Methods	Project, Package, Class
	NOA	Number of Attributes	Class
Complexity	**WMCNAMM**	Weighted Methods Count of Not Accessor or Mutator Methods	Class
	AMW	Average Methods Weight [2]	Class
	AMWNAMM	Average Methods Weight of Not Accessor or Mutator Methods	Class
	CLNAMM	Called Local Not Accessor or Mutator Methods	Method
	NOP	Number of Parameters	Method
	ATLD	Access to Local Data	Method
	NOLV	Number of Local Variable [34]	Method
Coupling	FANOUT	- [2]	Class, Method
	FANIN	-	Class
	FDP	Foreign Data Providers [2]	Method
	RFC	Response for a Class [35], [7]	Class
	CBO	Coupling Between Objects Classes	Class
	CFNAMM	Called Foreign Not Accessor or Mutator Methods	Class, Method
	MaMCL	Maximum Message Chain Length	Method
	MeMCL	Mean Message Chain Length	Method
	NMCS	Number of Message Chain Statements	Method
Encapsulation	LAA	Locality of Attribute Accesses [2]	Method
Inheritance	DIT	Depth of Inheritance Tree [35]	Class
	NOI	Number of Interfaces	Project, Package
	NOC	Number of Children [35]	Class
	NMO	Number of Methods Overridden [33]	Class
	NIM	Number of Inherited Methods [33]	Class
	NOII	Number of Implemented Interfaces	Class

understand. Brain Methods tend to centralize the functionality of a class, in the same way as a God Class centralizes the functionality of an entire subsystem, or sometimes even a whole system.

Dispersed Coupling: This is the case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes. In other words, this is the case where a single operation communicates with an excessive number of provider classes, whereby the communication with each of the classes is not very intense, i.e., the operation calls one or a few methods from each class.

Shotgun Surgery: Not only outgoing dependencies cause trouble, but also incoming ones. This design disharmony means that a change in an operation implies many (small) changes to a lot of different operations and classes. This disharmony tackles the issue of strong afferent(incoming) coupling and it regards not only the coupling strength but also the coupling dispersion.

Message Chain: You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, and so on. You may see these as a long line of getThis methods, or as a sequence of temps.

APPENDIX B
ADDITIONAL METRICS FOR SMELL DETECTION

Table VII lists all the other metrics, other than those listed in Table I, we computed and for which we derived thresholds. We report the quality dimension it is related to each metric, and the kind of software elements it has been applied to (Granularity).

Metrics in bold are metrics we have defined for code smell detection. We think these metrics capture useful features for smell detection. We include other metrics defined in the literature for software quality assessment.

In our metrics, we often exclude accessor methods (getters and setters). We think that these methods often create noise in metrics extraction, especially regarding complexity and size measures, since they are usually created and perceived as generated code.