


On the proposal and evaluation of a benchmark-based threshold derivation method

Gustavo Vale^{1,2}  · Eduardo Fernandes^{1,3}  ·
Eduardo Figueiredo¹ 

Published online: 1 May 2018

© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract Software-intensive systems have been growing in both size and complexity. Consequently, developers need better support for measuring and controlling the software quality. In this context, software metrics aim at quantifying different software quality aspects. However, the effectiveness of measurement depends on the definition of reliable metric thresholds, i.e., numbers that characterize a metric value as critical given a quality aspect. In fact, without proper metric thresholds, it might be difficult for developers to indicate problematic software components for correction, for instance. Based on a literature review, we have found several existing methods for deriving metric thresholds and observed their evolution. Such evolution motivated us to propose a new method that incorporates the best of the existing methods. In this paper, we propose a novel benchmark-based method for deriving metric thresholds. We assess our method, called Vale’s method, using a set of metric thresholds derived with the support of our method, aimed at composing detection strategies for two well-known code smells, namely god class and lazy class. For this purpose, we analyze three benchmarks composed of multiple software product lines. In addition, we demonstrate our method in practice by applying it to a benchmark composed of 103 Java open-source software systems. In the evaluation, we compare Vale’s method to two state-of-the-practice threshold derivation methods selected as a baseline, which are Lanza’s method and Alves’ method. Our results

✉ Gustavo Vale
vale@fim.uni-passau.de

Eduardo Fernandes
eduardofernandes@dcc.ufmg.br

Eduardo Figueiredo
figueiredo@dcc.ufmg.br

¹ Department of Computer Science, Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil

² Department of Computer Science, University of Passau, Passau, Germany

³ Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil

suggest that the proposed method provides more realistic and reliable thresholds, with better recall and precision in the code smell detection, when compared to both baseline methods.

Keywords Software metric · Threshold · Benchmark · Software product lines · Code smell

1 Introduction

Since software-intensive systems have been growing in both size and complexity, developers require better support for measuring and controlling the software quality (Fenton 1991; Gamma et al. 1995). In this context, software metrics may guide developers in assessing different software quality aspects, such as maintainability and changeability (Chidamber and Kemerer 1994; Lorenz and Kidd 1994). Therefore, we may improve the quality of software-intensive systems by using metrics related to different software entities, such as classes, methods, and concerns. However, the effectiveness of measurement directly depends on the definition of reliable thresholds, i.e., numbers that characterize a metric value as critical for a specific quality aspect (Oliveira et al. 2014). That is, a single metric value may not suffice to draw a conclusion on the quality of a software entity and the whole system.

In order to visualize the state-of-the-art on threshold derivation, we have conducted an ad hoc literature review of methods for deriving metric thresholds. We aimed at understanding the strengths and weaknesses of the existing methods. Based on this review, we observed that threshold derivation first relied on the opinion of developers (McCabe 1976; Nejme 1988). Second, it evolved to rely on the consensus of developers (Coleman et al. 1995). Third, methods for threshold derivation emerged and relied on data from a single software system (Erni and Lewerentz 1996). Finally, as from the first to the second case, software engineers started to use a set of software systems, so-called benchmarks (Alves et al. 2010; Ferreira et al. 2012; Oliveira et al. 2014). In a benchmark-based method, we derive thresholds based on data from similar systems, i.e., systems that share the same programming language, domain, or size scale, for instance (Mori et al. 2018). From the ad hoc review, we propose a method that combines the main strengths of existing methods (Vale and Figueiredo 2015).

The proposed method, called Vale's method, is a benchmark-based threshold derivation method composed of five well-defined steps. It derives metric thresholds that allow developers to characterize a metric value into five labels: *very low*, *low*, *moderate*, *high*, and *very high*. Additionally, Vale's method respects the metric distribution because it does not correlate metrics or assign weights to the analyzed metrics, for instance. To assess the reliability of the thresholds derived by Vale's method, we compare its results with the thresholds derived by two state-of-the-practice methods, namely Lanza's method (Lanza and Marinescu 2006) and Alves' method (Alves et al. 2010). We selected these two methods because both are benchmark-based, composed of well-defined steps, provide thresholds in a step-wise format, and have interesting particularities to discuss. For instance, Lanza's relies on mean and standard deviation to derive thresholds, while Alves' correlates metrics for the same goal.

This paper evaluates our method in two ways. First, we compare our method with Lanza's method and Alves' method using software product lines (SPL). At this evaluation, we build three SPL benchmarks and explore specific characteristics of the three threshold derivation methods. We also evaluate each method by using the derived thresholds to detect two code anomalies: god class and lazy class. God class is a class with too much knowledge and responsibility in the system and lazy class is a class with too little knowledge and responsibility

(Fowler 1999). In addition, we demonstrate the use of our method in practice by comparing the three threshold derivation methods with larger, object-oriented systems. Some of these systems are well-known and have several clients, such as ArgoUML, Hibernate, and JavaCC.

Our results in the first evaluation suggest that Vale's method provides better recall and precision than the baseline methods. In our generalizability study, we observe lower and more balanced thresholds derived by Vale's than derived by the two other methods. As lessons learned, we present a list of eight desirable points for threshold derivation methods. In summary, we learned that methods should (i) be systematic and deterministic, (ii) derive thresholds in a step-wise format, (iii) be weakly dependent on the number of systems, (iv) be strongly dependent on the number of entities, (v) not correlate metrics, (vi) compute upper and lower thresholds, (vii) provide representative thresholds regardless of the metric distribution, and (viii) provide tool support. Finally, we observe varying thresholds per benchmark, which suggests that threshold derivation is sensitive to the system characteristics, e.g., size and complexity.

We highlight that this paper evolves our previous work (Vale and Figueiredo 2015) with four major contributions. First, it revises and generalizes our previously proposed method to allow its application on general software systems. Second, in addition to the method evaluation with SPLs, we demonstrate Vale's method in this paper with a Java benchmark composed of 103 medium-to-large-sized software systems. Third, in addition to the comparison of Vale's with Lanza's methods (Lanza and Marinescu 2006), we compare our method with Alves' method (Alves et al. 2010). Fourth, we introduce a supporting tool for Vale's method, called TDTool which stands for threshold derivation tool (Veado et al. 2016). Furthermore, we add relevant discussions to our previous work, such as the discussions that compare the two evaluations (SPL and Java benchmarks, respectively) and updated related work.

The remainder of this paper is organized as follows. Section 2 provides background information and discussion of related work. Section 3 describes Vale's method for metric threshold derivation. Section 4 introduces the supporting tool for our method. Section 5 presents a practical example of our method. Section 6 evaluates Vale's method using three SPL benchmarks, by comparing both Lanza's and Alves' methods with Vale's one. Section 7 applies all three methods to a Java benchmark. Section 8 discusses the main study decisions regarding the Vale's method proposal, the different thresholds derived per benchmark, and a theoretical comparison of four related methods. Section 9 discusses threats to the study validity. Section 10 concludes the paper and suggests future work.

2 Background and related work

This section presents the description of some metrics used in this study and an overview of methods to derive metric thresholds. Section 2.1 presents the metrics used in this study. Section 2.2 describes the protocol we followed to identify threshold derivation methods. We then group and describe the main strategies and methods to derive thresholds in the following three sections. Section 2.3 discusses thresholds derived by programming experience and metric analysis. Section 2.4 presents methods for characterizing metric distributions. Section 2.5 presents methods that are benchmark-based and consider the skewed distribution of metrics. Strategies and methods of Sects. 2.3 to 2.5 are related to the method we propose in this study. Finally, Sect. 2.6 summarizes the main results of our ad hoc literature review.

2.1 Software metrics

In this study, we use eight software metrics: coupling between objects (CBO), depth inheritance tree (DIT), lack of cohesion in methods (LCOM), number of children (NOC), response for a class (RFC), weight method per class (WMC), lines of code (LOC), and number of constant refinements (NCR). We chose these metrics because they capture different quality attributes (e.g., size, coupling, cohesion, and complexity) and, except for NCR, they are well known by the software quality community. For instance, CBO, DIT, LCOM, NOC, RFC, and WMC are six metrics that compose the CK suite (Chidamber and Kemerer 1994). Although NCR is not a well-known metric, we chose it because this metric captures an interesting view of the complexity of SPL components. We describe each of the eight metrics as follows.

- *Coupling between objects (CBO)* (Chidamber and Kemerer 1994) counts the number of classes called by a given class. CBO measures the degree of coupling among classes. Figure 1a illustrates an example of how CBO is calculated. In this example, each box represents classes and each arrow represents the relation between two classes.
- *Depth of inheritance tree (DIT)* (Chidamber and Kemerer 1994) counts the number of levels a subclass inherits methods and attributes from a superclass in the inheritance tree. It is another metric to estimate the class complexity/coupling. Figure 1b presents an example of DIT computation. As can be seen in Fig. 1b, if a class has DIT = 0, the subclass of this class has DIT = 1 and so on.
- *Lack of cohesion in methods (LCOM)* (Chidamber and Kemerer 1994) measures the cohesion of methods of a class in terms of the frequency that they share attributes. LCOM is calculated by subtracting the number of method pairs that share an attribute access from the number of method pairs that do not share any attribute access. Figure 1c shows an example of how LCOM is computed. For instance, M1 and M2 are methods that share the use of A1 attribute. However, M3 does not share any attribute access with M1 or M2. Therefore, LCOM is 1 in this example.
- *Number of children (NOC)* (Chidamber and Kemerer 1994) counts the number of direct sub-classes of a given class. This metric indicates code reuse. Figure 1d presents an example of NOC computing. For instance, the class root in the inheritance tree has two sub-classes at the level immediately below. Therefore, this class has NOC = 2. On the other hand, leaf classes have no sub-classes. Therefore, their NOC is 0.
- *Response for a class (RFC)* (Chidamber and Kemerer 1994) counts the number of methods in a class that may be executed when an object of the given class receives a message. This metric supports the assessment of class complexity. Figure 1e presents an example of RFC. In this figure, *class1* implements two methods, and it calls three methods from other classes. These five methods may be potentially called in objects of this class. Therefore, RFC is 5 for *class1*.
- *Weighted method per class (WMC)* (Chidamber and Kemerer 1994) counts the number of methods in a class. This metric can be used to estimate the complexity of a class. Each method weights equal one in this study. Figure 1f illustrates how WMC is computed. For each method in a class, the value of WMC is incremented. Therefore, in case of Fig. 1f, there are three methods and, hence, WMC is 3.
- *Lines of code (LOC)* (Lorenz and Kidd 1994) counts the number of uncommented lines of code per class. The value of this metric indicates the size of a class. Figure 1g presents an

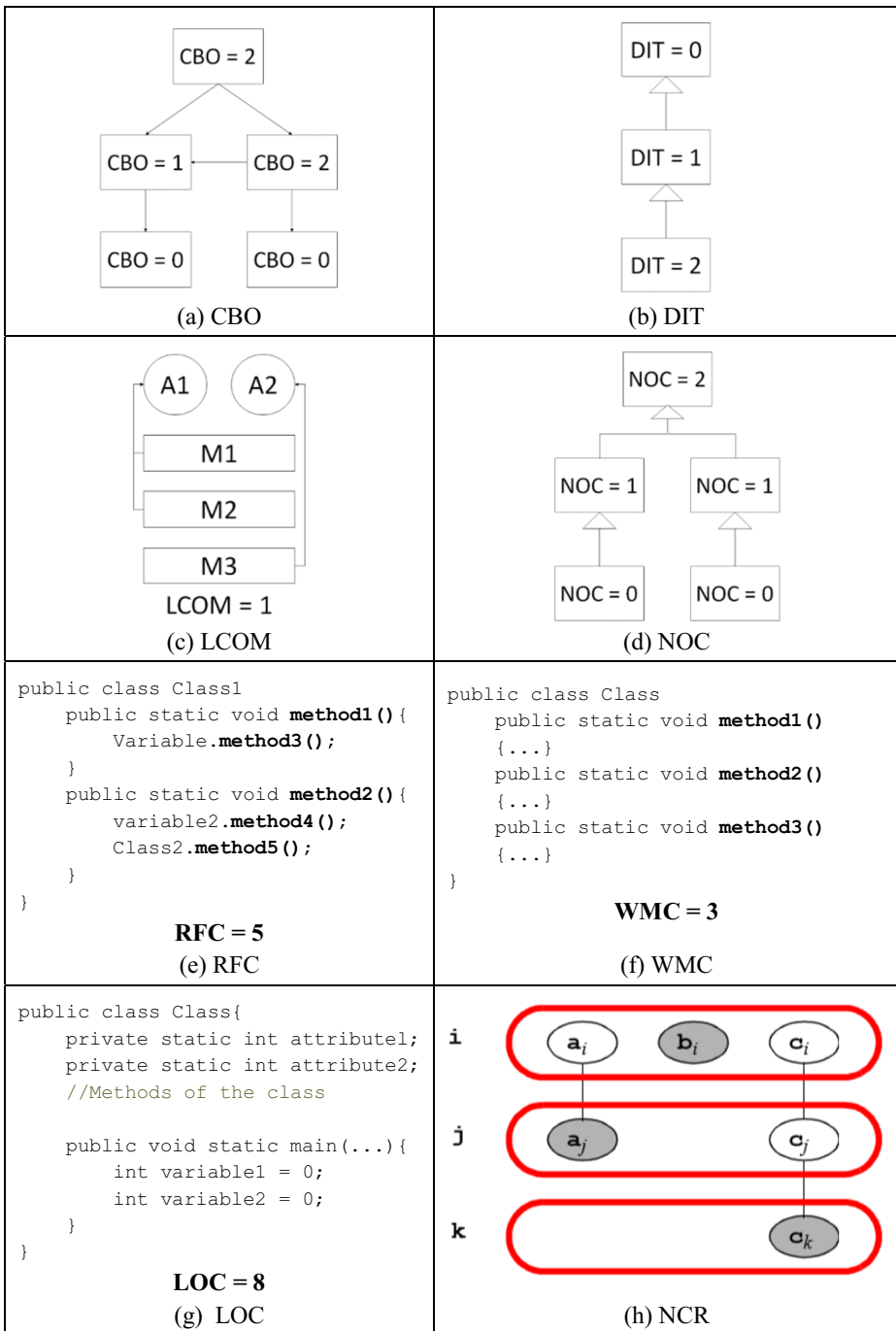


Fig. 1 Examples of computing metrics

illustrative example of LOC. As can be seen in this example, LOC counts code lines, but LOC ignores either commented lines or blank lines. Hence, in Fig. 1g, LOC is equal to 8.

- *Number of constant refinements (NCR)* (Abilio et al. 2016) counts the number of refinements that a constant has. NCR indicates how complex the relationship between a constant and its features is. Constants and refinements are files that can often be found in feature-oriented programming (FOP) (Batory and O'Malley 1992). That is, refinements can change the behavior of a constant (such as a class) if a certain feature is included in a product. Figure 1h presents an example of NCR. In Fig. 1h, we have the features i, j , and k ; classes a, b , and c ; a_i, b_i , and c_i are constants of feature i ; and a_j, c_j , and c_k are refinements of features j and k . Therefore, constants a_i, b_i , and c_i have NCR equals to 1, 0, and 2, respectively.

2.2 Literature review protocol

This section presents the protocol of a literature review to identify methods to derive thresholds. To identify such methods, we performed an ad hoc literature review. We opted for an ad hoc literature review because we have found a systematic literature review (SLR) (Lima et al. 2016) with a similar purpose of our review. An SLR is a well-defined method to identify, evaluate, and interpret all relevant studies regarding a particular research question, topic area, or the phenomenon of interest (Kitchenham and Charters 2007). The purpose of the related SLR (Lima et al. 2016) is to identify papers that present and discuss metric thresholds. Hence, as we are interested in methods to derive thresholds, we assume that they identify such methods or, at least, present strategies and papers related to this context.

Therefore, we used the related SLR (Lima et al. 2016) as our starting point. This SLR reports 19 papers that present and discuss thresholds. We have not only analyzed all these 19 papers, but also performed the *snowballing* technique. This technique consists in investigating the references retrieved in the selected papers to find additional relevant papers to increase the scope of the search, providing broader results (Brereton et al. 2007). Hence, after applying the forward snowballing, we got an amount of 50 primary studies. These primary studies were selected following four inclusion criteria: (i) the paper must be in computer science area, (ii) the paper must be written in English, (iii) the paper must be available in electronic form, and (iv) the paper must propose or use at least one method to derive metric thresholds. The list of primary studies can be found on the supplementary website (LabSoft 2017).

2.3 Thresholds derived from programming experience and metric analysis

Many authors defined metric thresholds according to their programming experience. For example, the values 10 and 200 were defined as thresholds for *McCabe* (1976) and *NPATH* (Nejmeh 1988), respectively. The aforementioned values are used to indicate the presence (or absence) of code smells. Regarding *Maintainability Index* (MI), the values 65 and 85 are defined as thresholds (Coleman et al. 1995). When MI values are higher than 85, between 85 and 65, or smaller than 65, they are considered as highly maintainable, moderately maintainable, and difficult to maintain, respectively. These thresholds rely on programming experience and they are difficult to reproduce or generalize. Additionally, the lack of scientific support can lead to disagreement about the values. However, unlike these papers (Coleman et al. 1995; McCabe 1976; Nejmeh 1988), our research does not aim to propose a method to derive thresholds based on programming experience.

Erni and Lewerentz (1996) propose the use of mean (μ) and standard deviation (σ) to derive a threshold (T) from software data. A threshold is calculated as $T = \mu + \sigma$ and $T = \mu - \sigma$ when high and low values of a metric indicate potential design problems, respectively. Lanza and Marinescu (2006) use a similar method in their research for 45 Java projects and 37 C++ projects. Nevertheless, they use four labels: low, mean, high, and very high. Labels low, mean, and high is calculated in the same way as Erni and Lewerentz (1996). Label very high is calculated as $T = (\mu + \sigma) \times 1.5$. Abilio et al. (2015) use the same method than Lanza and Marinescu (2006), but they derive thresholds for eight SPLs. These methods rely on a common statistical technique. However, Erni and Lewerentz (1996), Abilio et al. (2015), and Lanza and Marinescu (2006) do not analyze the underlying distribution of metrics. The problem with these methods is that they assume that metrics are normally distributed, limiting the use of these methods. In contrast, our research focuses on a method that does not make assumptions about data normality.

French (1999) also proposes a method based on the mean and standard deviation. However, French used the Chebyshev's inequality theorem (whose validity is not restricted to normal distributions). A metric threshold T can be calculated by $T = \mu + k \times \sigma$, where k is the number of standard deviations. Additionally, this method is sensitive to the large number of outliers (Mori et al. 2018). For metrics with high range or high variation, this method identifies a smaller percentage of observations than its theoretical maximum. Herbold et al. (2011) propose a method for threshold derivation that does not depend on the context of the collected metrics, e.g., the target programming language or abstraction level. For this purpose, the authors rely on machine learning and data mining techniques. Finally, Perkusich et al. (2015) propose a method to support the interpretation of values for software metrics. Instead of deriving thresholds that indicate acceptable values for a metric, the proposed method relies on Bayesian networks that consider subjective factors of software development. Thus, their method aims to support managers in minimizing wrong decisions based on software measurement and assessment. In contrast to French (1999), Herbold et al. (2011), and Perkusich et al. (2015), our method was designed to derive thresholds from benchmark data and, as such, it is resilient to high variation of outliers. In addition, we did not use Chebyshev's inequality theorem, machine learning, or Bayesian networks.

2.4 Methods for characterizing metric distributions

Chidamber and Kemerer (1994) use histograms to characterize and analyze data. For each of their 6 metrics (e.g., WMC and CBO), they plotted histograms per programming language to discuss metric distribution and spotted outliers in C++ and Smalltalk systems. Spinellis (2008) compares metrics of four operating system kernels (i.e., Windows, Linux, FreeBSD, and OpenSolaris). For each metric, boxplots of the four kernels are put side-by-side showing the smallest observation, lower quartile, median, mean, higher quartile, and the highest observation and identified outliers. The boxplots are then analyzed by the author and used to give ranks, + or - to each kernel. However, as the author states, the ranks are given subjectively. Vasa et al. (2009) propose the use of *Gini coefficients* to summarize a metric distribution across a system. The analysis of the Gini coefficient for 10 metrics using 50 Java and C# systems revealed that most of the systems have common values. Moreover, higher Gini coefficient values indicate problems and, when analyzing subsequent releases of source code, a difference higher than 0.04 indicates significant changes in the code. In contrast to Chidamber and Kemerer (1994), Spinellis (2008), and

Vasa et al. (2009), we did not use histograms, mean, median, or Gini coefficient to calculate thresholds and we derive thresholds based on data from a benchmark.

2.5 Benchmark-based methods that consider the skewed metric distribution

This section describes methods closer to ours because they are transparent, the thresholds are extracted from benchmark data, and the methods consider the skewed distribution of metrics. Alves et al. (2010) propose a method that weights software metrics by lines of code and aim at labeling each entity of a system based on thresholds. Each label is defined based on a pre-determined percentage of entities. This method proposes 70, 80, or 90% to represent the following labels: *low* (between 0 and 70%), *moderate* (70–80%), *high* (80–90%), and *very high* (> 90%). Similarly, Ferreira et al. (2012) present a method for calculating thresholds. This method consists in grouping the extracted metrics in a file and gets three groups, with high, medium, and low frequency. The groups are called good, regular, and bad measurements, respectively. The authors do not make clear how to extract the three groups since they argue that the groups rely on visual analysis. Oliveira et al. (2014) propose a method based on a formula, named *compliance rate*. The main differences of this method are that it extracts relative thresholds instead of absolute ones and it does not label components. Oliveira's method defines one threshold for software metric per system (Oliveira et al. 2014). In contrast to Alves, Ferreira, and Oliveira, our method does not correlate metrics, we present different labels in a step-wise format, and our method has lower bound thresholds.

2.6 Literature review summary

By analyzing all the 50 primary studies, we could see that researchers have been worried about thresholds for a long time once the first published paper identified in our review is from 1976. In addition, this topic is still opened because researchers nowadays propose methods to derive thresholds. Despite it being an open topic, we can see an evolution in such methods. This evolution can be summarized in three key points: (i) to be derived from benchmarks, (ii) to be based on well-defined methods, and (iii) to consider the skewed metric distribution.

The first key point regards the confidence of the derived thresholds. In the past, software engineers derived thresholds through their subjective opinion (McCabe 1976; Nejme 1988). After, they started to discuss in groups to get a consensus about the thresholds (Coleman et al. 1995). Later, software engineers started to use systems to help them to derive thresholds (Erni and Lewerentz 1996). Finally, as from the first to the second case, they started to use a group of software systems, called benchmarks, to support the threshold derivation (Ferreira et al. 2012; Oliveira et al. 2014). The idea of using benchmark-based methods is to get common characteristics of entities from a benchmark and assume that discrepant values might have a problem. In this context, benchmark means a set of similar systems, e.g., developed in the same programming language, domain, or with similar size or type.

The second key point is related to the method replicability. A method should output the same threshold for a metric when the same input is provided. Therefore, the method should be deterministic. Consequently, deterministic methods prevent subjective decisions in threshold derivation and they can be automatically performed. Hence, huge benchmarks can be used as input increasing the scope and reliability of the derived thresholds.

The third key point is related to the statistical approach used by methods to derive thresholds. This point is important because software metrics can have different distributions,

such as normal, power law (heavy-tail), and common distributions. A method using only mean and median can provide invalid or non-representative thresholds for non-normal distributions. Previous works assumed that software metrics follow a normal distribution (Chidamber and Kemerer 1994; Erni and Lewerentz 1996). Despite that assumption, several studies (Concas et al. 2007; Ferreira et al. 2012; Louridas et al. 2008; Oliveira et al. 2014) clearly demonstrate that most software metrics do not follow normal distributions, limiting the use of methods that use mean to derive thresholds, for example.

Therefore, based on the results of this literature review, we conclude that methods should be benchmark-based, transparent, straightforward, replicable, and respect skewed metric distribution. Hence, if they respect these guidelines, methods tend to provide more reliable thresholds.

3 The proposed method to derive thresholds

The method proposed in this section was designed according to the following guidelines: (i) it should be based on data analysis from a representative set of systems (benchmark); (ii) it should be a strong dependence on the number of entities; (iii) it should be a weak dependence on the number of systems; (iv) it should calculate upper and lower thresholds; (v) it should derive thresholds in a step-wise format; (vi) it should respect the statistical properties of the metric; and (vii) it should be systematic, repeatable, transparent, and straightforward to execute. These guidelines are mainly based on the desirable points described in a previous work (Vale et al. 2015). Figure 2 summarizes the steps of the proposed method, called Vale’s method from now on.

Vale’s method steps are described as follows.

1. *Measurement*: metrics are collected from a benchmark of software systems (input). We record a metric value, for each system and for each entity (e.g., class) belonging to the system. Hence, if we think in a spreadsheet, each row indicates an entity metric value and each column indicates a metric (e.g., LOC, CBO, and so on).

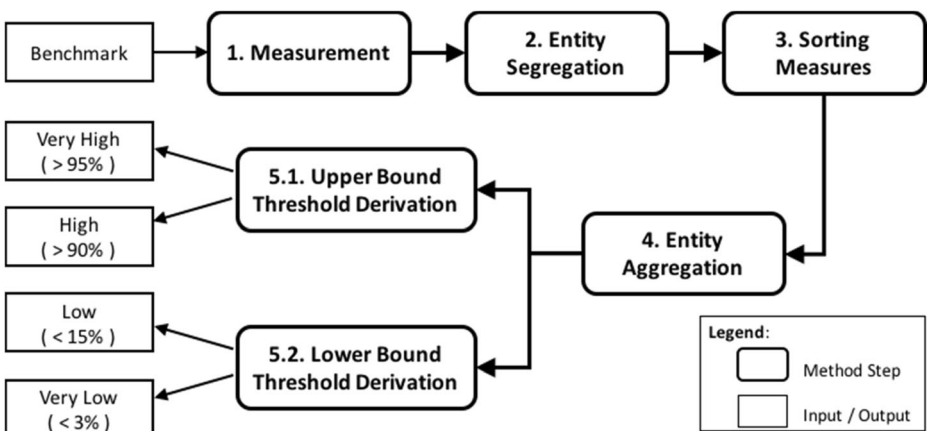


Fig. 2 Summary of Vale’s method steps

2. *Entity segregation*: for each entity, we compute its percentage with respect to the total number of entities, i.e., we divide 100 by the total number of entities. That is, every entity represents the same amount of the benchmark in our method. In other words, each entity has the same weight. The sum of all entities must be 100%. For instance, if one benchmark has 10,000 entities, each entity represents 0.01% of the overall ($0.01\% \times 10,000 = 100\%$).
3. *Sorting measures*: we order the metric values in ascending order and take the maximal metric value that represents 1, 2, up to 100%, of the weight. This step is equivalent to computing a density function, in which the x-axis represents the weight ratio (0–100%) and the y-axis the metric scale. For instance, all entities with WMC value that is 4 must come before all metrics in which WMC value is 5.
4. *Entity aggregation*: we aggregate all entities per metric value, which is equivalent to computing a weighted histogram (the sum of all bins must be 100%). For example, if we have four entities with WMC value equal to 4 and each entity representing 0.01%, the aggregated entities correspond to 0.04% of all code.
5. *Upper Bound Threshold Derivation*: upper bound thresholds are derived by choosing the 90 and 95% percentages of the overall metric values we want to represent. We propose the use of 90% to identify “high” metric values and 95% to identify “very high” metric values. Let us suppose that to represent 90% of the overall code for the WMC metric, the derived threshold is 18. This threshold is meaningful, since not only does it mean that 90% of the code of a benchmark of systems is below 18, but it also can be used to identify the top 10% of the worst code in terms of WMC (greater than 18).
6. *Lower bound threshold derivation*: a unique characteristic of the proposed method compared to the most related work (see Sect. 2.5) is that it can extract lower bound thresholds, in addition to upper bound thresholds. To extract lower bound thresholds, we propose the percentiles of 3% (very low) and 15% (low). We decided to create four different labels to give more power to the user. For instance, these labels allow identifying metrics value to be fixed in long term, medium term, and short term.

In summary, the percentiles used in the proposed method can be used to characterize metric values according to five categories (which are not mutually exclusive): *very low* values (between 0 and 3%), *low* values (0–15%), *moderate* values (15–90%), *high* values (90–100%), and *very high* values (95–100%). In Sect. 8.4, we discuss why we chose these percentages and labels.

4 TDTool: threshold derivation tool

This section presents the TDTool (threshold derivation tool), which supports Vale’s method. The tool expects as input a set of CSV files in which each file must represent the measures of the entities of a system. In addition, it expects files in the following format: each column must represent a metric and each row must represent an entity. For a benchmark with 100 systems, 100 CSV files are expected. The results of TDTool are displayed on the screen, and they can be exported to CSV format (Veado et al. 2016). Figure 3 also presents the main three modules of TDTool. The modules are three listed as follows: configuration, processing, and presentation. In the *configuration* module, the user must select the files in which composes the benchmark, and after, he should select metrics he wants to derive thresholds. The *processing* module is

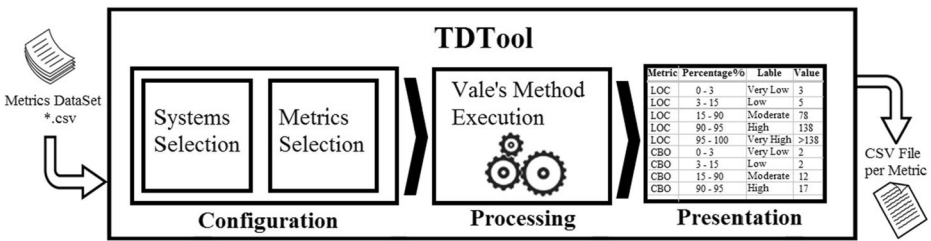


Fig. 3 TDTool architecture

responsible for deriving the thresholds, as described by the proposed method in Sect. 3. The *presentation* module shows the results as a table that summarizes the derived thresholds.

Table 1 represents an example of an expected file to derive thresholds for four metrics (LOC, CBO, WMC, and NCR) in part of the TankWar source code. TankWar is a game developed by students at German University of Magdeburg as a software product line (SPL) to adhere to portability requirements common to mobile devices (Schulze et al. 2010). In this case, the entities represent the classes of the system. TDTool does not depend on the way the metrics have been calculated. Instead, it only needs to receive as input the file in the expected CSV format for deriving the metric thresholds.

Figure 4 presents the selection metrics view. In this illustration, TDTool identified component, LOC, CBO, WMC, and NCR as metrics. Despite that, the first column represents the entities' name. Therefore, we did not want to derive thresholds for this column. Hence, we selected only LOC, CBO, WMC, and NCR. For each derivation step, the user can close the tool, return to the previous step, or go to the next one. In the final step, the user can export the results by clicking on the *Save* button. The results are arranged in a CSV file containing all the entities with their respective percentage. To evaluate the thresholds provided by TDTool, we compared the thresholds obtained

Table 1 Example of input for TDTool

Component	LOC	CBO	WMC	NCR
Maler.jak	8	2	1	0
Option.jak	27	2	3	0
Maler.jak	300	12	63	16
TankManager.jak	11	2	4	0
Menu.jak	254	7	66	0
KeyMonitor.jak	63	4	28	0
GameManager.jak	2	1	0	0
Missile.jak	6	2	1	0
Tank.jak	6	2	1	0
ExplodierenEffekt.jak	126	3	18	0
TankManajer.jak	6	2	2	0
Maler.jak	321	8	104	0
Tank.jak	15	3	3	0
Missile.jak	13	4	3	0
Tank.jak	41	4	13	0
TankManager.jak	18	4	5	0
SoundPlayer.jak	51	5	11	0
Tank.jak	49	3	16	0
Sprach.jak	18	1	0	0
Sprach.jak	18	1	0	0

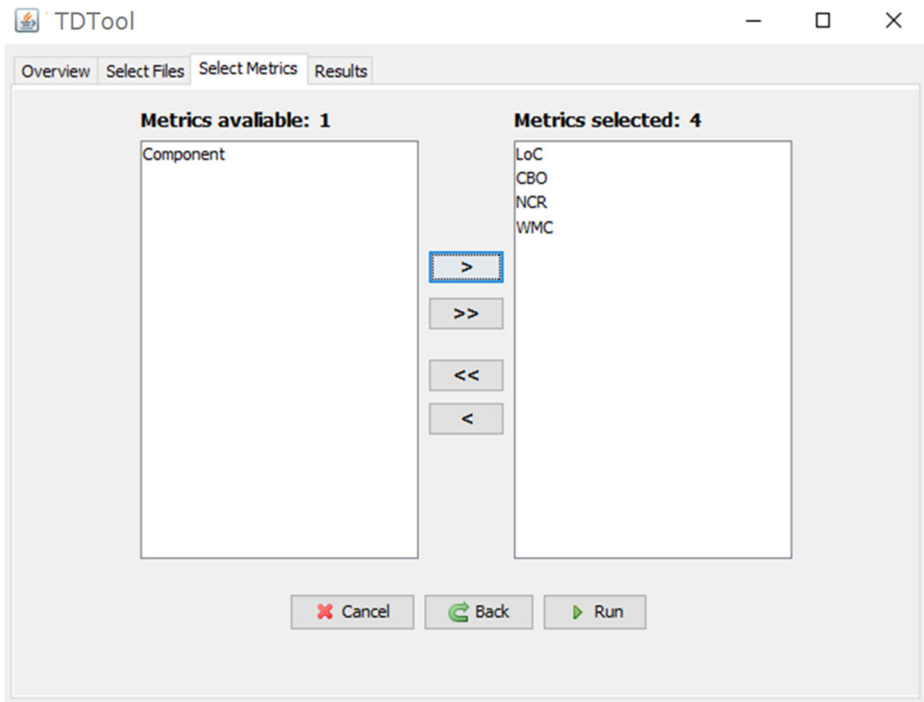


Fig. 4 Metric selection in TDTool

manually with the thresholds obtained by the tool; the thresholds are the same. Therefore, we believe that TDTool works correctly.

5 Example of use

Vale's method can be applied in different ways, such as using SIG quality model (Heitlager et al. 2007), analyzing metrics individually, or using a metric-based detection strategy. The SIG quality model combines metrics that capture source code properties until those metrics represent some characteristics of International Standards Organizations (ISO), such as ISO/IEC 9126 and ISO/IEC 25000. In this section, we illustrate the use of Vale's method by analyzing individually the derived thresholds of four metrics.

Before applying the method, it is necessary to follow three previous steps: (i) to have a benchmark composed of software systems, (ii) to choose a set of metrics to derive thresholds, and (iii) to choose a tool able to extract these metric values from each system of the target benchmark. Section 5.1 explains how we built three SPL benchmarks. We selected four metrics (LOC, CBO, WMC, and NCR), described in Sect. 2.1 to derive metric thresholds. We then extracted the value for these four metrics for each class of each system from these three SPL benchmarks using VSD tool (Abilio et al. 2014). Section 5.2 presents the derived thresholds for the chosen set of metrics obtained by using Vale's method (Sect. 3).

5.1 Software product line benchmarks

This section presents three benchmarks of software product lines (SPLs) shown in Table 2. An SPL is a configurable set of systems that share a common, managed set of features in a particular market segment (SEI 2016). Features can be defined as modules of an application with consistent, well-defined, independent, and combinable functions (Apel et al. 2009). We decided to build SPL benchmarks because SPLs have been increasingly adopted in the software industry to support coarse-grained reuse of software assets (Dumke and Winkler 1997). To build these benchmarks, we focused on SPLs developed using FOP (Batory and O’Malley 1992). The main reason for choosing FOP is because this technique aims to support modularization of features, i.e., the building blocks of an SPL. In addition, we have already developed a tool, named *variability smell detection* (VSD) (Abilio et al. 2014), which is able to measure FOP code (step one of the proposed method).

We selected 47 SPLs from repositories, such as SPL2GO (2015) and FeatureIDE (2017) examples, and 17 SPLs from research papers, summing up to 64 SPLs in total. In order to have access to the source code of each SPL, we either e-mail the paper authors

Table 2 Software product lines benchmarks

	Id	SPL	Tech.	LOC	
Benchmarks 1, 2, and 3	1	BerkeleyDB (SPL2GO 2015)	FH-Java	37,247	
	2	AHEAD-Java (Abilio et al. 2014)	AHEAD	16,719	
	3	AHEAD-guidsl (Abilio et al. 2014)	AHEAD	8738	
	4	TankWar (FeatureIDE 2017; SPL2GO 2015)	AHEAD	4670	
	5	AHEAD-Bali (Abilio et al. 2014)	AHEAD	3988	
	6	Devolution (FeatureIDE 2017)	AHEAD	3913	
	7	MobileMedia v.7 (Ferreira et al. 2014)	AHEAD	2691	
	8	WebStore v.6 (Ferreira et al. 2014)	AHEAD	2082	
	9	DesktopSearcher (FeatureIDE 2017; SPL2GO 2015)	AHEAD	1858	
	10	GPL (FeatureIDE 2017)	AHEAD	1824	
	11	Notepad v.2 (SPL2GO 2015)	FH-Java	1667	
	12	Vistex (SPL2GO 2015)	FH-Java	1480	
	13	GameOfLife (SPL2GO 2015)	FH-Java	1047	
	14	Prop4J (SPL2GO 2015)	FH-Java	1047	
Benchmarks 1 and 2	15	Elevator (SPL2GO 2015)	FH-Java	728	
	16	ExamDB (SPL2GO 2015)	FH-JML	568	
	17	PokerSPL (SPL2GO 2015)	FH-JML	461	
	18	EmailSystem (SPL2GO 2015)	FH-Java	460	
	19	GPLscratch (SPL2GO 2015)	FH-JML	405	
	20	Digraph (SPL2GO 2015)	FH-JML	374	
	21	MinePump (SPL2GO 2015)	FH-JML	367	
	22	Paycard (SPL2GO 2015)	FH-JML	319	
	Benchmark 1	23	IntegerSet (SPL2GO 2015)	FH-JML	225
		24	UnionFind (SPL2GO 2015)	FH-JML	194
25		NumberContractOverriding (SPL2GO 2015)	FH-JML	165	
26		NumberConsecutiveContractRef (SPL2GO 2015)	FH-JML	148	
27		NumberExplicitContractRef (SPL2GO 2015)	FH-JML	143	
28		BankAccount (SPL2GO 2015)	FH-JML	122	
29		EPL (FeatureIDE 2017)	AHEAD	98	
30		IntList (SPL2GO 2015)	FH-JML	94	
31		StringMatcher (SPL2GO 2015)	FH-JML	45	
32		Stack (SPL2GO 2015)	FH-Java	22	
33		HelloWorld (FeatureIDE 2017)	AHEAD	22	

or search on the Web. In the case of SPL repositories, the source code was available. When different versions of the same SPL were found, we picked up the most recent one. Some SPLs were developed in different languages or technologies. For instance, GPL (FeatureIDE 2017) has four different versions implemented in AHEAD, FH-C#, FH-Java, and FH-JML. FH stands for FeatureHouse and FH-Java means that the SPL is implemented in Java using FeatureHouse as a composer. In cases where the SPL was implemented in more than one technique, we preferred to select the AHEAD or FeatureHouse implementation based on a tool support constraint (see Sect. 9). After filtering our original dataset by selecting only one version and one programming language for each SPL, we end up with 33 SPLs listed in Table 2. The source code of all SPLs of our benchmark and the step-to-step filtering is further explained on the supplementary website (LabSoft 2017).

In order to generate different benchmarks for comparison, we split the 33 SPLs into three benchmarks according to their size in terms of *Lines of Code* (LOC). Table 2 presents the 33 SPLs ordered by their values of LOC. This table also shows implementation technology (Tech.) per SPL and groups the SPLs by the respective benchmarks. Benchmark 1 includes all 33 SPLs. Benchmark 2 includes 22 SPLs with more than 300 LOC. Finally, benchmark 3 is composed of 14 SPLs with more than 1000 LOC. The goal of creating three different benchmarks is to analyze the results with varying levels of thresholds.

5.2 Derived thresholds

This section presents the thresholds derived by Vale’s method per benchmark presented in the previous section for LOC, CBO, WMC, and NCR metrics. Vale’s method presents five labels, but these labels are established in four percentages. Hence, Table 3 shows the values that represent the percentages (key values). This table should be read as follows: the first column represents the benchmarks, the second column indicates the different labels, and the other columns determine the thresholds of LOC, CBO, WMC, and NCR metrics, respectively. For example, the labels are defined as: *very low* (0–3%), *low* (3–15%), *moderate* (15–90%), *high* (90–95%), and *very high* (95–100%). They are represented in benchmark 1 by the intervals 0–2, 3–4, 5–77, 78–138, and > 139 LOC, respectively.

Table 3 Threshold values from the proposed method

Benchmark	%	LOC	CBO	WMC	NCR
1	3	2	1	0	0
	15	4	1	1	0
	90	77	11	17	3
	95	138	16	31	7
2	3	2	1	0	0
	15	4	1	1	0
	90	78	11	17	3
	95	142	16	32	8
3	3	2	1	0	0
	15	4	1	1	0
	90	79	12	18	3
	95	146	19	34	8

It should be observed that there is a difference between the thresholds varying the benchmark for the same label, although it is a minor difference in most cases. The thresholds by the same metric from benchmarks 1, 2, and 3 (in this order) increased. It makes sense because small SPLs were removed and both constants and refinements from SPLs whose compose benchmark 1 are generally smaller than constants and refinements from SPLs whose compose benchmarks 2 and 3. In addition, evidence can be seen that the proposed method is concerned with the entity values to derive thresholds because, in theory, the quality of the benchmarks increases.

6 Evaluation using the SPL benchmarks

This section evaluates the derived thresholds from Vale’s, Lanza’s (Lanza and Marinescu 2006), and Alves’ (Alves et al. 2010) methods. We selected these two methods to evaluate Vale’s method because both are benchmark-based methods, have well-defined steps, and present thresholds in a step-wise format like our method. In addition, we selected Lanza’s method, because it is the only benchmark-based method found in the literature which derives lower bound thresholds and we intend to analyze if the metric distribution really influences a method based just on mean and standard deviation—disrespecting the third key point (see Sect. 2.6 for details).

To perform such evaluation, we derive thresholds using Lanza’s method (Sect. 6.1) and Alves’ method (Sect. 6.2), choose two metric-based detection strategies for two code smells (Sect. 6.3), choose a target SPL and define a reference list of code smells (Sect. 6.4), and perform the comparison of effectiveness of these methods (Sect. 6.5).

6.1 Derived thresholds using Lanza’s method

As described in Sect. 2.3, Lanza’s method uses $\mu - \sigma$, μ , $\mu + \sigma$, $\mu + \sigma \times 1.5$ to the labels *low*, *mean*, *high*, and *very high*, respectively, where μ is the mean and σ is the standard deviation. Table 4 presents the thresholds necessary to represent Lanza’s method. This table should be read as follows: the first column represents the benchmarks, and the second column indicates the different labels. The other columns determine the thresholds of LOC, CBO, WMC, and NCR, respectively. For example, the labels defined as *low*, *mean*, *high*, and *very high* are

Table 4 Thresholds values from Lanza’s method

Benchmark	Label	LOC	CBO	WMC	NCR
1	Low	0	0	0	0
	Mean	36.80	5.33	8.14	1.07
	High	126	11.70	28.90	4.05
	Very high	189	17.55	43.35	6.075
2	Low	0	0	0	0
	Mean	37.40	5.44	8.30	1.07
	High	128	11.8	29.40	4.08
	Very high	192	17.7	44.10	6.12
3	Low	0	0	0	0
	Mean	37.80	5.59	8.40	1.14
	High	130	12.1	29.90	4.27
	Very high	195	18.15	44.85	6.405

Table 5 Thresholds values from Alves’ method

Benchmark	Percentage	LOC	CBO	WMC	NCR
1	70	92	9	14	1
	80	151	13	31	2
	90	252	21	58	4
2	70	127	13	25	1
	80	221	19	45	1
	90	328	24	75	5
3	70	192	18	40	1
	80	293	22	58	1
	90	442	29	84	7

represented by the values 0, 5.33, 11.7, and 17.55, respectively, for CBO in benchmark 1. Lanza’s method may provide negative thresholds to metrics, given the use of standard deviation to derive thresholds. Considering the chosen software metrics for this study, a negative value does not make sense and it requires a special treatment. Hence, we converted these negative values to zero.

6.2 Derived thresholds using Alves’ method

As described in Sect. 2.5, Alves’ method uses the percentages 70, 80, and 90 to represent the labels *moderate*, *high*, and *very high*, respectively. We should remember that Alves’ method weights by LOC all other analyzed metrics. Table 5 presents the derived thresholds for this method to the three SPL benchmarks. This table should be read like Table 4. As an example, for benchmark 1, the values 151, 13, 31, and 2 are considered high (80%) for LOC, CBO, WMC, and NCR, respectively.

6.3 Metric-based detection strategies

Despite the extensive use of metrics, they are often too fine-grained to comprehensively quantify deviations from good design principles (Lanza and Marinescu 2006). In order to overcome this limitation, metric-based detection strategies have been proposed (Marinescu 2004). A detection strategy is a composed logical condition, based on metrics and threshold values, which detects design fragments with specific code smells (Lanza and Marinescu 2006).

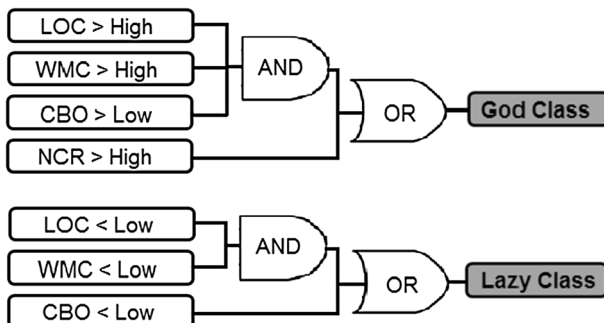


Fig. 5 Code smells detection strategy

Code smells describe a situation where there are hints that suggest a flaw in the source code (Riel 1996). This section illustrates the detection strategies of two code smells: god class and lazy class.

The literature defines god class as a class with excessive knowledge and responsibilities in the software system (Fowler 1999). We select this code smell because it is one of the most relevant and studied in the context of software maintenance (Fernandes et al. 2016; Padilha et al. 2014). In addition, we should mention that god class is a strong indicator that a software component is accumulating the implementation of many other ones (captured by NCR metric). On the other hand, lazy class is a class that has just a few responsibilities and holds little knowledge on the software system (Fowler 1999). As can be seen, lazy class is the opposite of god class.

In this work, we selected detection strategies in the literature to identify god classes and lazy classes for the following reasons. First, they have been evaluated in other studies and presented good results for the detection of god class and lazy class (Abilio et al. 2014; Vale et al. 2015). Second, these detection strategies use a straightforward way for identifying instances of god class and lazy class using four different metrics. We also believe that these strategies are better than traditional ones because they were adapted for SPL by using NCR (an FOP-specific metric), for example. This metric fits complexity properties of SPLs that traditional metrics cannot fit.

Figure 5 shows the god class and lazy class detection strategies adapted from previous work (Abilio et al. 2014; Munro 2005). LOC, CBO, WMC, and NCR refer to the metrics used in these detection strategies (presented in Sect. 2.1). The original detection strategies rely on absolute values, but we substitute these values by labels, such as *low* and *high*, to provide strategies more dependent on the derived thresholds.

6.4 Choosing the target SPL and creating a reference list of code smells

We choose an SPL called MobileMedia for this evaluation. It is an SPL for manipulating photos, music, and videos on mobile devices (Figueiredo et al. 2008). It is an open-source SPL implemented in several programming languages, such as Java, AspectJ, and AHEAD. We selected MobileMedia version 7 - AHEAD implementation (Ferreira et al. 2014). This SPL was chosen because (i) it was successfully used in other previous empirical studies (Fernandes et al. 2017; Ferreira et al. 2014; Figueiredo et al. 2008), (ii) it is part of the three benchmarks of this study, and (iii) we have access to its software developers. Table 6 presents the reference list

Table 6 Code smell reference list for MobileMedia

Code smell	Classes in the reference list
God class	<i>MainUIMidlet (Base)</i> , <i>MediaAccessor (Base)</i> , <i>MediaController (MediaManagement)</i> , <i>MediaListController (MediaManagement)</i> , <i>MediaListScreen (MediaManagement)</i> , <i>AlbumData (AlbumManagement)</i> , and <i>SmsMessaging (SMSTransfer)</i>
Lazy class	<i>Constants (AlbumManagement)</i> , <i>MediaData (SetFavourites)</i> , <i>ControllerCommandInterface (Base)</i> , <i>ControllerInterface (Base)</i> , <i>Constants (Base)</i> , <i>PhotoViewController (CaptureVideo)</i> , <i>Constants (CreateAlbum)</i> , <i>Constants (CreateMedia)</i> , <i>Constants (DeleteAlbum)</i> , and <i>Constants (MediaManagement)</i>

The first word refers to constant or refinement and the word in parenthesis is the name of the feature in which this constant or refinement is

of code smells for the MobileMedia SPL. This list includes seven god class instances and ten lazy class instances.

The reference list of code smells can be understood as the list of the actual code smells found in the MobileMedia SPL. The reference list is used for comparison of methods to derive thresholds, and it is the basis for determining whether the derived thresholds (computed by each method) are effective on the identification of code smells in a specific SPL. Aiming to provide a reliable reference list, we analyzed the source code and defined some god class and lazy class instances. This preliminary reference list has been later validated by experts, i.e., the MobileMedia developers. The final version of the reference list was produced as a joint decision.

6.5 Evaluation of the derived thresholds

This section presents the results of the evaluation of recall and precision applied to the derived thresholds of Vale's, Lanza's, and Alves' methods. These results of recall and precision refer to the MobileMedia reference list of code smells, presented in Sect. 6.4. It is important to mention that the thresholds of each method were used with the same detection strategy for each code smell (Sect. 6.3), and it was applied to the three SPL benchmarks (Sect. 5.1). As Alves' method does not provide lower bound thresholds, we analyzed just god class for this method. Hence, we defined as "not applicable (n/a)" all measures regarding lazy class for this method.

Table 7 describes the results per method, summarizing the true positives (TPs), false positives (FPs), and false negatives (FNs). TP and FP quantify the number of correctly and wrongly identified code smell instances by the detection strategy. FN, on the other hand, quantifies the number of code smell instances that the detection strategy missed out. Additionally, the derived thresholds were applied for benchmarks 1, 2, and 3. For instance, by using the thresholds derived by Vale's method in benchmarks 1, 2, and 3, the number of TP, FP, and FN for god class candidates was 7, 1, and 0. The derived thresholds for both Lanza's and Vale's methods find the same values for FP. Differently of Vale's and Lanza's, Alves' method present different values of TP, FP, and FN for benchmark 1 and the other two benchmarks. For example, for benchmark 1, we found six TPs and for benchmarks 2 and 3, we found five TPs. Although some values of TP, FP, and FN are equal for different benchmarks and same method, the thresholds are different. For instance, for benchmarks 1 and 3, Lanza's method finds 126 and 130 to LOC, respectively (see Table 4).

Table 7 Identification of the selected code smells based on thresholds derived from each method

Code smell	No.	Vale's method			Lanza's method			Alves' method		
		Benchmarks			Benchmarks			Benchmarks		
		1	2	3	1	2	3	1	2	3
God class	TP	7	7	7	3	3	3	6	5	5
	FP	1	1	1	1	1	1	3	8	8
	FN	0	0	0	4	4	4	1	2	2
Lazy class	TP	9	9	9	1	1	1	n/a	n/a	n/a
	FP	0	0	0	0	0	0	n/a	n/a	n/a
	FN	1	1	1	9	9	9	n/a	n/a	n/a

Aiming to provide an additional perspective on the effectiveness on the identification of code smells, we also analyzed precision and recall measures. Recall (R) quantifies the rate of TP by the number of *existing code anomalies* (TP + FN). Precision (P) quantifies the rate of TP by the number of *detected code anomalies* (TP + FP). Precision and recall are represented by Eqs. 1 and 2, respectively:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2}$$

Table 8 presents recall (R) and precision (P) of the detection strategies applied to MobileMedia using the derived thresholds for the three methods. We can observe that for the detection strategy to identify god class instances, the precision and recall are 87.50 and 100% to Vale’s method and 75 and 42.86% in the case of Lanza’s method for the three benchmarks. In the case of Alves’ method, we found 66.70 and 85.70% for benchmark 1 and 38.50 and 71.40% for benchmarks 2 and 3. For the detection strategy that aims to identify lazy class instances, the values of precision are equals for Vale’s and Lanza’s methods (100%). Nevertheless, Vale’s method has higher values of recall (90%) when compared with Lanza’s method (10%). As we mentioned above, we did not compute precision and recall for Alves’ method in the lazy class evaluation.

The recall is considered more useful than precision in the context of identification of code smells because recall is a measure of completeness. That is, high recall means that the detection strategy was able to identify a high number of code smells in software. In other words, it is better to get more anomalous components (high recall) even with non-anomalous components also captured (low precision) than to get a small number of anomalous components (high precision) and miss many others (low recall). As general conclusions, Vale’s method fared better in the evaluation for both code smells.

7 Generalizability study

We also conducted a generalizability study aimed at applying Vale’s method to another benchmark type. This benchmark is composed of larger systems and by a larger set of systems. Hence, with this section, Vale’s method has derived thresholds from different program

Table 8 Recall and precision based on thresholds derived from each method

Code smell	No.	Vale’s method			Lanza’s method			Alves’ method		
		Benchmarks			Benchmarks			Benchmarks		
		1	2	3	1	2	3	1	2	3
God class	P	87.50	87.50	87.50	75.00	75.00	75.00	66.70	38.50	38.50
	R	100.00	100.00	100.00	42.86	42.86	42.86	85.70	71.40	71.40
Lazy class	P	100.00	100.00	100.00	100.00	100.00	100.00	n/a	n/a	n/a
	R	90.00	90.00	90.00	10.00	10.00	10.00	n/a	n/a	n/a

Table 9 Threshold value from Vale's, Lanza's, and Alves' methods

Method	Label	LOC	CBO	DIT	LCOM	NOC	RFC	WMC
Vale's Method	3%	3	0	1	0	0	1	1
	15%	11	2	1	0	0	3	2
	90%	308	24	4	66	1	58	42
	95%	510	33	5	186	2	85	70
Lanza's method	Low	0	0	1	0	0	0	0
	Mean	137	11	2	93	1	26	20
	High	490	24	3	2213	5	66	87
	Very high	735	36	5	3319	8	99	130
Alves' method	70%	565	29	2	90	0	84	75
	80%	901	39	2	240	0	119	123
	90%	1650	59	3	851	1	191	233

languages, domains, and sizes, making it more universal and comprehensive. The main goal of this section is to show that Vale's method is applicable for different benchmark types, as well as Lanza's and Alves' methods. In addition, this section supports discussions, such as the comparison of Sect. 8.6.

To conduct this generalizability study, we chose Qualitas Corpus¹ (Tempero et al. 2010), a well-known benchmark composed of industry-strength Java systems. JavaCC, JBoss, ArgoUML, and Hibernate are examples of systems that compose Qualitas Corpus. This benchmark has more than 100 systems, and most of these systems are larger and more complex than the product lines of the SPL benchmarks. In addition, the systems from Qualitas Corpus were developed in another programming language (Java, not AHEAD or FeatureHouse, like the SPL benchmarks). We used the release 20,101,126, composed of 106 Java open-source software systems. For each system, the corpus presents a set of 21 software metrics; 20 of them are numeric values. The corpus also provides a metric in nominal scale, namely inherited methods. In this study, we aim to derive thresholds for a subset of seven metrics, described in Sect. 3.1: CBO, DIT, LCOM, LOC, NOC, RFC, and WMC. From the 106 systems provided by Qualitas Corpus, only three of them do not have all these metrics computed; therefore, we excluded them.

Table 9 shows the obtained threshold values for Vale's, Lanza's, and Alves' methods. For example, for LOC, Vale's method derives 3, 11, 308, and 510 as thresholds to the labels 3% (very low), 5% (low), 90% (high), and 95% (very high). Lanza's method derives 0, 137, 490, and 735 for LOC to the labels low, mean, high, and very high, respectively. Alves' method for the same metric derives 565, 901, and 1650 for the labels 70% (moderate), 80% (high), and 90% (very high). In general, we observe that Vale's method provides lower thresholds when compared with Lanza's and Alves' methods. For instance, with respect to LCOM, the highest value is 186 (in the 95% label) for Vale's method, against 3319 and 851 for Lanza's (very high label) and Alves' (90%) methods. Despite existing a huge difference for the thresholds of some metrics (e.g., LCOM), we have similar thresholds derived from the three methods for other metrics (e.g., DIT and CBO). In these cases, the highest and lowest threshold values for each method are almost the same.

Furthermore, all threshold values provided by Vale's method and Alves' method are integer values, against decimal values for most of the provided threshold values by the Lanza's method. We converted these decimal values to integer values because, usually, a decimal

¹ www.qualitascorpus.com

value of the analyzed metrics does not make sense. Finally, Lanza's method may provide negative thresholds to metrics, given the use of standard deviation to derive thresholds. Considering the chosen software metrics for this study, a negative value did not make sense and required a special treatment. We converted these negative values to zero like we did for the SPL benchmarks. It is important to mention that this problem does not occur in Vale's and Aves' methods because all thresholds are derived based on values of metrics present in the benchmark. Therefore, in this case, both Vale's and Alves' methods are the ones which fit better to the aforementioned characteristic.

8 Discussion

This section presents discussions related to some choices of our work. In a previous work (Vale et al. 2015), we listed eight desirable points that methods to derive thresholds should follow: (i) the method should be systematic, (ii) derived thresholds should be presented in a step-wise format, (iii) it should be a weak dependence on the number of systems, (iv) it should be a strong dependence on the number entities, (v) it should not correlate to metrics, (vi) it should calculate upper and lower thresholds, (vii) it should provide representative thresholds independent of metric distribution, and (viii) it should provide tool support. Based on these desirable points, we listed seven requirements that Vale's method follows (Sect. 4). Those requirements and desirable points are explained in the following five sections. After, we discuss the derived thresholds obtained by Vale's, Lanza's, and Alves' methods for the different benchmarks reported in this study. Finally, we provide a theoretical analysis of the methods explored in this study with the other two methods presented in Sect. 2.5.

8.1 The method should be based on a benchmark

The first question that we want to discuss is why to use a benchmark-based method? Deriving thresholds from a benchmark data gives more confidence than deriving thresholds from a unique system. Doing an analogy, it is the same as asking for an expert the threshold of a metric against asking for many experts the same thing. The second option should give more confident results because we have more information and different opinions. This is the main reason Vale's method is based on benchmark data.

8.2 The method should have a strong dependence on the number of entities and a weak dependence on the number of systems

We want to know thresholds of metrics that are represented by a set of entities. Hence, the derived thresholds should be based on the number of entities and not by the number of systems (or SPLs). Nevertheless, the number of systems is also important, because using the same analogy at the last question, we want to know the answers from different experts and our benchmark should be composed by different systems developed by different experts. In addition, it is important to choose mature systems to increase the benchmark quality and indirectly increase the threshold quality. Although the number of systems can be considered important in terms of representativeness, we believe that the number of entities is more important. For this reason, Vale's method considers the number of entities explicitly and the number of systems is a consequence (implicitly) of the number of entities. We believe that if

we have a representative number of entities, we will have high probability that they are a representative number of systems.

Our SPL benchmarks have at least 14 SPLs and 2450 entities. We assume that this number of SPLs and entities are representative because there is few open-source code feature-oriented SPLs that use a compositional approach in the literature. Nevertheless, in another context, it can be a non-representative number of systems. For instance, for object-oriented systems, it is easy to find more than 100 systems; hence, there is no constraint regarding the number of systems. Anyway, this is a discussion of future work because more data is required to have consistent findings.

8.3 The method should calculate upper and lower thresholds in a step-wise format

Thresholds are often used to filter upper bound outliers. However, in some cases, it may make sense to identify lower bound outliers. For instance, the LOC, WMC, and CBO metrics used in the detection strategy of lazy class (see Fig. 5). For this reason, the proposed method derives upper and lower thresholds. In addition, the step-wise format can help separate outliers, for example, *very high* or *high* values. Of course, the highest threshold values, in the case of upper thresholds, have more chance to be a problem and these values should be treated firstly. However, we would like to discuss it with another perspective, in the case of detection strategies; a metric can be more important than other ones and this metric should be highlighted in the detection strategy. One way to do that is to evaluate such metric separately, but we trust that a good choice is to define *very high label* threshold for this metric instead of *high*. One example of that is the NCR metric in the detection strategy of god class presented in Sect. 6.3. The threshold used is high, but we believe that very high could fit better, and it would avoid some false positives.

Another point that we want to discuss here is the name of labels. Labels of the proposed method are very low, low, moderate, high, and very high. We believe that a very low label can be bad and a very high label can also be bad. It depends on the context, and it does not mean that a very low label is always good or always bad. For this reason, we preferred not to call the labels relating to risks or bad design.

8.4 The method should respect the statistical properties of metrics

Thresholds are derived to find outliers in a system; if the statistical properties of metrics are changed, the derived thresholds probably find wrong outliers. Hence, we believe that a good method should analyze the metric distributions without changing anything in their statistical properties. It implies in not correlating metrics or weighting entities differently (all entities of all systems should have the same importance). Therefore, the analysis of the metric distributions can be done by viewing the metrics distribution in different ways. For example, an alternative way to examine a distribution of values is to plot the probability density function (PDF) for each analyzed metric.

Figure 6 depicts the distribution of the CBO, WMC, and NCR values for SPL benchmark 3, using a PDF. The x-axis represents the metric values; they range from 0 to 66 to CBO, from 0 to 383 to WMC, and from 0 to 9 to NCR. The y-axis represents the percentage of observations (percentage of entities). The use of the PDF is justifiable because we want to determine thresholds (the dependent variables, the metric values in this case) as a function of the percentage of observations (independent variable). Furthermore, by using the percentage of

observations instead of the frequency, the scale becomes independent of the size of the benchmark, making it possible to compare different distributions. For instance, we can observe in Fig. 6a that 68.35% of entities have $CBO \leq 5$. Similarly, in Fig. 6b and supported by Table 3, we can observe that 90% of entities have $WMC < 19$. However, after these two points, the measures come to increase quickly. For example, 95% of entities have $WMC < 35$. Looking at the first time, the labels high (90%) and very high (95%) of Vale’s method look too rigid, but as can be seen for WMC, CBO, and NCR, it is not.

On the other hand, if we listed low percentages of CBO values for SPL benchmark 3, the values do not have a big difference. For instance, to 1, 3, 5, 10, 15, and 20%, the values are 1, 2, 2, 2, 2, and 3, respectively. Similarly, it happens with WMC values of SPL benchmark 3, the values for such percentages are 1, 1, 1, 2, 2, and 2. We have a variation of two units in the first case and one unit in the second case. This happens because following the distribution of Weibull, these metrics are close to having or it has a *heavy-tailed distribution* (with *shape parameter* equals to 1.2244 and 0.72041, respectively, to CBO and WMC). Analyzing these data, we see that very low label should be stronger than a very high and distant to a low label. We considered 1% very rigid; hence, we chose 3% for the very low label. On the other hand, we chose 15% for the low label to be a greater percentage difference.

Another point that we want to highlight here is that Lanza’s method assumes that metrics follow a normal distribution. As the examples of CBO and WMC showed, it is not always true. The metric distributions do not follow a normal distribution impact in the low labels derived from Lanza’s method (see Table 5) for all metrics and benchmarks where the values were 0. This happened because the standard deviation is higher than the mean (low label is calculated by mean minus standard deviation), and it does not make sense that the set of metrics chosen have negative values. Additionally, several studies show that different software metrics follow heavy-tailed distribution (Concas et al. 2007; Louridas et al. 2008). Hence, if a metric does not follow a heavy-tailed distribution, are the derived thresholds from Vale’s method valid? The proposed method takes into account the metric distributions focused onto identifying outliers. Hence, if the metric follows a normal distribution, heavy-tailed, or a common distribution, for example, the outliers are identified using the proposed labels.

As a concrete example, Fig. 7 presents the DIT (depth of inheritance tree) metric from the benchmark of Ferreira et al. (2012). This metric has a common distribution and the common value is 1. Probably, with Vale’s method, the high and very high labels for this metric would be 2 or higher; that is, a value above the common (above 1). The low value would be 0 or 1; it is a

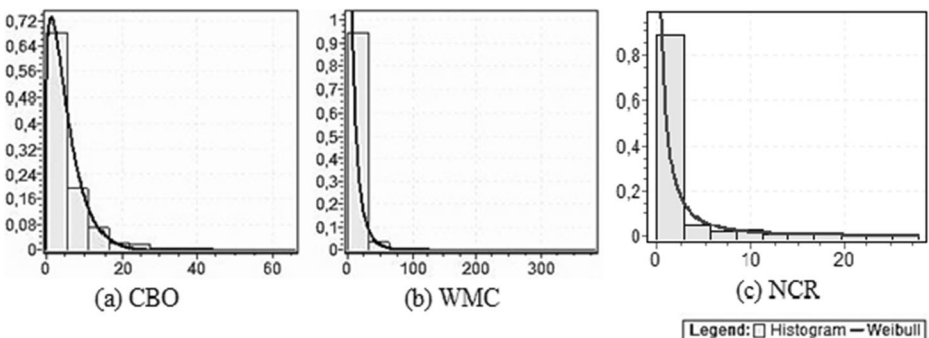


Fig. 6 Probably density function

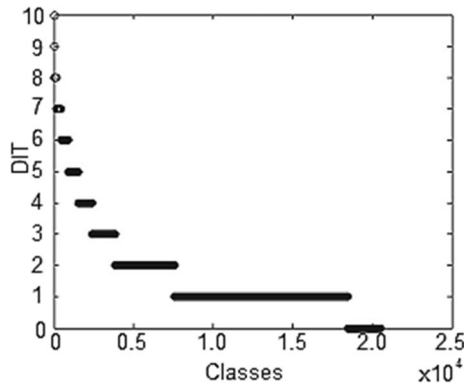


Fig. 7 Probably density function (Ferreira et al. 2012)

value below or equal to the common. In other words, we are only identifying discrepant values (outliers) related to the provided data (benchmark).

For the reasons explained in this section, the proposed method uses the percentages 3, 15, 90, and 95 to represent the following labels: very low (0–3%), low (3–15%), moderate (15–90%), high (90–95%), and very high (>95%). Hence, we do not change anything in the metric distributions, and we believe that Vale’s method is a good way to derive thresholds with different metric distributions.

8.5 The method should be systematic, repeatable, transparent, and straightforward

These points are important characteristics for any method that derives thresholds. Vale’s method is considered systematic because it has five well-defined steps. If the steps are followed, the same results should be obtained. Although we give autonomy for the user to define the percentages of each label, the same percentages should be defined to get the same results. Hence, the method is repeatable and transparent. As explained above, we do not change any statistical properties of the target metric to derive thresholds and all entities are equally weighted. Therefore, this method is straightforward. In addition, we developed a tool to support the steps of the proposed method (Sect. 4). The tool helps the user to derive thresholds, and this task comes to be easier (less time-consuming and less error-prone).

8.6 Discussing the derived thresholds

In this study, we evaluate Vale’s method using three different SPL benchmarks. We also apply Vale’s method to one Java-based benchmark to show that our method is as applicable for different benchmark types as Lanza’s and Alves’ methods are. Figures 8 and 9 present the derived thresholds side by side for the four benchmarks using the three subject methods in this study. These figures focus on the high and very high labels for the metrics LOC, CBO, and WMC. Benchmarks 1, 2, and 3 correspond to the three SPL benchmarks presented in Sect. 5.1 while benchmark 4 corresponds to the Java-based benchmark presented in Sect. 7.

When the thresholds of these four benchmarks are compared, we can see that benchmarks composed of larger software systems present higher thresholds. Hence, as the size of the Java-based benchmark systems is much larger than the SPL benchmark systems, the derived thresholds are also larger. For instance, considering the label high for LOC metric of benchmarks

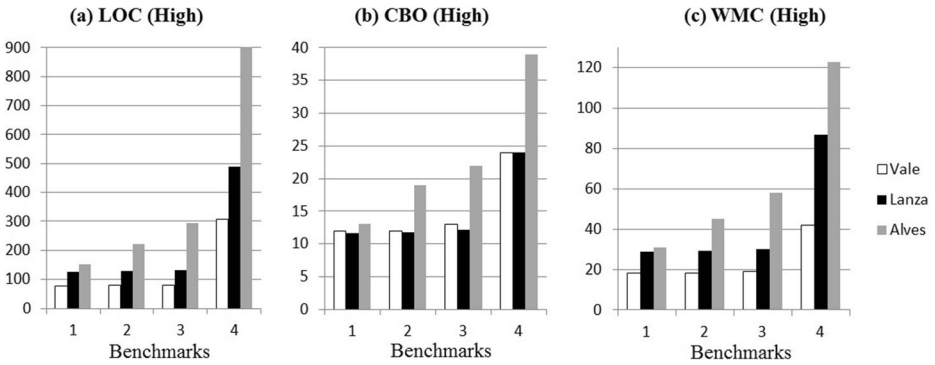


Fig. 8 Metric thresholds side by side for high label

1 and 4, we have a difference of 231, 364, and 750 for Vale’s, Lanza’s, and Alves’ methods, respectively. As another example, considering the label very high for the same metric and benchmarks, we have a difference around 372, 546, and 1398, for Vale’s, Lanza’s, and Alves’ methods, respectively. Therefore, we conclude that the derived thresholds are directly influenced by the benchmark and, as consequence, low benchmark quality corresponds to low threshold quality. Low benchmark quality may mean a benchmark composed of non-similar systems.

8.7 Theoretical analysis of benchmark-based methods

In the previous sections, we discussed desirable points of methods to derive metric thresholds. As described in Sect. 2.5, we consider Vale’s method more related to three methods (Ferreira’s, Oliveira’s, and Alves’ methods). And based on the reasons described in Sect. 6, we decided to compare the thresholds derived using Vale’s method with the thresholds derived by Lanza’s and Alves’ methods. Therefore, this section aims to compare these five benchmark-based methods theoretically by relying on the key and desirable points described in this study.

Hence, Tables 10 highlight the main differences of Lanza’s, Ferreira’s, Oliveira’s, Alves’, and Vale’s methods. Lanza’s and Oliveira’s methods are deterministic because these methods present a set of formulas and the results are always the same. On the other hand, Vale’s and Alves’ methods recommend percentages to derive thresholds, but the users must decide if they use the recommended percentages or not and because of that, they are considered partially

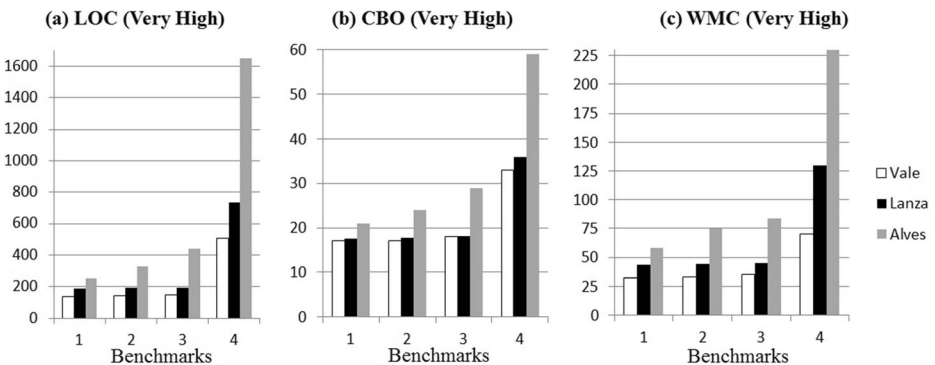


Fig. 9 Metric thresholds side by side for very high label

Table 10 Comparative evaluation of the method for calculating thresholds

Question	Method				
	Lanza	Ferreira	Oliveira	Alves	Vale
Is it deterministic?	Yes	No	Yes	Partially	Partially
Are step-wise outliers identified?	Yes	Yes	No	Yes	Yes
Is the metric distribution considered?	No	Yes	Yes	Yes	Yes
Does the number of systems impact?	Weak	Weak	Strong	Strong	Weak
Does the number of entities impact?	Strong	Strong	Weak	Weak	Strong
Does it correlate with other metrics?	No	No	No	Yes	No
Lower bound thresholds?	Yes	No	No	No	Yes
Provides tool support?	No	No	Yes	No	Yes

deterministic. As Ferreira's method does not explicitly describe how the groups of thresholds are extracted, we classified it as non-deterministic.

Related to derive step-wise thresholds, only Oliveira's method does not do that. Related to the metric distribution, Lanza's method uses just mean and standard deviation to derive thresholds and, hence, this method is classified as a method that does not consider the metric distribution. Related to the impact of the number of systems and entities, Vale's, Lanza's, and Ferreira's methods consider the number of entities more important than the number of systems, differently of Oliveira's and Alves' methods. Alves' method is the one that correlates metrics. Lanza's and Vale's methods are the ones that calculate lower bound thresholds. Both Oliveira's and Vale's method provide free and open-source tool support.

9 Threats to validity

Even with the careful planning, this research can be affected by different factors which, while extraneous to the concerns of the research, can invalidate its main findings. Actions to mitigate their impact on the research results are described, as follows.

- **SPL repository** — we followed a careful set of procedures to create the SPL repository and build the benchmarks. As the number of open-source SPLs found is limited, we could not have a repository with a higher number of SPLs. This limitation has implication in the number of analyzed components, which is particularly relevant to the NCR metric. This factor can influence the derived thresholds as the number of components for NCR analysis is further reduced. Therefore, in order to mitigate this limitation, we created different benchmarks for comparison of the derived thresholds.
- **Measurement process** — the SPL measurement process in our study was automated based on the use of existing tooling support, named VSD (Abilio et al. 2014), to measure AHEAD code. However, there was no existing tool defined to explicitly collect metrics in FeatureHouse (FH) code. Therefore, the SPLs developed with this technology had to be transformed into AHEAD code. This transformation was made by changing the composer of FH to the composer of AHEAD. There are reports in the literature showing that this transformation preserves all properties of FH (Apel et al. 2009). We also reduced possible

threats by performing some tests with few SPLs. In fact, we observed all software proprieties were preserved after the transformation.

- Metric labels — we define the labels very low (0–3%), low (3–15%), moderate (15–90%), high (90–95%), and very high (> 95%), although the chosen percentages cannot be the best. To try generalizing and providing default labels, we decide to use these percentages. In addition, it can be seen that very low and low labels should be equal or similar values, in spite of that, we prefer to keep both labels and increase their difference in terms of percentages. High and very high labels have a small difference in terms of percentage than very low and low labels. This small difference (5%) was chosen because at the end (tail), the difference of the values is greater. In other words, these values (percentages) were defined based in our experience analyzing some metric distributions, although if someone thinks that these values do not fit well in their metric distribution, other values can be used.
- Code smells — we discuss only two types of code smells (i.e., god classes and lazy classes). Fowler has cataloged a list with more than 20 code smells (Fowler 1999). Therefore, these smells used to evaluate the effectiveness of both methods (Lanza's and Vale's method) may not necessarily be a representative sample of code smells found in certain SPL. In addition, we have to adapt the lazy class detection strategy changing the absolute values to a low label of the target metric. It can be affected by the evaluation, but this choice affected all subject methods. Hence, we assume that we were fair.
- Tooling support and scoping — the computation of metric values and metric thresholds can be affected by the tooling support and by scope. Different tools implement different variations of the same metrics. To overcome this problem, the same tool (i.e., VSD) was used both to derive thresholds and to analyze systems. The tool configuration with respect to which files to include in the analysis (scoping) also influences the computed thresholds. For instance, the existence of test code, which contains very little complexity, may result in lower threshold values. On the other hand, the existence of generated code, which normally has high complexity, may result in higher threshold values. As previously stated, for deriving thresholds, we removed manually all supplementary code from our analysis. Regarding the Qualitas Corpus, we already got the measures from their website.
- Reference list generation — a reference list for each code smell had to be defined to calculate recall and precision measures. Several precautions were taken. Despite that, we can have omitted some code smell instances or chosen a code smell instance that does not represent a design problem. In order to mitigate this threat, we rely on experts of the target application in order to validate the final reference list.
- Tool evaluation — we compared the results calculated manually with the thresholds calculated by TDTool (automatically). If some mistake occurred in both cases (manually and automatically), we have presented wrong thresholds. To minimize it, we manually calculate the thresholds for three SPL benchmarks. Therefore, if some mistake happened, it happened in the three cases. We think that it is more improbable than if we had calculated the thresholds for only one benchmark.
- Qualitas Corpus — we also chose a well-known open-source system benchmark, called Qualitas Corpus, to apply all three threshold derivation methods, namely Vale's, Lanza's, and Alves' methods. Considering that some metrics used in this study were not available for 3 of the 106 systems that compose the selected corpus, we decided to discard these 3 systems from our study. These systems (Eclipse, JRE, and NetBeans) are larger and more complex than others from the corpus, and this discard can affect the threshold derivation.

Although this discard is a threat, it does not invalidate the findings of our study because we just reduced our system corpus; 103 systems still composed this benchmark.

10 Conclusion and future work

This paper describes the importance of software metrics, the use of a set of metrics to measure quality attributes, and the calculation of representative thresholds. In order to focus on the last point, we proposed a method to derive thresholds. The proposed method, called Vale's method, tries to get the best of previously proposed methods and follows an evolution of these kinds of methods that we saw in a literature review. In addition, we demonstrated that Vale's method provides threshold values that can be used in different contexts, such as the SIG quality model and software anomaly detection.

We evaluated Vale's method comparing it with Lanza's and Alves' methods, two other methods with the same purpose. The evaluation was conducted by deriving thresholds using three SPL benchmarks for the three subject methods and analyzed the effectiveness of the thresholds derived by these three methods in detecting software anomalies. As a complementary analysis, we derived thresholds for the same three methods but using another benchmark of single systems and different characteristics. The three SPL benchmarks are based on one repository we created containing SPLs developed using AHEAD and FeatureHouse. Our three SPL benchmarks are composed of 33 (benchmark 1), 22 (benchmark 2), and 14 (benchmark 3) SPLs. The fourth benchmark used in this paper was found in the literature and it is composed of 103 Java systems. The main differences from the SPL benchmarks to the fourth one are (i) the number of systems that compose the benchmark, (ii) the size of the systems, and (iii) the programming language.

As results from the evaluation, we could see that Vale's method fared better than did Lanza's and Alves' methods in anomaly detection. Vale's method achieved 87.5% and 100% of precision and recall for god classes while Lanza's method achieved 75% and 42.86% and Alves' method scored 66.70% and 85.70% in the best case for precision and recall, respectively. Regarding the lazy class detection strategy, Vale's method obtained 100% and 90% of precision and recall while Lanza's method obtained the same precision, but just 10% of recall. As results of the complementary analysis (Sect. 7), we concluded that Vale's method provides more balanced (more realistic probably) and lower thresholds than the thresholds of Lanza's and Alves' methods. As Lanza's method is based on mean and standard deviation, it presents decimal and negative values. Hence, as it does not make sense for the metrics we used in the study, we converted the decimal values for integers and the negative values to zero.

With the results of the evaluation, generalizability study, and lessons learned of this study, we provided discussions on different topics, such as the justification of the steps of Vale's method and why a method should be benchmark-based, respect metrics' statistical properties, and have a strong dependence on the number of entities. The topic related to statistical properties (Sect. 8.4) helps us to justify many decisions we made in this study, such as the percentages chosen and why we trust that Vale's method can be applied on metrics with different metric distributions. In addition, the generalizability study shows that Vale's method can be applied to different systems and benchmarks and it also supported a discussion comparing the threshold values derived by all subject benchmarks presented in this study.

After all comparisons and analyses, we showed that Vale's method fits the eight desirable points: (i) it is based on data analysis from a representative set of systems (benchmark); (ii) it has

a strong dependence on the number of entities; (iii) it has a weak dependence on the number of systems; (iv) it calculates upper and lower thresholds; (v) it derives thresholds in a step-wise format; (vi) it respects the statistical properties of the metric; (vii) it is systematic, repeatable, transparent, and straightforward to execute; and (viii) it provides tool support. Therefore, we concluded that the provided thresholds, using Vale's method, are considerably different between benchmarks. It implies that thresholds should not be universal but dependent on a benchmark. Consequently, the quality of the benchmark impacts on the quality of the thresholds.

As future work, we intend to explore how to build representative benchmarks. We plan to do that because, as we saw in this paper, the derived thresholds are influenced by the benchmark and we did not find any study showing which factors may influence the derived thresholds most. Such a study can help software engineers to circumvent their limitations, such as the number of systems to compose their benchmark. We also intend to provide an extension of our tool to run other benchmark-based methods. Fulfilling that, we can make the threshold calculation easier, and software engineers can choose the method that better fits their demands for a specific context.

Acknowledgments This work was partially supported by CAPES, CNPq (grant 424340/2016-0 and 290136/2015-6), and FAPEMIG (grant PPM-00651-17).

References

- Abilio, R., Vale, G., Oliveira, J., Figueiredo, E., Costa, H. (2014). Code smell detection tool for compositional-based software product lines. In *Proceedings of the 5th Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session* (pp. 109–116).
- Abilio, R., Padilha, J., Figueiredo, E., Costa, H. (2015). Detecting code smells in software product lines: An exploratory study. In *Proceedings of the 12th International Conference on Information Technology: New Generations (ITNG)* (pp. 433–438).
- Abilio, R., Vale, G., Figueiredo, E., Costa, H. (2016). Metrics for feature-oriented programming. In *Proceedings of 7th International Workshop on Emerging Trends in Software Metrics (WETSoM)* (pp. 36–42).
- Alves, T., Ypma, C., Visser, J. (2010). Deriving metric thresholds from benchmark data. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM)* (pp. 1–10).
- Apel, S., Kästner, C., Lengauer, C. (2009). FeatureHouse: language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)* (pp. 221–231).
- Batory, D., & O'Malley, S. (1992). The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering Methodology*, 1(4), 335–398.
- Brereton, P., Kitchenham, B., Budgen, D., Turner, M., & Khalil, M. (2007). Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software (JSS)*, 80(4), 571–583.
- Chidamber, S., & Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 476–493.
- Coleman, D., Lowther, B., & Oman, P. (1995). The application of software maintainability models in industrial software systems. *Journal of Systems and Software*, 29(1), 3–16.
- Concas, G., Marchesi, M., Pinna, S., & Serra, N. (2007). Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10), 687–708.
- Dumke, R., & Winkler, A. (1997). Managing the component-based software engineering with metrics. In *Proceedings of the 5th International Symposium on Assessment of Software Tools and Technologies (SAST)* (pp. 104–110).
- Emi, K., & Lewerentz, C. (1996). Applying design-metrics to object-oriented frameworks. In *Proceedings of the 3rd International Symposium on Software Metrics (METRICS)* (pp. 64–72).
- FeatureIDE. (2017). https://urldefense.proofpoint.com/v2/url?u=https-3A_featureide.github.io_&d=DwIDaQ&c=vh6FgFnduejNhPPD0fl_yRaSfZy8CWbWnIf4XJhSqx8&r=ao7Vv0uBvR-wgd0

- ykVbHjMjeV7vz8HzQ1TmA0JY0voNuKuAvWNpcltsSHnxfZLM&m=NXtTfn_yFTngAOYPIXUR6CiVMbeellwiOxvPQCx-ywc&s=7fd1VjMY-X8pVI_KRWfYU0ymJlMEQ-rXjpSLxafCo3A&e=.
- Fenton, N. (1991). *Software metrics: a rigorous Approach* (pp. 28–37). London: Chapman-Hall.
- Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. In Proceedings of the 20th International Conference on Evaluation and assessment in software engineering (EASE). Limerick, 1–3 June 2016.
- Fernandes, E., Vale, G., Sousa, L., Figueiredo, E., Garcia, A., Lee, J. (2017). No code anomaly is an island: anomaly agglomeration as sign of product line instabilities. In *Proceedings of the 16th International Conference on Software Reuse (ICSR)*, pp. 48–64.
- Ferreira, K., Bigonha, M., Bigonha, R., Mendes, L., & Almeida, H. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2), 244–257.
- Ferreira, G., Gaia, F., Figueiredo, E., & Maia, M. (2014). On the use of feature-oriented programming for evolving software product lines: a comparative study. *Science Computer Programming*, 93(1), 65–85.
- Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro M, Kulesza U, Garcia A, Soares S, Ferrari F, Khan S, Castor Filho F, Dantas F (2008) Evolving software product lines with aspects: an empirical study on design stability. In: *Proceeding of the 30th International Conference on Software Engineering (ICSE)* (pp. 261–270). Leipzig: IEEE Computer Society.
- Fowler, M. (1999). *Refactoring: improving the design of existing code*. Reading: Addison Wesley.
- French, V. (1999). Establishing software metric thresholds. In *Proceedings of the 4th International Workshop on Software Measurement (IWSM)*.
- Gamma, G., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Reading: Addison-Wesley.
- Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC)* (pp. 30–39).
- Herbold, S., Grabowski, J., & Waack, S. (2011). Calculation and optimization of thresholds for sets of software metrics. *Empirical Software Engineering*, 16(6), 812–841.
- Kitchenham, B., & Charters, S. (2007). *Guidelines for performing systematic literature reviews in software engineering*. EBSE Technical Report, Keele University.
- LabSoft (2017). http://labsoft.dcc.ufmg.br/doku.php?id=%20about:spl_list.
- Lanza, M., & Marinescu, R. (2006). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Berlin Heidelberg: Springer-Verlag.
- Lima, E., Resende, A., & Lethbridge, T. (2016). The uncomfortable discrepancies of software metric thresholds and reference values in literature. In *Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA)* (pp. 1–9).
- Lorenz, M., & Kidd, J. (1994). *Object-oriented software metrics*. New York: Englewood Cliffs.
- Louridas, P., Spinellis, D., & Vlachos, V. (2008). Power laws in software. *ACM Transactions on Software Engineering Methodology*, 18(1), 1–26.
- Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th International Conference on Software Maintainability (ICSM)* (pp. 350–359).
- McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4), 308–320.
- Mori, A., Vale, G., Vigiato, M., Oliveira, J., Figueiredo, E., Cirilo, E., Jamshidi, P., Kastner, C. (2018) *Evaluating domain-specific metric thresholds: an empirical study*. International Conference on Technical Debt (TechDebt).
- Munro, M. (2005). Product metrics for automatic identification of “bad smell” design problems in java source-code. In *Proceeding of the 11th international software METRICS symposium (METRICS)* (pp. 1–9).
- Nejmeh, B. (1988). NPATH: A measure of execution path complexity and its applications. *Communications of the ACM*, 31(2), 188–200.
- Oliveira, P., Valente, M., Lima, F. (2014). Extracting relative thresholds for source code metrics. In *Proceedings of the 18th International Conference on Software Maintenance and Reengineering (CSMR)* (pp. 254–263).
- Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A., Sant'Anna, C.. (2014) On the effectiveness of concern metrics to detect code smells: an empirical study. In *Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAISE)*.
- Perkusich, M., Medeiros, A., Silva, L., Gorgônio, K., Almeida, H., Perkusich, A. (2015). A Bayesian network approach to assist on the interpretation of software metrics. In *Proceedings of the 30th Symposium on Applied Computing (SAC)* (pp. 1498–1503).
- Riel, J. (1996). *Object-oriented design heuristics*. Boston: Addison-Wesley.
- Schulze, S., Apel, S., Kastner, C. (2010). Code clones in feature-oriented software product lines. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)* (pp. 103–112).

- Software Engineering Institute – SEI (2016). <http://www.sei.cmu.edu/productlines/>
- Spinellis, D. (2008). A tale of four kernels. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)* (pp. 381–390).
- SPL2GO (2015). <http://spl2go.cs.ovgu.de>.
- Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J. (2010). Qualitas Corpus: A curated collection of Java code for empirical studies. In *Proceedings of 17th the Asia-Pacific Software Engineering Conference (APSEC)* (pp. 336–345).
- Vale, G., & Figueiredo, E. (2015). A method to derive metric thresholds for software product lines. In *Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES)* (pp. 110–119).
- Vale, G., Albuquerque, D., Figueiredo, E., Garcia, A. (2015). Defining metric thresholds for software product lines: a comparative study. In *Proceedings of the 19th International Software Product Line Conference (SPLC)* (pp. 176–185).
- Vasa, R., Lumpe, M., Branch, P., Nierstrasz, O. (2009). Comparative analysis of evolving software systems using the Gini coefficient. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM)* (pp. 179–188).
- Veado, L., Vale, G., Fernandes, E., Figueiredo, E. (2016). TDTool: threshold derivation tool. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, Tools Session (Article No. 24).



Gustavo do Vale is bacharel of Information Systems by Federal University of Lavras (UFLA) and Master in Computer Science by Federal University of Minas Gerais (UFMG). He is a member at team PqES and LabSoft from UFLA and UFMG, respectively. Furthermore, he is currently a PhD student of Computer Science in the department of computing at University of Passau.



Eduardo Fernandes holds a degree in Computer Science from Federal University of Mato Grosso do Sul (2014) and a Master's degree in Computer Science from Federal University of Minas Gerais (2017). He is

currently a PhD student in Informatics at Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He is also a researcher in Software Engineering for Opus Research Group at PUCRio. His research interests include refactoring, software metrics, empirical software engineering, and software reuse.



Eduardo Figueiredo is an assistant professor and head of the Software Engineering Laboratory (LabSoft) at the Federal University of Minas Gerais (UFMG) since 2010. He received his PhD degree in Software Engineering from Lancaster University (UK) in 2009 and also holds MSc degree in Software Engineering from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. His research interests include aspect-oriented programming, software product lines, empirical software engineering, and software metrics. Website: <http://www.dcc.ufmg.br/~figueiredo>.