A Method to Derive Metric Thresholds for Software Product Lines

Gustavo Vale
Software Engineering Laboratory (Labsoft)
Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
gustavovale@dcc.ufmg.br

Eduardo Figueiredo
Software Engineering Laboratory (Labsoft)
Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
figueiredo@dcc.ufmg.br

***Abstract*—A software product line (SPL) is a set of software systems that share a common and variable set of components (features). Software metrics provide basic means to quantify several quality aspects of SPL components. However, the effectiveness of the SPL measurement process is directly dependent on the definition of reliable thresholds. If thresholds are not properly defined, it is difficult to actually know whether a given metric value indicates a potential problem in the component implementation. There are several methods to derive thresholds for software metrics. However, there is little understanding about their appropriateness for the context of SPLs. This paper aims to propose a method to derive thresholds in the SPL context. Our method is evaluated in terms of recall and precision using two code smells (God Class and Lazy Class) detection strategies. The evaluation of our method is performed based on a benchmark of 33 SPLs and the results were compared with a method (baseline) with the same purpose used in the context of SPLs (not proposed). The results show that our method has better recall when compared with baseline.**

***Index Terms*— Metrics; Thresholds; Software Product Lines**

I. Introduction

With software-intensive systems growing in size and complexity, better support is required for measuring and controlling the software quality [18]. These tasks become harder in software projects composed of several varying features, such as Software Product Lines (SPLs) [22]. SPLs are been increasingly adopted in software industry to support coarse-grained reuse of software assets [10]. Each SPL is a configurable set of systems that shares a common, managed set of features in a particular market segment [32]. Features can be defined as modules of an application with consistent, well-defined, independent, and combinable functions [4].

Software metrics are the pragmatic means for assessing different quality aspects, such as maintainability and changeability [7][23]. Certain metric values can help to reveal specific components (or modules) of a software system that should be closely monitored [10]. For instance, such measures can be used to indicate whether a critical anomaly (or smell) is affecting a component structure. Then, developers may suspect that something is wrong in the component implementation. Typical examples include cases of large classes indicating the enclosing module may be realizing non-cohesive characteristics of the system [1].

Nevertheless, the effective measurement of software systems is directly dependent on the definition of appropriate thresholds and, for SPLs, this is not different. Thresholds allow to objectively characterize or to classify each component according to one of the quality metrics. The definition of appropriate thresholds needs to be tailored to each metric. In the past few years, thresholds were calculated by software engineers experience and/or using a single system as reference [7][8][11] [17][26][28][33][37]. Recently, this concept has been changing and thresholds have been calculated considering three points [3][13][29]: (i) well-defined methods, (ii) methods that consider the skewed distribution of software measurements, and, (iii) derived from benchmarks.

We performed a comparison of methods that address these three points in a previous work [36]. As result of this comparison eight desirable points in methods to derive thresholds were obtained [36]. This paper aims to propose a method to derive thresholds that considers the desirable points proposed in a previous work [36]. To evaluate the use of the method proposed in this paper, we build a benchmark composed by 33 feature-oriented SPLs derive the thresholds for this method. We apply the derived thresholds in two detection strategies (God Class and Lazy Class) found in the literature to evaluate the recall and precision of the proposed method. To be a baseline, we compare our results with the results of a method with the same purpose. This was the unique method applied in the context of SPLs to derive thresholds found in the literature, although, it was proposed in another context.

As results, the proposed method fared better in the evaluation in terms of recall. We illustrate and justify each step of the proposed method. Furthermore, it can be said that the proposed method fits the following requirements: (i) it is benchmark-based; (ii) it has strong dependence with the number of entities; (iii) it has a weak dependence with the number of systems; (iv) it calculate upper and lower thresholds; (v) it derives thresholds in a step-wise format; (vi) it respects the statistical properties of metrics; (vii) it is systematic, repeatable, transparent and straightforward to execute.

The remainder of this paper is organized as follows. Section II presents some related work. Section III describes the proposed method to derive thresholds. Section IV presents an example of use of our method. Section V describes the evaluation of the proposed method. Section VI presents a discussion about the choices and decisions to create our method. Section VII discusses threats to validity this study. Section VIII concludes this paper and presents suggestions for future work.

II. Background and Related Work

In this section, we review previous attempts to define metric thresholds. For this, we performed an *ad-hoc* literature review to find methods to derive thresholds. The literature review was held in four different electronic databases: IEEExplore[1], Science Direct[2], ACM Digital Library[3], and El Compendex[4]. In addition to examining the items retrieved in these electronic databases, we used the s*nowballing* technique [6]. This technique consists in investigating the references retrieved in electronic databases in order to find additional relevant papers to increase the scope of the search, providing broader results [6].

The literature review of this work followed similar steps to the protocol of a Systematic Literature Review (SLR) [20]. A SLR is a well-defined method to identify, evaluate, and interpret all relevant studies regarding a particular research question, topic area, and phenomenon of interest [20]. After applying the review protocol, we selected 50 primary studies in order to extract information related to methods to derive thresholds. However our literature aimed to find methods to derive thresholds. The primary studies can be found on the project website [35]. Some of these primary studies are discussed in this section.

First, we present the metrics used in this study. To summarize our review, we start by describing work where thresholds are defined by programming experience. Then, we analyze in detail methods that derive thresholds based on data analysis, which are directly related to our research. Finally, we discuss techniques to analyze and summarize metric distributions and methods.

A. Software Metrics

In this study, we use the four following metrics:

Lines of Code (LOC) [23] counts the number of uncommented lines of code per class. The value of this metric indicates the size of a class. *Coupling between Objects (CBO)* **[7]** counts the number of classes called by a given class. CBO measures the degree of coupling among classes. *Weight Method per Class (WMC)* [7] counts the number of methods in a class. This metric can be used to estimate the complexity of a class. *Number of Constant Refinements (NCR)* [1] counts the number of refinements that a constant has. Its value 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) [5]. That is, refinements can change the behavior of a constant (such as, a class) if certain feature is included in a product. These metrics capture different attributes of a SPL implementation, i.e. size, coupling, complexity, and refinement. These metrics were chosen because of the Metric-Based Detection Strategies used in this paper (explained in Section IV.A).

B. Thresholds derived from programming experience

Many authors defined metric thresholds according to their programming experience. For example, the values 10 and 200 were defined as thresholds for *McCabe* [26] and *NPATH* [28], 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 [8]. When MI values are higher than 85, between 85 and 65, and are smaller than 65 they are considered as highly-maintainable, moderately-maintainable, and difficult to maintain, respectively. These thresholds rely on programming experience and it is difficult to reproduce or generalize these results. Additionally, the lack of scientific support can lead to dispute about the values. However, unlike the related work cited above [8][26][28], our research aims to propose a method to derive thresholds for SPLs not based on programming experience.

C. Thresholds derived from metric analysis

Erni et al. [11] propose the use of mean (μ) and standard deviation (σ) to derive a threshold (T) from project 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 [21] 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 at the same way as Erni. Labels very high is calculated as $T = (\mu + \sigma) \times 1.5$. Abilio et al. [1] use the same method than Lanza and Marinescu, but they derive thresholds based on eight SPLs. These methods are a common statistical technique. However, Erni et al. [11], Abilio et al. [1], and Lanza and Marinescu [21] do not analyze the underlying distribution of metrics. The problem with these methods is that they assume that metrics that are normally distributed, limiting the use of these methods. In contrast, our research focuses on a method that does not make assumption about data normality.

French [17] also proposes a method based on the mean (μ) and standard deviation (σ). 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 large numbers of outliers. For metrics with high range or high variation, this method identifies a smaller percentage of observations than its theoretical maximum. In contrast, our method was designed to derive thresholds from benchmark data (of SPLs) and, as such as, it is resilient to high variation of outliers.

D. Methodologies for characterizing metric distributions

Chidamber and Kemerer [7] use histograms to characterize and analyze data. For each of their 6 metrics (i.e. WMC and CBO), they plotted histograms per programming language to discuss metric distribution and spot outliers in C++ and Smalltalk systems. Spinellis [33] compares metrics of four operating

[1] ieeexplore.ieee.org/
[2] www.sciencedirect.com
[3] www.acm.org/
[4] www.engineeringvillage.com

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. [37] 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 [7], Spinellis [33], and Vasa [37], we did not use histograms, mean, median or Gini coefficient to calculate thresholds and we derive thresholds based in data from a benchmark.

E. Methods to derive thresholds

This section describes methods closer to our, because they are transparent, the thresholds are extracted from benchmark data, and the methods consider the skewed distribution of metrics. Alves et al. [3] proposed a method that weight 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 fix and pre-determined percentage of entities. Generally, this method proposes 70%, 80%, or 90% to represent the labels: *low* (between 0-70%), *moderate* (70-80%), *high* (80-90%), and *very high* (>90%). Similarly, Ferreira et al. [13] presented a simple method for calculating thresholds. The method consists in grouping the extracted metrics in a file and gets three boxplots, with high, medium, and low frequency. The boxplots are called good, regular, and bad measurements, respectively. In the Ferreira's method description, it is not clear how to extract the three boxplots. Oliveira et al. [29] proposed a more systematic method. 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 [29]. 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.

Several studies [3][9][13][24][29] clearly demonstrate that most software metrics do not follow *normal distributions*, limiting the use of any statistical method that uses mean to derive thresholds, for example. Thus, the studies fall short in concluding how to use these distributions, and their coefficients, to establish baseline values to measuring and controlling the software quality. Moreover, even if such baseline values were established it would not be possible to identify the code responsible for deviations, since there is no traceability of results. Additionally, several studies [9][24] show that different software metrics follow *heavy-tailed distribution*. Concas et al. [9] show that for a large Smalltalk system most of the Chidamber and Kemerer's metrics [7] follow heavy-tailed distribution. Louridas et al. [24] show that different software metrics also follow heavy-tailed distribution.

III. The Proposed Method to Derive Thresholds

The method proposed in this section was designed according to the following requirements: (i) it should be based on data analysis from a representative set of systems (benchmark); (ii) it should be a strong dependence with the number of entities; (iii) it should be a weak dependence with the number of system; (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; (vii) it should be systematic, repeatable, transparent and straightforward to execute. These requirements are mainly based on the desirable points described by in our previous work [36].

Figure 1 summarizes the 5 steps of the proposed method.

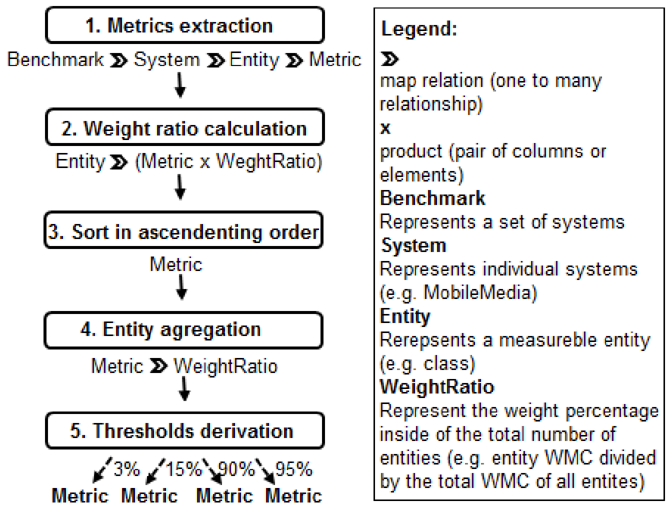
1. Metrics extraction
Benchmark ⪢ System ⪢ Entity ⪢ Metric
2. Weight ratio calculation
Entity ⪢ (Metric x WeghtRatio)
3. Sort in ascendenting order
Metric
4. Entity agregation
Metric ⪢ WeightRatio
5. Thresholds derivation
3%
15%
90%
95%
Metric
Metric
Metric
Metric
Legend:
⪢
map relation (one to many relationship)
x
product (pair of columns or elements)
Benchmark
Represents a set of systems
System
Represents individual systems (e.g. MobileMedia)
Entity
Rerepsents a measureble entity (e.g. class)
WeightRatio
Represent the weight percentage inside of the total number of entities (e.g. entity WMC divided by the total WMC of all entites)

Fig. 1. Summary of the method steps

1. Metrics extraction: metrics are extracted from a *benchmark* of software systems. For each *system*, and for each *entity* belonging to the system (e.g., class), we record a *metric* value. The metric value of each entity of the entire benchmark must be in the same file, such as a spreadsheet. Each column represents a metric and each row represents an entity.

2. Weight ratio calculation: for each entity, we compute the weight percentage within the total number of entities, i.e., we divide the entity weight by the total number of entities, and then it is multiplied by one hundred. All entities have the same weight and the sum of all entities must be 100%. As example, if one benchmark has 10,000 entities, each entity represents 0.01% of the overall (0.01% × 10,000 = 100%).

3. Sort in ascending order: we order the metric values in ascending order and take the maximal metric value that represents 1%, 2%, …, 100%, of the weight. This 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. As example, all entities with WMC value that is 4 must come first that 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%). As example, if we have four entities with WMC value 4 and each entity representing 0.01%; it corresponds to 0.04% of all code.

5. Thresholds derivation: thresholds are derived by choosing the percentage of the overall metric values we want to represent. For instance, to represent 90% of the overall code for the WMC metric, the derived threshold is 18. This threshold is meaningful, since not only it means that it represents 90% of the code of a benchmark of systems, but it also can be used to identify 10% of the worst code in terms of WMC. We trust that it is necessary to have different labels, for this reason the thresholds derived by choosing 3%, 15%, 90% and 95% of the overall metric value, which derive thresholds X1, X2, X3, and X4, respectively. This allows identifying metrics value to be fixed in long-term, medium-term and short-term. Furthermore, these percentiles are used in quality profiles to characterize metrics value according to five categories: *very low* values (between 0-3%), *low* values (3-15%), *moderate* values (15-90%), *high* values (90-95%), *very high* values (95-100%). In section VI, we provide some discussions, such as why we chose these percentages and labels, and why the requirements listed in the beginning of this section should be followed.

IV. Example of Use

The proposed method can be applied in different ways, such using SIG quality model [19], using metrics individually or using a metric-based detection strategy. This section illustrates an example of use of the proposed method in two metric-based detection strategies. For this, before the method be applied, it is necessary, to build or have a benchmark composed by software systems, to choice a set of metrics to derive thresholds, and to choice a tool able to extract these metrics value from each system of the target benchmark.

Section IV.A presents the detection strategies and the set of metrics used in these detection strategies. Section IV.B explains how we built the SPL benchmarks. Section IV.C presents the derived thresholds for the chosen set of metrics (Section II.A) obtained by our method (Section III) using the benchmarks (Section IV.B).

A. Metric-Based Detection Strategies

Despite of the extensive use of metrics, they are often too fine grained to comprehensively quantify deviations from good design principles [21]. In order to overcome this limitation, the metric-based detection strategies were proposed [25]. Detection strategies is a composed logical condition, based on metrics and threshold values, which detects design fragments with specific code smells [21]. Code smells describe a situation where there are hints that suggest a flaw in the source code [31]. This section illustrates the detection strategies of two code smells: God Class and Lazy Class.

God Class is defined as a class that knows or does too much in the software system [16]. 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 defined as a class that knows or does too little in the software system [16]. As can be seen by definition Lazy Class is the opposite of God Class.

In this work, we selected detection strategies in the literature to identify God Classes [1] and Lazy Classes [27] for the following reasons. First, they have been evaluated in other studies and presented good results for the detection of God Class and/or Lazy Class [1][2][36]. Second, these detection strategies use a straightforward way for identifying instances of God Class and Lazy Class using 4 different metrics. We also believe that these strategies are better than traditional ones because they were adapted for SPL by using NCR (a FOP-specific metric), for example. This metric is able to fit complexity properties of SPLs that traditional metrics cannot fit.

Figure 2 shows the God Class and Lazy Class detection strategies adapted from [1] and [27], respectively. LOC, CBO, WMC, and NCR refer to the metrics used in these detection strategies (presented in Section II.A). In original detection strategies can be found absolute values. To try providing strategies more dependent of the derived thresholds, we substitute absolute values by labels, such as *Low*.

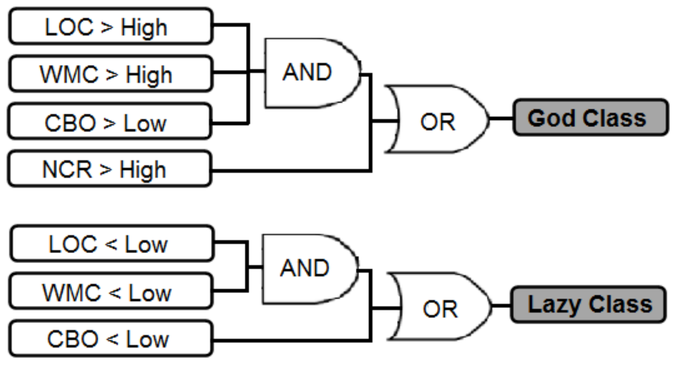

Fig. 2. Code Smells Detection Strategy

B. Software Product Lines Benchmarks

This section presents three benchmarks of Software Product Lines (SPLs). To build these benchmarks, we focus on SPLs developed using FOP [5]. The main reason for choosing FOP is because this technique aims to support modularization of features - i.e., the building blocks of a SPL. In addition, we have already developed a tool, named *Variability Smell Detection* (VSD) [2], which is able to measure FOP code (step one of the proposed method).

We selected 47 SPLs from repositories, such as SPL2go [34] and FeatureIDE examples [12], and 17 SPLs from research papers; summing up to 64 SPLs in total. In order to have access to the SPLs source code, we either email the paper authors 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 [12] has 4 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 selected either the AHEAD or FeatureHouse implementation. 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 I. The source code of all SPLs of our benchmark and the step-to-step filtering is further explained on the project website [35].

TABLE I. TABLE TYPE STYLES

	Id	SPL	Tech.	LOC
Benchmark 1, 2, and 3	1	BerkeleyDB [34]	FH-Java	37247
	2	AHEAD-Java [1]	AHEAD	16719
	3	AHEAD-guidsl [1]	AHEAD	8738
	4	TankWar [12], [34]	AHEAD	4670
	5	AHEAD-Bali [1]	AHEAD	3988
	6	Devolution [12]	AHEAD	3913
	7	MobileMedia v.7 [14]	AHEAD	2691
	8	WebStore v.6 [14]	AHEAD	2082
	9	DesktopSearcher [12], [34]	AHEAD	1858
	10	GPL [12]	AHEAD	1824
	11	Notepad v.2 [34]	FH-Java	1667
	12	Vistex [34]	FH-Java	1480
	13	GameOfLife [34]	FH-Java	1047
	14	Prop4J [34]	FH-Java	1047
Benchmarks 1 and 2	15	Elevator [34]	FH-Java	728
	16	ExamDB [34]	FH-JML	568
	17	PokerSPL [34]	FH-JML	461
	18	EmailSystem [34]	FH-Java	460
	19	GPLscratch [34]	FH-JML	405
	20	Digraph [34]	FH-JML	374
	21	MinePump [34]	FH-JML	367
	22	Paycard [34]	FH-JML	319
Benchmark 1	23	IntegerSet [34]	FH-JML	225
	24	UnionFind [34]	FH-JML	194
	25	NumberContractOverrinding [34]	FH-JML	165
	26	NumberConsecutiveContractRef [34]	FH-JML	148
	27	NumberExplicitContractRef [34]	FH-JML	143
	28	BankAccount [34]	FH-JML	122
	29	EPL [12]	AHEAD	98
	30	IntList [34]	FH-JML	94
	31	StringMatcher [34]	FH-JML	45
	32	Stack [34]	FH-Java	22
	33	HelloWorld [12]	AHEAD	22

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 I presents the 33 SPLs ordered by their value of LOC, implementation technology (Tech.) and grouped by their 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 1,000 LOC. The goal of creating three different benchmarks is to analyze the results with varying levels of thresholds.

C. Derived Thresholds

This section presents the derived thresholds that were obtained using the proposed method to derive thresholds (Section III) according to each benchmark (Section IV.B). The process was performed with the four metrics used in this study. Only the key values of the proposed method are presented. For example, the proposed method presents five labels, but these labels are established in four percentages. Hence, Table II shows just the values that represent the percentages. 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, 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%) that are represented by the intervals 0-2; 3-4; 5-77; 78-138 and >139, respectively for LOC in benchmark 1.

TABLE II. THRESHOLDS VALUES FROM THE PROPOSED METHOD

Benchmark	%	LOC	CBO	WMC	NCR
1	3	3	2	1	1
	15	5	2	2	1
	90	78	12	18	4
	95	139	17	32	8
2	3	3	2	1	1
	15	5	2	2	1
	90	79	12	18	4
	95	143	17	33	9
3	3	3	2	1	1
	15	5	2	2	1
	90	80	13	19	4
	95	147	18	35	9

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

V. EVALUATION

This section evaluates the derived thresholds from the proposed method and the method proposed by Lanza and Marinescu, called in this study as Lanza's method [21]. To perform this evaluation, we (i) derive thresholds using Lanza's method [21] (Section V.A), (ii) choose a target SPL (Section V.B), (iii) define the oracle of code smells (Section V.C), and (iv) perform the comparison of effectiveness of these two methods (Section V.D).

A. Derived Thresholds defined by Lanza's Method

Despite of Lanza's method uses mean and standard derivation (see Section II.C), we decide to choose this method because it was used in a previous work [1] to derive thresholds for SPLs. In other words, this is the unique method to derive thresholds found in the literature to the context of SPLs. As can be seen in Figure 2 some labels are used, such as high and low. These labels are calculated by following the Lanza's method [21]. For instance, $\mu + \sigma$, μ, and $\mu - \sigma$ are used to high, mean, and low labels, respectively, where μ is the mean and σ is the standard deviation (see Section II.C). Abilio et al. [1] built a SPL benchmark to derive thresholds, but, their benchmark is composed only by eight SPLs which are also part of our benchmark. For this reason, we considered that our benchmark extends their benchmark. Hence, to provide the more representative thresholds for the labels, we calculate them from our benchmarks. Table III presents the thresholds necessary to represent their labels. 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 are defined as: *low*, *mean*, and *high* that are represented by the intervals 0, 5.33, and 11.7, respectively for CBO and benchmark 1.

TABLE III. THRESHOLDS VALUES FROM LANZA'S METHOD

Benchmark	Label	LOC	CBO	WMC	NCR
1	Low	0	0	0	0
	Mean	36.8	5.33	8.14	1.07
	High	126	11.7	28.9	4.05
2	Low	0	0	0	0
	Mean	37.4	5.44	8.3	1.07
	High	128	11.8	29.4	4.08
3	Low	0	0	0	0
	Mean	37.8	5.59	8.4	1.14
	High	130	12.1	29.9	4.27

B. Choosing the Target SPL

We choose a SPL, called MobileMedia [15], which is a SPL for manipulating photos, music, and videos on mobile devices [15]. It is an open source SPL implemented in several programming languages, such as Java, AspectJ, and AHEAD. We selected MobileMedia version 7 - AHEAD implementation [14]. This SPL was chosen because: (i) it was successfully used in other previous empirical studies [14][15][30], (ii) it is part of the three benchmarks of this study, and (iii) we have access to its software developers.

C. Oracle of Code Smells

The oracle can be understood as the reference model of the actual smells found in a SPL. The reference model is used for evaluating methods to derive thresholds. In particular, the oracle is the basis for determining whether the derived thresholds (computed by both methods) are effective on identification of code smells in a specific SPL. In order to provide a reliable oracle, we analyze the source code and we defined some God Class and Lazy Class instances. This preliminary oracle has been validated by experts, and the final version of the oracle was produced as a joint decision. Table IV presents the final version of the oracle that includes seven God Class instances and ten Lazy Class instances.

D. Evaluation of the Derived Thresholds

This section presents the results of the evaluation of recall and precision applied in the derived thresholds of the proposed method (Section III) and Lanza's method (Section V.A) using the MobileMedia SPL (Section V.B) supported by the oracle of code smells (Section V.C). It is important to mention that each method was used with the same detection strategy for each code smell (Section IV.A) and it was applied to the three benchmarks (Section IV.B).

TABLE IV. CODE SMELL ORACLE FOR MOBILEMEDIA

Code Smell	Classes in the Oracle
God Class	*MainUIMidlet (Base), MediaAccessor (Base), MediaController (MediaManagement), MediaListController (MediaManagement), MediaListScreen (MediaManagement), AlbumData (AlbumManagement),* and *SmsMessaging (SMSTransfer)*
Lazy Class	*Constants (AlbumManagement), MediaData (SetFavourites), ControllerCommandInterface (Base), ControllerInterface (Base), Constants (Base), PhotoViewController (CaptureVideo), Constants (CreateAlbum), Constants (CreateMedia), Constants (DeleteAlbum),* and *Constants (MediaManagement)*

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

Table V describes the results per method, summarizing the true positives (TP), false positives (FP), and false negatives (FN). 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 the benchmarks 1, 2, and 3. For instance, by using the thresholds derived by the proposed method in benchmarks 1, 2, and 3 the number of TP for God Class candidates) was 3. In all cases, one FP was found. The derived thresholds for both, Lanza's method and ours find the same values for TP, FP, and FN metrics to the three benchmarks, in spite to be a difference in some values. For instance, for benchmarks 1 and 3 Lanza's method finds 126 and 130 to LOC, respectively (see Table III).

TABLE V. IDENTIFICATION OF THE SELECTED CODE SMELLS BASED ON THRESHOLDS DERIVED FROM EACH METHOD

Code Smell	#	The Proposed Method			Lanza's Method		
		Benchmarks			Benchmarks		
		1	2	3	1	2	3
God Class	TP	3	3	3	3	3	3
	FP	1	1	1	1	1	1
	FN	4	4	4	4	4	4
Lazy Class	TP	10	10	10	1	1	1
	FP	11	11	11	0	0	0
	FN	0	0	0	9	9	9

Aiming to provide an additional perspective of the effectiveness on identification of code smells, we also analyzed precision and recall measures. Recall (R) quantifies the rate of TP by the number of e*xisting code anomalies* (TP + FN). Precision (P) quantifies the rate of TP by the number of *detected code anomalies* (TP + FP). Table VI presents recall and precision of the detection strategies applied to MobileMedia using the derived thresholds for both methods.

TABLE VI. RECALL AND PRECISION BASED ON THRESHOLDS DERIVED FROM EACH METHOD

Code Smell	Benchmarks	The Proposed Method		Lanza's Method	
		P	R	P	R
God Class	1, 2, and 3	75.00	42.86	75.00	42.86
Lazy Class	1, 2, and 3	47.62	100	100	10.00

We can observe that for the detection strategy to identify God Class instances the precision and recall are the same for the methods, 75% and 42.86%. For the detection strategy that aims to identify Lazy Class instances the values of recall and precision are different. The proposed method have lower values of precision (47.62%) and higher recall (100%) when compared with the Lanza's method (100% of precision and 10% of recall). In addition, recall is considered more useful than precision in the context of identification of code smells as recall is a measure of completeness [30]. That is, high recall means that the detection strategy was able to identify a high number of code smells in software. Then, the proposed method fared better in the evaluation.

VI. Discussion

This section presents discussions related to some choices of our work. In a previous work [36], 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 with the number of systems; (iv) it should be a strong dependence with the number entities; (v) it should not to correlate 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 the introduction. Based in these desirable points we listed seven requirements that the proposed method follows (Section III). Those requirements and desirable points are explained in the following five sections.

A. To be a benchmark-based method

The first question that comes in my mind is: why to use a benchmark-based method? Derive threshold from a benchmark data is more confident than derived thresholds from a unique system. Doing an analogy, it is the same in which we ask for an expert the threshold of a metric and ask for many experts the same thing. The second one should be more confident, because we have more information and different opinions. Because of that the proposed method is based on a benchmark data.

B. To be a strong dependence with the number of entities and to be a weak dependence with the number of systems

We want to know the thresholds of a metric that is represented by a set of entities. Hence, the derived thresholds should be based by 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 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 thresholds 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, the proposed method considers the number of entities explicitly. The number of systems is a consequence (implicitly) of the number of entities. We trust that if we have a representative number of entities we will have high probably to be a representative number of systems.

Our benchmarks have at least 14 SPLs and 2,450 entities. We assume that this number of SPLs and entities are representative, because there are few open source code Feature-Oriented SPLs that use compositional approach in the literature, but in another context it can be a non-representative number of systems, such as in Object-Oriented systems, because is easy to find more than 100 systems. We want to explore this topic in future work.

C. 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 Figure 2). For this reason, the proposed method derives upper and lower thresholds.

Step-wise format can help separate outliers, for example, very high values of high values. Of course, if we have a threshold the highest values, in the case of upper thresholds, have more chance to be a problem and these values can be threated firstly. But we want to talk with another perspective. In the case of detection strategies, a metric can be more important than other ones and this metric should be highlight in the detection strategy. One way to do that is to evaluate this metric separate, 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 on Section IV.A (Figure 2). 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 attach here is the name of labels. We called the labels of the proposed method as: very low, low, moderate, high, and very high values. We believe that a very low value can be bad and a value high can be also bad, it depends on the context and it does not means that a very low value is always good or always bad. For this reason, we preferred not to call the labels relating with risks or bad design.

D. Respect the statistical properties of the metric

The thresholds are derived to find outliers in a system; if the statistical properties of metrics are changed the derived thresholds can be finding wrong outliers. Hence, it is believed that a good method should analyze the metric distributions without change anything in their statistical properties. It implies in not correlate metrics nor weight 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 its Probability Density Function (PDF). Figure 3 depicts the distribution of the CBO and WMC values for the benchmarks 3, using a PDF. The x-axis represents the CBO and WMC values (ranging from 0 to 66 to CBO and from 0 to 383 to WMC) and the y-axis represents percentage of observations (percentage of entities). The use of the PDF is justifiable, because we want to determine thresholds (the dependent variables, in this case the CBO and WMC values) as a function of the percentage of observations (independent variable). Also, by using the percentage of observations instead of the frequency, the scale becomes independent of size of the benchmark making it possible to compare different distributions [3]. In Figure 3a we can observe that 68.35% of entities have a CBO value <=5. In Figure 3b and supported by Table II, we can observe that 90% of entities have a WMC value < 19. But, after these two points, the metrics values come to increase quickly. For example, 95% of entities have a WMC value < 35. Looking at first time, the labels high (90%) and very high (95%) of the proposed method looks too much rigid, but as can be seen, it is not.

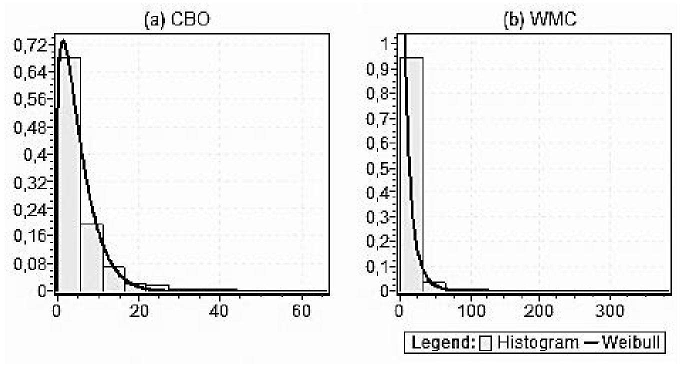

Fig. 3. Probably Density Function

On the other hand, if we listed low percentages of CBO values of 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. Similar thing happens with WMC values of benchmark 3, the values for the same 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 have 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 is possible to see that very low label should be stronger than very high and distant to low label. We considered 1% very rigid; hence, we choose 3% for very low label. On the other hand, we choose 15% for low label to be a greater difference in terms of percentages.

Another point that we want to highlight here is that Lanza's Method assumes that metrics follow a normal distribution, as can be seen in the examples of CBO and WMC it is not always true. The metric distributions do not follow a normal distribution impacts in the low labels (see Table III) for all metrics and benchmarks the values were 0. This happened because the standard variation is higher than mean (low label is calculated by mean minus standard derivation) and, it does not make sense the set of metrics chosen have negative values.

Additionally, several studies show that different software metrics follow heavy-tailed distribution [3][9][24][36]. But, if a metric does not follow a heavy-tailed distribution, are the derived thresholds from the proposed method valid? The proposed method takes account the metric distributions focused in to identify outliers. Hence, if the metric follows a normal distribution or a common distribution, for example, the outliers will be identified using the proposed labels. As a concrete example, Figure 4 presents the DIT (Degree of Inherence Tree) metric from the benchmark of Ferreira et al. [13] in which have a common distribution (the common value is 1). Probably, with the proposed method the high and very high values for this metric would be 2 or higher, that is a value above the common (above than 1), and the low value would be 0 or 1; it is a value bellow or equal to the common. In other words, we are only identifying discrepant values (outliers) related to the provided data (benchmark).

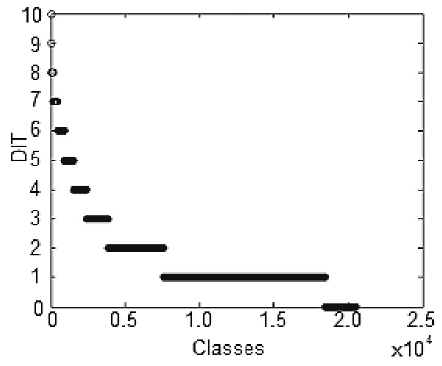

Fig. 4. Probably Density Function [13]

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

E. A method systematic, repeatable, transparent, and straightforward

These are qualities interesting for any method. The method proposed is considered systematic because it has five well-defined steps. If the steps are followed the same results will be obtained. Hence, the method is repeatable and transparent. As can be better explained with previous topics we do not change any statistical properties of the target metric to derive thresholds and all entities are equally weighted. Wherefore, this method is straightforward. We developed a prototype of tool to support the steps of the proposed method, however, this prototype has no interface (it works in command line) and we want to add some functions such as plot graphs of the metric distributions. In addition, the tool can reduce the chance of manual errors. We want to propose a tool to automatize the steps of the proposed method in future work.

VII. 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. As well as 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 derive a repository with a higher number of SPLs. This limitation has implication in the amount 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. 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 changing the composer of FH to the composer of AHEAD. There are reports in the literature justifying this transformation preserves all properties of FH [4]. We also reduced possible threats by performing some tests with a 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. But, to try generalize and provides default labels we decide to use these percentages. In addition, can be seen that very low and low labels should be equals or similar values, in spite of that we prefer 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 it someone think that these values do not fit well in their metric distribution another values can be used.

Code smell: We discuss only two types of code smell (i.e. God Classes and Lazy Classes). Fowler has cataloged a list with more than twenty code smells [16]. Therefore, these smells used to evaluate the effectiveness of both methods (Lanza and our 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 low label of the target metric. It can be affected the evaluation, but this choice affected both our method and Lanza's method. 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 scoping. Different tools implement different variations of the same metrics [3]. 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 [3]. On the other hand, the existence of generated code, which normally has high complexity, may result in higher threshold values [3]. As previously stated, for deriving thresholds we removed all useless code from our analysis.

Oracle Generation – An oracle for each code smell had to be defined in order to calculate recall and precision measures. Several precautions were taken. In spite of 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 oracle.

VIII. Conclusion and Future Works

This paper describes the importance of software metrics, the use of a set of metrics to measure some quality attribute and, the calculation of representative thresholds. In order to focus in the last point, a method to derive thresholds were proposed, described and compared (with a method with the same purpose) using as input data metrics collected from benchmarks composed by SPLs. We believe that the proposed method is systematic, repeatable, transparent and straightforward to execute.

To perform the evaluation it was necessary three different benchmarks, two different code smells (God Class and Lazy Class), and a method to derive thresholds to be used as baseline (Lanza's method). Hence, we created a repository with 64 SPLs, in spite of that only 33 SPLs were used (benchmark 1) following some restrictions, such as we picked up only the most recent one. In addition, we applied two refinements to extract the benchmark 2 and 3. These two refinements consist in keep only SPLs with more than 300 and 1,000 LOC, respectively. The proposed method and Lanza's method were evaluated in terms of recall and precision on the identification of the two code smells chosen.

As results, the proposed method provides thresholds that can be used in different contexts such as SIG model and to identify code smells. In comparison to Lanza's method, the proposed method presents equal and better recall for the identification of God Class and Lazy Class instances, respectively. Our method fared 100% of recall to Lazy Class detection strategy. The discussion relate many different topics, such as, the justification of the steps of the proposed method and why a method should be benchmark-based, respect the statistical properties of metrics, and to be a strong dependence with the number of entities. The topic related to statistical properties (Section VI.D) help us to justify many decisions such as the percentages chosen and why we trust that the proposed method can be applied in metrics with different distribution.

After all comparison and analyzes can be said that our method fits seven requirements: i) it is based on data analysis from a representative set of systems (benchmark); ii) it has a strong dependence with the number of entities; iii) it has a weak dependence with the number of system; 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. As future work, we want explore how to build representative benchmarks and provide a tool to run the proposed method (to help prevent possible manual mistakes in the method execution).

Acknowledgments

This work was partially supported by CAPES, CNPq (grant 485907/2013-5), and FAPEMIG (grant PPM-00382-14).

References

[1] R. Abilio, J. Padilha, E. Figueiredo, and H. Costa, "Detecting Code Smells in Software Product Lines - An Exploratory Study". In proceedings of 12th International Conference on Information Technology: New Generations, 2015.

[2] R. Abilio, G. Vale, J. Oliveira, E. Figueiredo, and H. Costa, "Code Smell Detection Tool for Compositional-based Software Product Lines". In Proceedings of 21th Brazilian Conference on Software, Tools Session, pp. 109-116, 2014.

[3] T.L. Alves, C. Ypma, and J. Visser, "Deriving Metric Thresholds From Benchmark Data". In Proc. of 26th Int. Conf. on Software Maintenance (ICSM), pp. 1–10, 2010.

[4] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-Independent, Automated Software Composition". In Proc. of the Int. Conf. on Soft. Eng., pp. 221–231, 2009.

[5] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchial Software Systems with Reusable Components", ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 1, Issue 4, pp. 335-398, October, 1992.

[6] P. Brereton, B. Kitchenham, D. Budgen, M. Tumer, and M. Khalil, "Lessons From Applying the Systematic Literature Review Process within the Software Engineering Domain", J. of Systems and Software, Vol. 80/4, pp. 571-583, April, 2007.

[7] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, vol. 20, Issue 6, pp. 476–493, June, 1994.

[8] D. Coleman, B. Lowther and P. Oman, "The Application of Software Maintainability Models in Industrial Software Systems". Jour. on Syst. Soft., vol. 29, pp. 3–16, Feb., 1995.

[9] G. Concas, M. Marchesi, S. Pinna and N. Serra, "Power-Laws in a Large Object-Oriented Software System". IEEE Transactions on Soft. Engineering, vol. 33, Issue 10, pp. 687–708, Oct., 2007.

[10] R.R. Dumke, and A.S. Winkler, "Managing the Component-Based Software Engineering with Metrics". In proceedings of International Symposium on Assessment of Software Tools and Technologies, pp.104-110, 1997.

[11] K. Erni and C. Lewerentz, "Applying Design-Metrics to Object-Oriented Frameworks". In Proceedings of the 3rd International Symposium on Soft. Metrics. pp. 64-72, 1996.

[12] FeatureIDE – Available in: <http://www.iti.cs.uni-magdeburg.de/iti_db/research/featureide/>. Access in 04/2014.

[13] K. Ferreira, M. Bigonha, R. Bigonha, L. Mendes and H. Almeida, "Identifying Thresholds for Object-Oriented Software Metrics". Journal of Systems and Software, vol. 85, Issue 2, pp. 244–257, February, 2012.

[14] G.C. Ferreira, F.N. Gaia, E. Figueiredo, and M.A. Maia, "On the Use of Feature-Oriented Programming for Evolving Software Product Lines – A Comparative Study". Science of Computer Prog., Vol. 93, pp. 65-85, November, 2014.

[15] E. Figueiredo, *et al., "*Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability". In Proc. of the Int. Conf. on Soft. Eng. (ICSE), pp. 261-270, 2008.

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, "Refactoring: Improving the Design of Existing Code". Addison-Wesley Professional, 1999.

[17] V.A. French, "Establishing Software Metric Thresholds". In Proc. of the Int. Work. on Soft. Measurement (IWSM'99), 1999.

[18] E. Gamma, R. Helm, R., Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.

[19] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," Int. Conf. on the Quality of Inf. and Communications Tech. (QUATIC'07), pp. 30–39, 2007.

[20] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering". EBSE Technical Report, Keele University, 2007.

[21] M. Lanza and R. Marinescu, "Object-Oriented Metrics Practice", Springer, p. 205, 2006.

[22] F. Loesch and E. Ploedereder, "Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations". In proc. of Int. Conference on Software Maintenance and Reengineering (CSMR), pp. 159-170, 2007.

[23] M. Lorenz and J. Kidd, "Object-oriented software metrics". New York: Prentice Hall, p. 146, 1994.

[24] P. Louridas, D. Spinellis and V. Vlachos, "Power Laws in Software". ACM Transactions on Soft. Eng. and Methodology (TOSEM), Vol. 18, Issue 1, Article Nr. 2, September, 2008.

[25] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws". In: Proceedings of the International Conference on Software Maintainability, pp. 350-359, 2004.

[26] T.J. McCabe, "A Complexity Measure", IEEE Transactions on Soft. Engineering, Vol. SE-2, Issue 4, pp. 308–320, Dec., 1976.

[27] M. J. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code", Proceeding of 11th IEEE Int. Sof. Metrics Symposium (METRICS), IEEE Computer Society Press, 2005.

[28] B.A. Nejmeh, "NPATH: A Measure of Execution Path Complexity and its Applications". Magazine Communications of the ACM, vol. 31, Issue 2, pp. 188–200, February, 1988.

[29] P. Oliveira, M. Valente and F. Lima, "Extracting Relative Thresholds for Source Code Metrics". In proc. of the Int. Conf. on Soft. Maintenance, and Reeng. (CSMR), pp.254-263, 2014.

[30] J. Padilha, J. Pereira, E. Figueiredo, J. Almeida, A. Garcia and C. Sant'Anna, "On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study". In Proc. of 26th Int. Conf. on Advanced Inf. Syst. Eng. (CAISE), pp. 656-671, 2014.

[31] J. Riel, "Object-Oriented Design Heuristics". Addison-Wesley Professional, p. 400, 1996.

[32] Software Engineering Institute. Software product line. Available in: <http://www.sei.cmu.edu/productlines/>. Access in 04/2014.

[33] D. Spinellis, "A Tale of Four Kernels". In Proc. of the Int. Conf. on Software Engineering (ICSE), pp. 381–390, 2008.

[34] SPL2GO - In: <http://spl2go.cs.ovgu.de/>. Access in 04/2014.

[35] SPL Metric Thresholds – Available in: <http://labsoft.dcc.ufmg.br/doku.php?id=%20about:spl_list>. Access in 04/2014.

[36] G. Vale, D. Albuquerque, E. Figueiredo, A. Garcia, "Defining Metric Thresholds for Software Product Lines: A Comparative Study". In: 19th Int. Software Product Line Conf. (SPLC), 2015.

[37] R. Vasa, M. Lumpe, P. Branch and O. Nierstrasz, "Comparative Analysis of Evolving Software Systems Using the Gini Coeficient". In Proceedings of the International Conf. on Soft. Maintenance (ICSM), pp. 179–188, 2009.